COHERENT™

A Multi-User, Multi-Tasking Operating System
for IBM PC Compatibles.

Mark Williams
Company

Revision 11 Printing 5 4 3 2 1

Published by Mark Williams Company, 60 Revere Drive, Northbrook, Illinois 60062.

Sales: Phone: (800) 636-6700 (US) — (708) 291-6700 (outside United States)
FAX: (708) 291-6750
E-mail: sales@mwc.com
Technical Support:
Phone: (708) 291-6700
FAX: (708) 291-6750
E-mail: support@mwc.com
CompuServ: 76256,427

This manual was written under the COHERENT operating system, using the MicroEMACS text editor. Text formatting was performed by the COHERENT edition of the **troff** text formatter, generating PostScript. Page design was implemented with custom-written macros written in the **troff** text-formatting language and in PostScript. Capitals and ornaments are derived from the Paris Book of the Hours (1510), and were supplied in encapsulated PostScript form by BBL Typographic, 137 Narrow Neck Rd., Katoomba, NSW 2780, Australia (bblart@peg.apc.org). The key-caps font was supplied by SoftMaker, Inc., 2195 Faraday Avenue, Suite A, Carlsbad, CA 92008. Diagrams were drawn under the COHERENT implementation of X, using the program **xfig**, which was originally written by Supoj Sutanthavibul of the University of Texas at Austin. Typesetting of this manual, from the table of contents through the index, was executed by one script written in the COHERENT Bourne shell. Camera-ready copy was printed on a Hewlett-Packard LaserJet IIP printer using the Pacific Page PostScript cartridge.

Table of Contents

Preface

COHERENT is the work of a large number of exceptionally talented people. The development of a multi-user, multi-tasking operating system is a daunting task. Creating COHERENT took an enormous effort by all involved. The system and manual are dedicated to those who dedicated themselves to COHERENT.

These people include the following:

Jay Alter
Bob Beals
Luddyne Blue
Nigel Bree
Henry Cejtin
Roger Critchlow
Stephen Davis
Tom Duff
Charles Fiterman
Johann George
Michael Griffin
Scott Hermes
Owen Jacobsen
Nancy Kenston
Irene Lee
Jeanne Lewis
Udo Munk
Alex Nash
Steve Ness
Frank Pfeiffer
Abner Snell III
Michael Spertus
Angus Telfer
Rico Tudor
La Monte Yarroll
Riyaz Asaria
James Behr
Barry Bowen
Denise Buirge
David Conroy
Richard Critchlow
John R. Dennison
Mark Epstein
Charles Forsyth
Louis J. Giliberto, Jr.
Walter Grogan
Chris Hilton
Mary Karabatsos
J.T. Kittridge
James Leonard
Karen McBride
Esther Munoz
Asia Negron
Ciaran O'Donnell
Norma Reyes
Addison Snell
Julie Stewart
Trevor Thompson
Bernard Wald
Jim Yonan
Norman Bartek
Chris Berrios
Eduardo Bravo
Fred Butzen
Allan Cornish
Ella Dashevsky
Mimi Diaz
Michael Farley
Kim Fruin
Daniel Glasser
Robert Hemedinger
Randall Howard
Michael Kaufman
William Lederer
Dave Levine
Scott Moody
Tim Murphy
Gerson Negron
Douglas Peterson
Vladimir Smelyansky
Hal Snyder
Robert Swartz
Diane Tracey
Bill Witt

The "port" of the COHERENT kernel to the 80386 was implemented using software provided by Ciaran O'Donnell, Bievres, France.

Introduction

COHERENT is a professional operating system designed for use on machines that can run MS-DOS. It has many of the features and functionality of the UNIX operating system, but is the creation of Mark Williams Company. COHERENT gives your computer multi-tasking, multi-user capabilities without the tremendous overhead, both in hardware and money, required by current editions of UNIX. COHERENT is what UNIX used to be: a well-designed system with selected tools and well-designed utilities that bring out the best in modest computer systems.

The COHERENT system consists of the following:

- A fully multi-tasking, multi-user kernel.
- Choice of Bourne or Korn shells.
- The Mark Williams C compiler, linker, assembler, archiver, and other tools.
- A suite of commands, including editors, languages, tools, and utilities.
- Drivers for peripheral devices, including terminals; ASCII, PostScript, and PCL printers; dumb serial cards; and tape backup.
- Libraries, including the standard C library, the mathematics library, and libraries for **curses** and socket emulation.
- Numerous tools, utilities, and games.

For a list of some third-party programs that you can run under COHERENT, see the release notes that accompany this manual. New programs are released regularly, so consult the Mark Williams Bulletin Board for the latest information.

What Is COHERENT?

COHERENT is a multiuser, multitasking operating system. *Multiuser* means that with COHERENT, more than one person can use your computer at any given time. *Multitasking* means that with COHERENT, any user can run more than one program at any given time. The design of COHERENT employs a few elegant concepts to give you a powerful and flexible system that is easy to use.

What is an Operating System?

An *operating system* is the master program that controls the operation of all other programs. It loads programs into memory, controls their execution, and controls a program's access to peripheral devices, such as printers, modems, and terminals.

Some operating systems (e.g., MS-DOS) permit only one user to use the computer at a time; and that user can run only one program at a time. However, you may well want your computer to support more than one user at a time, and run more than one program simultaneously. Sharing not only yields many economies (such as allowing a group of users to share one printer), but also allows the users to communicate with each other and so work together more efficiently.

Any multitasking operating system must be able to do the following tasks efficiently:

- Schedule computer time
- Control mass-storage devices (disks and tape drives)
- Organize disk-storage space
- Protect programs from conflict
- Protect stored information from destruction
- Ease cooperation among users

Today's operating systems also provide *tools*. These are programs that are bundled with the operating system, and that are designed to help you do your work more efficiently. For example, you need editors, compilers, debuggers, and assemblers to develop and test programs. Text formatters and spelling checkers help you write memoranda, manuals, or books. Command processors (also called *shells*) help you run the computer easily. Status checkers tell you what programs are being run, who is using the system, and how much space is left on your disk.

2 Introduction

The combination of operating system and its tools transforms a boxful of wires and circuits into a useful machine.

Design Philosophy

A computer system is not an end in itself; rather, it is a "bench" for constructing tools to solve specific problems. If the operating system is too specialized or limited, the range of problems it can help you solve will be narrow. On the other hand, if the operating system is too detailed, then it becomes complex, idiosyncratic, and potentially unreliable.

The following quotation from John Conway summarizes well the philosophy that underlies the design of the COHERENT system:

> *The engineer who wants a machine for some specific purpose will normally approve the simplest machine that does the job. He will not usually prefer a multiplicity of parts with the same effect, nor will he countenance the insertion of components with no function.*

The COHERENT system follows this approach throughout. In brief, COHERENT is what UNIX used to be: an efficient system of selected tools and well-designed utilities, that brings out the best in your computer system.

Installation

The release notes that come with COHERENT describe how to install COHERENT. The release notes also list hardware that is known to work with COHERENT, and hardware that is known *not* to work with COHERENT. Before you begin to install COHERENT on your system, be sure to check those lists and make sure that your system is compatible with COHERENT.

Please note that Mark Williams Company tries to keep these lists up to date, but it is not possible to keep pace with the continual introduction of new machines and new models. If you do not find your machine on either list, the odds are that COHERENT will work correctly with it.

User Registration and Reaction Report

Before you continue, fill out the User Registration Card that came with your copy of COHERENT. When you return this card, you become eligible for direct telephone support from the Mark Williams Company technical staff, and you will automatically receive information about all new releases and updates.

If you have comments or reactions to the COHERENT software or documentation, please fill out and mail the User Reaction Report included at the end of the manual. We especially wish to know if you found errors in this manual. Mark Williams Company needs your comments to continue to improve COHERENT.

Technical Support

Mark Williams Company provides free technical support to all registered users of COHERENT. If you are experiencing difficulties with COHERENT, outside the area of programming errors, feel free to contact the Mark Williams Technical Support Staff. You can telephone, send electronic mail, or write. Please note that this support is available *only* if you have returned your User Registration Card for COHERENT.

Before you contact Mark Williams Technical Support with your problem, *please check this manual first.* If you do not find an article in the Lexicon that addresses your problem, be sure to check the index at the back of the manual. Often, the information that you want is kept in an article that you didn't consider, and the index will point you to it.

Another good way to find a topic in the manual is to use the command **apropos**, which is part of the COHERENT system. **apropos** finds every article in the Lexicon that mentions a given term or phrase. For details on how to use this command, see its entry in the Lexicon.

If the manual does not solve your problem — or if you find it to be misleading or difficult to understand — then Mark Williams Technical Support is available to help you. You can reach Technical Support via any number of routes:

Electronic Mail
: If you have access to the Internet, send mail to **support@mwc.com**. This is the preferred means of communication. Be sure to include your surface address and telephone number as well as your e-mail address, so we can contact you even if return electronic mail fails.

FAX Send your technical FAXes to 1-708-291-6750.

Surface Mail

Write to Technical Support, Mark Williams Company, 60 Revere Drive, Northbrook, IL 60062.

Telephone

To contact Technical Support via telephone, call 1-708-291-6700, between 9 AM and 5 PM, Central Time. Please have at hand your manual for COHERENT, as well as your serial number and version number. Please collect as much information as you can concerning your difficulty before you call. If possible, call while you are at your machine, so the technical support person can walk you through your problem.

Help Us Help You

Mark Williams Technical Support wants to help you fix your problem as quickly as possible, so you can enjoy your COHERENT system. You can help us to help you by doing the following:

Before you contact Technical Support, *write down* as carefully as possible what you did that triggered the problem. Copy down exactly any error messages that appeared on the screen.

If the problem is triggered by a script or program, try to edit the script or program to the chunk of code that triggers the problem. The smaller the chunk of code, the better.

In your message, please include the following information:

- The make of your computer, and the type and clock speed of its microprocessor.
- The amount of RAM that you have.
- The size and make of your hard disk, and the make of its controller.
- If the problem affects the video display, include the make of your display (i.e., tube) and controller card.

If you have found an error in the manual, please mention the page on which the error occurs.

This information will help us to to clear up your problem as quickly as possible.

How To Use This Manual

COHERENT encompasses an entire world of computing. Before you learn the signposts of this world, you may find it difficult to perform even simple tasks; and you may feel confusion and frustration.

COHERENT, however, more than anything else, consists of tools; and this manual is one of the most useful tools that your COHERENT system has. It is designed to guide you through the COHERENT system, to answer your questions, and to lead you into areas of the system that may never have explored on your own. If you can learn to use it effectively, you will both lower the amount of frustration you will have to endure, and increase your productivity.

Beginners, in particular, should look over this manual carefully. The following sub-sections describe the steps a beginner should take when she begins to work with this manual.

Elementary Tutorials

The following tutorials teach the essentials of COHERENT: they teach both essential information and essential skills. Every beginner should work through these tutorials:

- First, read the tutorial *Using the COHERENT System*. This tutorial will introduce the COHERENT system and its tools. It also teaches you such important tasks as how to shut the system down properly (hint — *never* just shutting the machine off!) and how to boot it again.
- Then look at the subsequent tutorial, *Introducing sh, the Bourne Shell*. The shell is the program through which you give commands to your COHERENT system; and it incorporates a powerful programming language of its own. Some of this information will seem obscure to you; but the as with any language, the more you use the shell's programming language the more quickly will you acquire fluency.
- You should then look into learning how to use a text editor under COHERENT. You will need to use a text editor in order to write scripts, programs, and documents, which is the heart of using any information-processing tool like COHERENT.

  COHERENT offers two screen-oriented text editors: MicroEMACS, and **vi**. The manual has a tutorial on MicroEMACS. You will find it to be fairly easy to learn how to perform simple text editing with MicroEMACS.

Advanced Tutorials

The above tutorials will teach you the rudiments of how to use COHERENT. The next set of tutorials introduce some of COHERENT's tools and languages. These tutorials are rather specialized. A beginner should look only at the ones that interest her:

Introduction to the ed Line Editor
: **ed** is an editor from the early days of UNIX. It is line oriented, which means that you edit text by typing commands rather than by moving a cursor around the screen. **ed** is powerful and useful tool, but is of limited appeal to most beginners.

Introduction to the sed Stream Editor
: **sed** is another line-oriented editor. However, it differs from **ed** in that it works non-interactively: you write a program in the **sed** language, then filter one or more files through the program. **sed** is very useful if you wish to process large amounts of text quickly and in an automated manner, but is of limited appear to most beginners.

The C Language
: This is a primer on the C language, for persons who have never programmed before. C is the native language of COHERENT and UNIX. If you are interested in learning something about C, you should look at this tutorial.

Introduction to the awk Language
: **awk** is a general-purpose for processing text. With **awk**, you can process both ordinary text and tables; thus, you can quickly implement simple data-base programs and other useful tools. If you are at all interested in processing text or data under COHERENT, you should look at this tutorial.

Introduction to lex, the Lexicon Analyzer
: **lex** is a tool with which you can generate programs to analyze text lexically. You write a set of rules for **lex**, and it generates a program that you can compile and run. **lex** is a very useful tool for programmers who have to perform sophisticated analysis of text. For many beginners, however, it is of limited appeal.

Introduction to yacc
: **yacc** is another tool that with which you can generate programs. It is often used with **lex** to build sophisticated tools, such as compilers. If you are interested in building such tools, or if the problem of parsing text interests you, you will find this tutorial to be helpful.

bc Desk Calculator Language
: **bc** is a language in which you can write programs to perform calculations. The numbers you calculate can be of infinite size and precision (or as infinite as your system's memory allows). You can perform calculations "on the fly," as with a desk calculator; or you can write programs that you store on disk and run repeatedly. If number-crunching interests you, you should find this tutorial helpful.

Introduction to the m4 Macro Processor
: **m4** is a macro processor. A *macro* is a sign or symbol whose interpretation can be deferred until needed. For example, a writer may embed a macro in a letter in place of the name of the person to whom the letter is addressed, and replace the macro with the name only when the letter is printed. This permits her to use the same letter over and over again, with the computer replacing the macro with a different name. Most COHERENT tools can process macros to a limited extent. **m4**, however, will teach you about macro processing, and let you write your own macro-process programs.

The make Programming Discipline
: **make** is a utility that builds things out of other things. This tool is absolutely essential to anyone who writes programs or generates reports.

nroff, The Text-Formatting Language
: **nroff** is a text-formatting language. With it (and its related program **troff**), you can format letters, documents, or even entire books. For example, this manual was formatted with COHERENT **troff**.

UUCP, Remote Communications Utility
: One of the most attractive features of COHERENT or UNIX is its ability to communicate with other computers without needing the assistance of a human operator. UUCP is a set of programs with which you can exchange electronic mail and files with other COHERENT and UNIX systems. With UUCP (and some help from other computer systems) you can even tie into the Internet, and participate in the global computer network that is being expanded every day. This tutorial will introduce you to UUCP, and help you get UUCP up and running on your system.

The Lexicon

The bulk of this manual consists of the Lexicon. This is a set of more than 1,000 articles, arranged in alphabetical order. Each article discusses one aspect of the COHERENT system: a command; a library function; a system file; or a general discussion of a technical topic, such as how to hook up a terminal to your system.

At first glance, the Lexicon looks like a rag-bag of material that is in no particular order. However, this is not true: it actually has a carefully designed internal structure. Once you learn this structure, you can use it both to help you look up a specific item of information quickly, or take a guided tour through some aspect of the COHERENT system that you otherwise might never have explored.

Internally, the Lexicon has a *tree structure*. Just as the roots and branches of a tree grow from a central trunk, dividing and subdividing as they progress, so too the articles of the Lexicon grow from one central article, and divide into ever more detailed discussions as they go along.

The central article is the one called **COHERENT**. It gives an overview of the COHERENT system — its design philosophy, how it relates to other operating systems, and its internal structure. This article introduces (or "branches into") the following three "overview" articles:

Running COHERENT
: This article introduces other articles that describe things you can do with COHERENT: the commands and tools that an ordinary user would use in the course of her daily work.

Programming COHERENT
: This articles introduces other articles that describe how to write programs for COHERENT. These articles describe, among other things, the COHERENT C compiler, and the the libraries and header files included with COHERENT.

Administering COHERENT
: This article introduces articles of interest to perons who administer a COHERENT system. These articles cover such topics as how to set up mail and UUCP on your system; the "magic" files that COHERENT uses to manage itself; and the COHERENT kernel and its device drivers.

For example, consider that you are a programmer who wants to learn if COHERENT has a library function that compares two strings. You would turn first to the article called **COHERENT**, which would point you to the overview article called **Programming COHERENT**. Looking in there, you see a reference to an article on **libraries**. This, in turn, points to the article on **libc**, which is the standard C library. Turning to this article, you find a section on string functions, which has brief summaries of the functions **memcmp()**, **strcmp()**, and **strncmp()**. Each of these, in turn, is described in detail in its own Lexicon entry, which By reading each entry, you can quickly find which function suits your purposes.

As you can see, the overview article briefly summarized the articles that are available on a given topic. If you want details, you can turn to the articles themselves — which you can find easily because all of the articles printed in alphabetical order.

Another approach is to look directly for an article on the subject that interests you. For example, suppose you wanted to learn about COHERENT's mail system. You could open the Lexicon and look for an article called **mail**; and just as you supposed, there it is.

If you're looking for a discussion of a specialized topic that does not have its own article in the Lexicon, look in the Index, which is at the back of the manual. Often, you will find an entry that points to the information you want.

Finally, many users just like to open the Lexicon and leaf through it at random. Often, they discover nooks and crannies within the COHERENT system that they never would have encountered otherwise.

Where To Go From Here

The next step is to install COHERENT if you have not yet done so. Fill out the user registration card at the back of this manual and mail it to Mark Williams Company, so you will become eligible for technical support. If you are new to UNIX or COHERENT, turn to the tutorial *Using the COHERENT System*; otherwise, you may wish to study a specialized tutorial or begin to explore the Lexicon.

We hope that you enjoy using your COHERENT system!

Using the COHERENT System

This tutorial introduces the COHERENT system. It introduces such basic concepts as *command* and *file system*, and walks you through simple exercises to help you gain some familiarity with the dimensions of COHERENT. If you are new to COHERENT, you should read through this tutorial first. Not every section in here will be immediately useful to every user; for example, a beginner will probably not need to study the section on system administration, at least at first. But sooner or later, you will need to work with all of the material in this tutorial.

If you are unfamiliar with what an *operating system* is, or if you are unsure how COHERENT differs from other operating systems (such as MS-DOS), turn to the Lexicon article for COHERENT. There, you will find a brief description of what an operating system is and what makes COHERENT special.

Before you can begin to use this tutorial, you must install COHERENT on your computer. If you have not yet done so, turn to the Release Notes that came with this manual and follow the directions in them.

How Do I Begin?

For everyone, there's that first time. You have installed COHERENT on your computer, you've checked the file system, mounted all of your file systems, and have gone into multi-user mode. Now you are sitting in front of your computer and all you see on your screen is the enigmatic phrase:

```
Coherent 386 login:
```

"What," you ask yourself, "do I do now?" Well, the rest of this section will tell you how to get started with COHERENT.

Logging in

To begin, you must *log in*. Unlike MS-DOS, COHERENT is a multi-user system: many people can use the same computer. They can access it either via terminals that you plug into your computer's serial ports, or via modem. Each user owns his personal set of files, his special way of setting up his environment, his own mailbox, and other things which are special to him alone. Because many people can use COHERENT, before you begin to work with COHERENT you must tell it who you are. This process of identifying yourself to COHERENT is called *logging in*. That mysterious prompt

```
Coherent 386 login:
```

is COHERENT's way of asking you who you are.

To log in, type your personal login identifier. You set this identifier when you installed COHERENT onto your computer. Most people set their login identifier to their initials or their first names, usually all in lower-case letters. Once you type your login identifier, press the `(¢)` key (sometimes labelled **<Enter>** or **<Return>**). If you did not set up a login for yourself during installation, log in as the superuser **root** and add one for yourself. For information on how to log in as the superuser, see below. For information on how to add a new user, see the section on **Adding a New User**, below, or see the Lexicon article for the command **newusr**.

While you were installing COHERENT on your system, you were given the option of setting a password for your login identifier. This is done to stop other users from logging in as you — and to keep "crackers" from dialing into your system and vandalizing it. If you did set a password, after you enter your login identifier COHERENT prompts you for it with the following prompt:

```
Password:
```

Type your password. Note that COHERENT does *not* display the password on the screen as you type it; this is to prevent bystanders from seeing your password over your shoulder as you type it. After you type your password, again type `(¢)`.

If you entered your login identifier and passwords correctly, COHERENT will display the command prompt:

```
$
```

This is COHERENT's way of saying, "Give me a command, I'm ready to go!" If you made a mistake while logging in, either with your login identifier or your password, COHERENT will reply,

```
Login incorrect:  try again.
```

and again display its login prompt:

```
Coherent 386 login:
```

Try again, until you do manage to log in. If you have received the '$' prompt, congratulations! COHERENT is now ready to work with you.

Special Terminal Keys

The next sections will introduce you to some elementary COHERENT commands. Before we continue, however, you must first become familiar with a few special keys on your computer's keyboard, and with the special meanings they have to the COHERENT system.

One special key on the keyboard will be used frequently in your work: the (¢) key. As noted above, this key is sometimes labelled **<Enter>**.

You must conclude every command you type into COHERENT by pressing (¢). This tells COHERENT that you have finished typing, and that you now want it to execute your command. COHERENT will not execute your command until you press this key.

Another special key is the **control** key. This key is usually labelled **Ctrl** or **cntl** or **cont**. Most terminals place it to the left of the keyboard. This key is used to send certain special characters.

The **ctrl** key is like another kind of shift key: to use it, hold it down while you press another key. For example, to send the computer a **<ctrl-D>** character, hold down the **ctrl** key, strike the **D** key, then release both keys.

Because control characters have no corresponding printable characters, in this tutorial they will be represented in the form:

```
<ctrl-D>
```

for the character **ctrl-D**.

While you are typing information into the COHERENT system, you can correct what you type before COHERENT processes it. Two keys will help you do this. The first is the **<kill>** character, which erases the line entirely and allows you to begin again. This is usually **<ctrl-U>**.

The other key is the **<erase>** character, normally **<ctrl-H>** or the **<backspace>** key. This moves the cursor one character to the left, to erase the most recently typed character.

One more special key is the **<interrupt>** key. This key aborts a command before it normally finishes. By default, **<ctrl-C>** is the abort key on your keyboard.

Try Some COHERENT Commands

Now that you've logged in to your COHERENT system, try a few simple COHERENT commands to get a feel for COHERENT. Type the following examples just as they are shown, and observe what COHERENT does in response to each. Be sure to press (¢) to end each line.

The first example uses the command **cat**, to let you type a small chunk of text and save it in a file.

```
cat >file01
This is a sample COHERENT file.
<ctrl-D>
```

Remember, don't type **<ctrl-D>** literally — rather, hold down the **ctrl** key and press 'D' at the same time.

In the above script, the characters **cat** tell COHERENT to invoke its concatenation program. The characters **>file01** tells COHERENT to write what you type into a file that you name **file01**. The line

```
This is a sample COHERENT file.
```

is the text that COHERENT writes into **file01**. Finally, **<ctrl-D>** signals COHERENT that you have finished typing.

Now type:

```
cat file01
```

This command again invokes the concatenation program **cat**, but this time tell it to print on your screen the contents of **file01**, which you just created. In reply to your command, COHERENT should print on your screen:

```
This is a sample COHERENT file.
```

which is the text you typed in the previous exercise.

Finally, type the command:

```
lc
```

This command lists all of the files that you have in the current directory. In reply to your command, COHERENT should print on your screen:

```
Files:
    file01
```

which is the file you just created. (You may see other files as well.)

Congratulations! You have just made COHERENT work for you.

To review: The first command, **cat**, created a file and filled it with some text. The second **cat** command copied the file onto your terminal's screen. Finally, the command **lc** printed the name of each of your files. The following sections of this tutorial describe each of these commands in more depth. Each command also has its own entry in the Lexicon, which appears in the second half of this manual; look there for a full description of each command, what it does, and how you can use it.

Giving Commands to COHERENT

Once you have logged into COHERENT, all of its resources are yours to command. COHERENT's *commands* give you control over these resources.

Every COHERENT command has the same structure: the *command name*, which tells COHERENT the command you want it to execute; and the *arguments*, which detail what you want the command to do, how you want it to do it, and to what you want it done.

Some commands consist only of the command name, and do not take arguments. For example, the command

```
lc
```

which was introduced in the previous section, has **lc** as the first part and prints the names of all files in the current directory, in columns. If you have no files, **lc** prints nothing.

The second part of the command consists of the *arguments* given to the command. (These are also known by the term *parameters*.) Arguments are separated from each other by spaces or tab characters.

The arguments of the command are further divided into *options* and *names*. *Names* usually name files; *options* modify the action of the command. An option is usually prefixed by a hyphen '-'.

An example of a *name* argument is shown in this example of a **cat** command:

```
cat file01
```

This command types the contents of **file01** on your terminal. The name argument is **file01**.

For an example of options, consider the command **ls**. **ls** lists your file names one name per line. Thus, typing

```
ls
```

produces a list of the form:

```
file01
```

However, **ls** can tell you more about a file than just its name. To see additional information about each file, type:

```
ls -l
```

The '-l' option to **ls** prints a "long" output, of the following form:

```
-rw-r--r-- 1 you    17 Sat Aug 15 17:20 file01
```

This listing shows the size of the file, the date it was created or last modified, and its degree of protection. The letters to the left of the listing give the permissions for the file; these describe who is allowed to do what to the file. These are described in detail in the Lexicon articles for the commands **ls** and **chmod**. The other entries on that line respectively name the owner of the file (in this case, *you*); the size of the file, in bytes; the date and time the file was last modified; and, finally, the file's name.

As an example of combining an option parameter with a name parameter, consider the command:

```
ls -l file01
```

This invokes the command **ls**, tells it to print a long listing, and tells it to list only the file **file01**.

As you will see in the following sections, almost all COHERENT commands have this syntax.

help, man, apropos: Help with Commands

The COHERENT system has three commands that give information about other commands: **help**, which prints a brief summary of how to use a command; **man**, which prints the full Lexicon entry for that command on your screen; and **apropos**, which shows all commands (all Lexicon entries, really) which relate to a given subject.

To find out about the **help** command, type

```
help
```

by itself, or type:

```
help help
```

The latter command tells **help** to print the help entry for the **help** command itself.

To get information on the **lc** command, type:

```
help lc
```

You will see something very like the following:

```
lc -- List directory's contents in columnar format

lc [ -1abcdfp ] [ directory ...]

Options:
     -1        List files one per line instead of in columns
     -a        List all files in directory (including '.' and '..')
     -b        List block-special files only
     -c        List character-special files only
     -d        List directories only
     -f        List regular files only
     -p        List pipe files only

Options can be combined.  If no directory is specified, the current
directory is used.
```

To obtain detailed information on a command, use the **man** command. (**man** is short for "manual".) As noted above, the **man** prints on your screen a duplicate of that command's entry in the Lexicon. To learn more about the **help** command, type:

```
man help
```

If your screen fills with information, **man** will wait for you to press the spacebar to continue. This is to prevent you from missing information should it scroll too fast.

Finally, the command **apropos** print information about all Lexicon articles that are *a propos* a given topic. For example, if you want to know what Lexicon articles are *a propos* the subject of printers, type the command:

```
apropos printer
```

COHERENT replies by printing something like the following:

```
chreq   Change priority, lifetime, or printer for a job
epson   Prepare files for Epson printer
hp      Prepare files for Hewlett-Packard LaserJet printer
hpd     Spooler daemon for laser printer
lp      Spool a job for printing
lpd     Spooler daemon for line printer
lpioctl.h      Definitions for line-printer I/O control
lpr     Spool a job for printing on a dot-matrix printer
lpshut  Turn off the printer daemon despooler
lpskip  Abort/restart current job on line printer
lpstat  Give status of printer or job
printer How to attach and run a printer
prps    Prepare files for PostScript-compatible printer
route   Show or reset a user's default printer
```

Read the summary descriptions of each Lexicon article to see which ones look promising; then either look them up in this manual, or use the **man** command to display them on your screen.

Our survey of elementary commands will conclude by describing two important tasks: how to reboot the computer, and how to log out.

Shutting Down COHERENT and Rebooting

Under many operating systems, such as MS-DOS, rebooting is as simple as pressing a couple of keys or cycling power on the computer. The COHERENT system, however, is a multi-user, multi-tasking operating system that is more sophisticated than MS-DOS or similar operating systems. COHERENT maintains an elaborate system of internal buffers that are designed to reduce the frequency with which a program has to read data from, or write data to, the hard disk. If you were just to turn the computer off and turn it on again, all of the data in those buffers would be lost. At the very least, each user would lose whatever data he was working with at the time; at worst, the COHERENT file system could be damaged and files lost.

For this reason, it is extremely important that you shut down COHERENT properly. You *must* follow these procedures if you want to shut off the computer, or if you wish to reboot MS-DOS.

To shut down COHERENT, do the following:

- When you see the COHERENT command prompt, type either **<ctrl-D>** or the command **exit**. This will log you out of your system. (Logging out is described in more detail in the following section.)
- When you see the prompt

  ```
  Coherent 386 login:
  ```

  type **root**, to log in as the superuser **root**. COHERENT will ask you for the superuser's password; type the password that you assigned to the superuser when you installed COHERENT onto your computer. The Lexicon article on **superuser** describes what the superuser is; as will later sections of this tutorial.
- Once you have logged in as the superuser, type the following command:

  ```
  /etc/shutdown halt 0
  ```

  As its name implies, this command shuts down the COHERENT system. The command will ask you if you really, truly wish to shut down COHERENT; reply 'y', for "yes".
- Now, you can turn the computer off. Or, you can type **<ctrl><alt><del>**, or press the reset button on your computer (should it have one).

  After you have rebooted your computer, just sit back and wait until you receive the **Coherent 386 login:** prompt on your screen.

  If you wish to reboot MS-DOS, watch the computer: wait until you see the computer attempting to read from the floppy-disk drive. At that moment, press the number key that corresponds to the hard-disk sector on which you stored MS-DOS, from 0 to 7. For example, if MS-DOS is kept on partition 2, then press **2** when the computer is attempting to read the floppy-disk drive. Be sure to press the number key that is on the main bank of keys, — *not* the key on the numeric keypad.

That's all there is to it. Shutting down is relatively simple and straightforward; but if you do not take the time to shut COHERENT down properly, you will find that you have destroyed some or all of your data.

By the way, the Lexicon articles on **booting** and **login** describe in detail the processes of booting and logging into your COHERENT system.

Logging Out

As noted above, *logging in* tells COHERENT who you are and that you wish to work with COHERENT for a while. When you have finished working with COHERENT, you must tell COHERENT that you are done for now. This process is called *logging out*.

There are two ways to log out. Each involves typing a special command to the COHERENT prompt. The first way is to type **<ctrl-D>** at the COHERENT prompt. The second is to type the command:

```
exit
```

Each of these commands has the same effect: the COHERENT system flushes all buffers that you "own" and prints the prompt

```
Coherent 386 login:
```

on your screen. At this point, you cannot issue any commands to COHERENT; but you (or someone else) can log into COHERENT from your terminal.

Please note that logging out is *not* the same as shutting down COHERENT. When you shut down COHERENT, you are shutting down the entire system. When you log out, however, you are simply ceasing to work with COHERENT. After you log out, however, COHERENT continues to work on its own: organizing files, exchanging information with other computers via modem, executing programs for users who have logged in via modem or other terminals, and in general making itself useful. If you shut off the computer after you log out, you will damage the file system, just the same as if you shut it off while you were logged in.

The following sections in this tutorial will go into COHERENT's commands in more detail. All, however, build on the elementary actions presented here: logging into COHERENT; issuing commands; receiving responses from COHERENT; and logging out.

Working With Files and Directories

The *file* and the *directory* are the cornerstones of the COHERENT system. Practically everything you do on the system will involve files: changing files, invoking files, transmitting or receiving files, filling files up or emptying files out. And, directories let you organize masses of files into a rational hierarchy.

This section discusses manipulating files and directories under the COHERENT system. It covers the following:

- What *file* and *directory* mean to COHERENT
- Introduces the commands for manipulating files, directories and their contents
- Discusses more advanced topics, such as creating and mounting new file systems
- Tours the COHERENT file system

This section of the tutorial covers much ground in a relatively brief space. Readers who are new to personal computers should concentrate on the earlier sub-sections, which cover elementary topics; whereas more experienced readers may wish to concentrate on the later sub-sections, which cover the more technical material.

File Names

A *file* is a mass of electronic impulses that is given a name and stored on a disk. Files are given names to make them easy for you to retrieve. COHERENT has rules about how files can be named, to ensure that each file's name is unique.

The following are examples of legal file names:

```
.profile
File01
cmd.sh
file01
test.c
```

File names are generally made up of upper-case and lower-case letters and numbers. COHERENT, unlike MS-DOS, distinguishes capital letters from lower-case letters; therefore, to COHERENT the file names **File01** and **file01** are different.

Any character can be used to name a file, including a control character. We recommend, however, that you name files using only upper- or lower-case alphabetic characters, numerals, and the punctuation marks '.' or '_'.

The file name must not be more than 14 characters long. If you specify a longer name, characters beyond the 14th will be lopped off and thrown away. For example, COHERENT regards the file names

```
this_is_very_long_file_name_1
```

and

```
this_is_very_long_file_name_2
```

as being identical.

Introduction to Directories

A *directory* is a group of files that have been given a name. Directories let you organize files systematically. This may not seem important now, but as you work with COHERENT you will find that you accumulate hundreds, or even thousands, of files; without system of directories to organize files, you would quickly lose track of what each file held, and find it nearly impossible to find any given file within your system.

Because files are stored within directories, the complete name of a file actually consists of its name plus the name of the directory in which it is stored. This lets COHERENT distinguish files that have the same name but are stored in different directories. COHERENT uses the slash character '/' to distinguish a directory name from a file name; for example, to view the contents of file **junk** in directory **text_files**, you would use the command:

```
cat text_files/junk
```

This system of naming will be described in full in the next sub-section; for the moment, just bear in mind that for COHERENT to find a file, you must tell COHERENT not only the name of the file, but the name of the directory in which it is kept.

When you work with COHERENT, you are always "in" a directory. The directory you happen to be "in" at any given moment is called the *current directory*. The current directory is the one whose files you are working with at this moment. When you type the name of a file and do not mention what directory it is stored in, COHERENT assumes that the file is kept in the current directory. COHERENT includes commands that let you shift from one directory to another.

When you log into COHERENT, COHERENT places you "in" a directory that you "own". This directory is called your *home directory*. You control all of the files in your home directory; it is your "base of operations" for working within COHERENT.

Path Names

As you may have deduced by now, a directory can contain both files and other directories. The directories within a directory may themselves contain both files and directories; which then may contain other files and directories; and so on.

This design of directories branching into other directories, which in turn branch into still other directories, is called *tree structured*. As the tree-metaphor implies, the COHERENT system of directories has a *root directory,* that is, a directory that is not contained in any other directory but from which all other directories descend, directly or indirectly. The name of the root directory is simply:

```
/
```

One subdirectory of the root directory is called **usr**. This subdirectory contains the home directories of all users. Other common paths for home directories are **/u** and **/usr/acct**. To list the names of all user directories, type the command:

```
lc /usr
```

If your login name is **henry**, then the command

```
lc /usr/henry
```

lists the names of the files in your home directory. Please note that in the argument **/usr/henry**, the first slash names the root directory; all subsequent slashes serve simply to separate one directory name from the next.

The name **/usr/henry** is called a *path name*. The term "path name" means the full name of a given file or directory — including all the directories that lead from the root directory to it.

Path names may be full or partial. All full path names begin with **/** for root, and continue with further subdirectory names. Path names that do not begin with a slash are partial; COHERENT automatically prefixes them with the path name of the current directory to make them complete before it uses them.

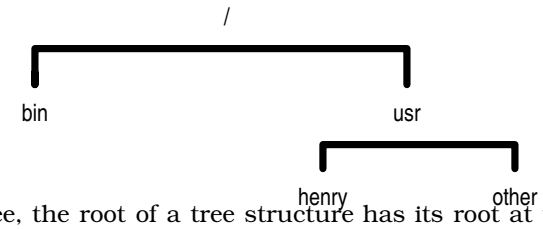
The elements of path names are separated by slashes, so if there were a file in **newdirectory** named **newfile**, you would refer to it as

```
newdirectory/newfile
```

The absence of a beginning slash indicates that the path name begins in the current directory. Thus, if your home directory name is **henry**, then another way to name the path to **newfile** is to type:

```
/usr/henry/newdirectory/newfile
```

The following diagram gives a rough description of the structure of the COHERENT file system:



Please note that unlike a real tree, the root of a tree structure has its root at the top rather than at the bottom. Here, the root directory '/' is at the top of the structure. It contains the directories **bin** and **usr** (among many others). Directory **usr** contains directories **henry** and **other** (again, among many others. These directories can contain many other directories and subdirectories.

In summary, a path name lists all the subdirectories leading from the root directory to the file in question. In the above example, **newfile** is a file in subdirectory **newdirectory**, which in turn is a file in the home directory **henry**, which is further a file in the directory **usr**. The directory **usr** is a file in the master or **root** directory for the system.

You don't need to specify all of this, fortunately, whenever you want to specify a file in a subdirectory. COHERENT assumes that partially specified path names are within the current directory. Therefore, you can specify a subdirectory by specifying the name of the directory first, followed by the rest of the path name.

COHERENT also allows two special abbreviations for directories. The abbreviation '..' always represents the current directory's *parent* directory. In the case of the directory **/usr/henry**, directory **usr** is the parent of directory **henry**. In other words, '..' stands for the directory in which the current directory resides. Every directory in the system except the root directory has a parent. For the root directory, '..' refers to itself.

Another directory abbreviation is '.', which means the *current* directory.

The following sub-sections describe the commands that COHERENT includes for manipulating files and directories. As you work with COHERENT, you will use these commands continually, so it would be worth your while to spend a little time learning them.

ls, lc: Listing Your Directory

This sub-section introduces two of the more commonly used commands: **ls** and **lc**. Both **ls** and **lc** list the files in a directory.

To see how these commands work, presume that your directory has the files created in previous sections and that you did not remove directory **newdirectory**. To list the files in your directory, simply use the command with no parameters:

```
ls
```

This produces a list of files, such as:

```
another
backup
doc1
doc2
file01
file02
newdirectory
stuff
```

The command **lc** also lists file names, but it prints the files and directories separately, in columns across the screen. For example, typing

```
lc
```

gives something of the form:

```
Directories:
   backup newdirectory
Files:
   another doc1 doc2 file01 file02
   stuff
```

If you want to list files in a directory other than your own, name that directory as an argument to the command. For example, **/bin** is a directory in the COHERENT system that contains commands. Type

```
lc /bin
```

and **lc** will print the contents of **/bin**.

Both **ls** and **lc** can take options. An option is indicated by a hyphen '-'. The option must appear before any other argument. For example, to list only the files in the directory for user **carol**, leaving out any directories, use the **f** option with **lc**:

```
lc -f /usr/carol
```

Or, if you type the command

```
lc -f
```

the COHERENT system prints all of the files in the current directory. The following gives the commonly used options to the command **lc**:

-d List directories only, omitting files
-f List files only, omitting directories
-1 List files in single-column format

ls produces a list of file names, one per line, and optionally much more information. To produce all the information, use the **-l** option (note that this is an "el", not a numeral 1):

```
ls -l
```

The following gives a sample of the long list that this option produces. Headings have been added to show the meaning of each column:

Mode	#	*Owner*	*Size, Bytes*	*Modification Date*	*Time*	*Name*
-rw-r--r--	1	you	17	Wed Aug 19	17:51	file01
drwxrwxrwx	2	you	32	Wed Aug 19	17:53	backup
-rw-r--r--	1	you	17	Wed Aug 19	17:53	doc1

The meaning of each column will be explained later. For now, note that the last column gives the name of each file, and the fourth column from the left gives the size of each file, in bytes.

cat: Print Contents of a File

The command **cat** opens and prints the contents of a text file — that is, a file of source code, a document, or a message file. For example, to list the contents of file **file01**, type:

```
cat file01
```

This command types the file's contents on the terminal (sometimes also called the *standard output*).

Another use for **cat** — the use from which it gets its name — is to *concatenate* several files on the standard output. For example, the command

```
cat one two three
```

prints the files **one**, **two**, and **three**, one after the other, on your screen.

You can use **cat** to concatenate several files into one file by *redirecting* the standard output into a file. The special character '>' tells COHERENT to redirect the standard output into a file. For example, the command

```
cat one two three >four
```

concatenates files **one two three** into file **four**. **four** need not exist prior to this command; if it does, its contents are replaced by the data redirected into it.

Redirection is a very useful feature of COHERENT that will be used through the rest of this tutorial. The '>' operator also gives an example of the set of operators that can be used with COHERENT commands. These operators, which increase the power of each COHERENT command, will be described in detail later in this tutorial.

more: List Files on the Screen

If the file you list with **cat** is more than 24 lines long, the beginning lines of the file scroll off the screen too quickly for you to read them. To ensure that you see all of the lines in the file, use the command **more**.

more prints a file in 24-line chunks. After it has listed a chunk of text, it pauses and waits for you to press **<space>**. If you call **more** with an option of **-s**,

```
more -s file
```

it will skip all blank lines that are in the text file.

mkdir: Create a Directory

The command **mkdir** creates a new directory. For example, to create a new directory named **newdirectory**, type the following command:

```
mkdir newdirectory
```

If you follow this command with **lc**, it lists your regular files, but it also lists **newdirectory** separately as a directory:

```
Directories:
    newdirectory
Files:
    file01      file02
```

To refer to any files in **newdirectory**, use its name in specifying the path name.

Now, create a file in the new directory:

```
cat >newdirectory/newfile
lines to be
contained in newfile
<ctrl-D>
```

This command copies lines to the file described by the partial path name **newdirectory/newfile**.

cd: Change Directory

The command **cd** changes the current working directory. For example, the command

```
cd newdirectory
```

moves you into directory **newdirectory** that you created in the previous sub-section. Now, if you type the command **lc**, to show the contents of the current directory, it will show the following:

```
Files:
    newfile
```

To return to the previous directory, use the command:

```
cd ..
```

As noted earlier, the abbreviation '..' always indicates the current directory's parent directory.

pwd: Print Working Directory

The command **pwd** prints the name of the current, or *working,* directory. For example, if your login name is **henry**, then if you type

```
pwd
```

you will see:

```
/usr/henry
```

Now, use the **cd** command to switch to directory **newdirectory**, as follows:

```
cd newdirectory
```

When you type

```
pwd
```

you will see:

```
/usr/henry/newdirectory
```

Finally, use the **cd** command to return to the previous directory, as follows:

```
cd ..
```

When you type

```
pwd
```

you now see:

```
/usr/henry
```

If you are ever unsure what directory you are in, use the **pwd** command.

mv, cp: Move and Copy Files

The command **mv** moves files. You can move a file from one name to another within the current directory (in effect rename the file), or you can move a file from one directory to another. **mv** takes two parameters: the first names the file to be moved; the second names either the new name that you are giving to the file, or the directory into which you are moving the file.

For example, to move file **file01** into directory **newdirectory**, type:

```
mv file01 newdirectory
```

To see where **file01** is now, type the following command:

```
lc newdirectory
```

The result is:

```
Files:
    newfile
```

To move **newfile** back into the current directory, use the command:

```
mv newdirectory/newfile .
```

Remember, the abbreviation '.' always stands for the current directory.

As noted above, the **mv** command can also be used to rename files within the current directory. For example, to change the name of **newfile** to **oldfile**, use the following command:

```
mv newfile oldfile
```

If the current directory already has a file named **oldfile**, it will be thrown away and replaced with the file that used to be named **newfile**.

The command **cp** copies a file. This command has two parameters: the first names the file to be copied, and the second names the file or directory into which it is to be copied. For example, to copy **oldfile** in the current directory back into **newfile**, use the following command:

```
cp oldfile newfile
```

If **newfile** already exists, it will be replaced by a copy of **oldfile**.

If you wished to copy **newfile** into directory **newdirectory**, use the command:

```
cp newfile newdirectory
```

Now, when you type the command

```
lc newdirectory
```

you will see:

```
Files:
    newfile
```

As you can see, **newfile** has been copied into **newdirectory**. If **newdirectory** had already contained a file called **newfile**, that file would have been replaced with the newer **newfile** being copied into **newdirectory**.

The following example summarizes what's been presented so far about files and directories. For purposes of the example, assume that your login name is **henry**, and that you have in your home directory files **doc1** and **doc2** that you wish to back up for safekeeping.

Before you can back up these files, you must first create them. First, use the command **cat** to create file **file01**, as follows:

```
cat >doc1
a few
lines of
text
<ctrl-D>
```

Likewise, create file **doc2**:

```
cat >doc2
second file
with some text
<ctrl-D>
```

(Don't forget that **<ctrl-D>** means to hold the control key down and simultaneously type **D**.)

The command **lc** will now show you the files and directories in your current directory:

```
Directories:
    newdirectory
Files:
    doc1    doc2    newfile    oldfile
```

The next step is to create the directory to hold the back-up copies. To help remind yourself what the directory is for, name it **backup**.

```
mkdir backup
```

Now, **lc** shows you:

```
Directories:
    backup      newdirectory
Files:
    doc1    doc2    newfile    oldfile
```

The next step is to use **cp** to copy your files into **backup**:

```
cp doc1 backup
cp doc2 backup
```

After you issue these commands, **lc** still says:

```
Directories:
    backup      newdirectory
Files:
    doc1    doc2    newfile    oldfile
```

However, if you list the contents of subdirectory **backup**

```
lc backup
```

you will see:

```
Files:
    doc1 doc2
```

The files have been successfully copied into the back-up directory.

TUTORIALS

For a full description of these commands and the options available with each, see their respective entries in the Lexicon.

rm, rmdir: Remove Files and Directories

The command **rm** removes a file. For example, if you wish to remove file **doc2** in directory **backup**, type the following command:

```
rm backup/doc2
```

After typing this command, use the command **lc** to show the contents of directory **backup**, as follows:

```
lc backup
```

You should see:

```
Files:
    doc1
```

As you can see, file **doc2** has been removed.

You can remove several files at once, simply by listing them on the **rm** command's command line. For example:

```
rm file01 file02
```

removes files **file01** and **file02**.

Note that once you remove a file with **rm**, it is gone forever. The COHERENT system does not warn you if you **rm** several files at once; it will assume that you know what you're doing and carry out your command silently. For this reason, be careful when you use the **rm** command, or you may receive a rude surprise.

You cannot use the command **rm** to remove a directory. COHERENT does this to help prevent you from wiping out an entire file system with one simple **rm** command. To remove a directory, use the command **rmdir**. For example, to remove the directory **newdirectory**, type:

```
rmdir newdirectory
```

Note that before you can delete a directory, that directory must not have any files or directories in it. If you try to remove a directory that has files or directories in it, COHERENT will print an error message on your screen and refuse to remove the directory.

For a full description of these commands and the options available with each, see their respective entries in the Lexicon.

du, df: How Much Space?

Files occupy space on your hard disk. (A corollary to Parkinson's law states that files expand to fill the disk allotted to them.) It is somewhat disconcerting to attempt to save a large file, only to find that you have run out of disk space. To help you manage your hard disk, COHERENT includes the commands **du** and **df**.

The disk-usage command **du** tells you how much disk space the files in the current directory occupy. If the directory has sub-directories, these are listed separately. **du** prints disk usage in blocks; each block is 512 bytes (half a kilobyte).

The disk-free command **df** tells you how many blocks are left free on your disk. By default it prints information only about the file system you are now in.

If you find that you are running low on disk space, you must free up some space. You can do that by removing files you no longer need; by *compressing* files that you do not use often; or by backing files up to floppy disk and then removing them. We have already described how to remove files. Look in the Lexicon entry for the command **compress** for information on how to compress and uncompress files. Following sections in this tutorial will describe how to copy files to floppy disk.

For more information on these commands, see their respective entries in the Lexicon.

ln: Link Files

COHERENT allows a file to have more than one name. When you create a file, you give it a name; COHERENT *links* the name you give the file with its internal system of managing files. (For more information on how COHERENT identifies files, see the Lexicon entry for **i-node**.) COHERENT allows you to give a file more than one name; another way of expressing this is to say that you can give a file *multiple links*.

To create a new link to an existing file, use the command **ln**. This command takes two arguments: the first names the file to which you wish to give a new link, and the second gives the name that you wish to link to that file. If the name you are linking to a file is already being used by a file, COHERENT will not let you link the file to that name.

For example to link the file **doc1** to the name **another**, use the following command:

```
ln doc1 another
```

The "new" file has the same data in it as the "old" file; in fact, the names **doc1** and **another** are synonyms for the same file.

The next point is somewhat subtle. When you use the command **rm** to remove a file, what you are actually doing is breaking the link between that file and its name. The file is not actually removed from disk until all links are broken between it and all of its names. In the above example, if you use the command

```
rm another
```

to remove the file **another**, the file **doc1** remains in existence, and the data to which the names **another** and **doc1** pointed remains on the disk. If you then use the command

```
rm doc1
```

to remove **doc1**, then you will have broken all links between that file and the COHERENT system, and COHERENT removes it from the disk.

Links are useful if you wish a file to be used in two different contexts but have the same data. For example, if you use file **doc1** in two different manuscripts, you can create links to the file in two different directories, one for each manuscript. Thus, any changes you make to the file under either its names appear automatically in both manuscripts.

As always, see the Lexicon for a full description of the **ln** command.

File Permissions

As you recall, the command **ls -l** prints a mass of information about each file. The following repeats the information that appeared when you typed **ls -l**:

Mode	#	*Owner*	*Size, Bytes*	*Modification Date*	*Time*	*Name*
-rw-r--r--	1	you	17	Wed Aug 19	17:51	file01
drwxrwxrwx	2	you	32	Wed Aug 19	17:53	backup
-rw-r--r--	1	you	17	Wed Aug 19	17:53	doc1

Column 3 names the owner; in this example, **you** represents your login name, whatever you have set it to. Column 4 gives the size of the file, in bytes. Columns 5 through 7 give the day of the week and the date on which the file was last modified. Column 8 gives the time the file was last modified or, if the file was last modified more than a year ago, the year it was last modified. Column 9 gives the name of the file.

Column 1 gives the *mode* of the file. The mode summarizes the *permissions* attached to this file.

Before going further, the concept of file permissions should be reviewed. COHERENT is a multi-user operating system, which means that more than one person can log into the system, walk through its file system, execute commands, and manipulate files. Every user has files that she "owns" — that is, that she has created and that she wishes to protect against being altered or removed by others. After all, it would be disconcerting if you were to log into your system, only to find that some of your key files had been trashed by another user, without your knowledge or permission.

The COHERENT system protects files by its system of file permissions. Permissions have two aspects: the *type* of permission, and the *scope* of permission. There are three *types* of permission:

read permission
Permission to read a file.

write permission
Permission to write into a file.

execute permission
Permission to execute a file, assuming that file contains executable code instead of text.

Likewise, there are also three types of *scope:*

user The permissions extended to the owner of the file.

group The permissions extended to the group of users to which the owner belongs. For more information on what *group* is, see the Lexicon entry for **group**.

other The permissions extended to all other users.

The *mode* column describes all permissions attached to a file. It also gives other information about a file, such as whether the file is a directory. Taking the entry for file **file01** as an example, we see:

```
1 2  3  4   # Owner            Size      Date         Time    File name
-rw-r--r--  1 you         17 Sat Aug 15 17:20  file01
```

As you can see, the mode field is divided into four subfields, in this example labelled '1' through '4'.

Subfield 1 indicates whether this file is a directory. If the file were a directory, this would contain a **d**; otherwise, it contains a hyphen.

Subfields 2 through 4 describe the type of permission extended to, respectively, the owner, the owner's group, and other users. Each subfield consists of three characters. The first character indicates whether the file is readable; if it is, then the character is an 'r'; otherwise, it's a hyphen. The second character indicates whether the file is writable; if it is, then the character is a 'w'; otherwise, it's a hyphen. The third character indicates whether the file is executable; if it is, then the character is an 'x'; otherwise, it's a hyphen.

In the above example, file **file01** has permissions:

```
-rw-r--r--
```

These grant read and write permission to its owner, read permission to the other members of the owner's group, and read permission to all other users.

The COHERENT system has a set of default permissions that it applies to every file when it's created. To change this default set of permissions, use the command **umask**. For information about this command, see its entry in the Lexicon. To change the permissions of an existing file, use the command **chmod**, as described in the following sub-section.

chmod: Change File Permissions

To change the mode of a file, use the change-mode command **chmod**. For example, to protect file **doc1** in directory **backup** from being overwritten, use the command:

```
chmod -w backup/doc1
```

where the **-w** means "remove write permission" and is followed by the file name. Henceforth, if you try to write into this file, the COHERENT system will refuse to do so and will print an error message on your screen.

To allow other users to read the backup file **doc2**, type:

```
chmod o+r backup/doc2
```

where the letter **o** signifies "other users", and the **+r** tells **chmod** to grant read permission.

To see the new set of permissions, type the command:

```
ls -l backup
```

As you can see, the mode string has changed from what it was above.

Directory access permissions are similar to file access permissions in that they can easily be changed via command **chmod**. However, the permission bits have different meanings for directories. Permitting reads on a directory allows the user to see the contents of the directory via commands such as **lc** or **ls**; permitting execution on a directory allows access to the files in the directory; and permitting writes on a directory allows the user to create or delete files in the directory, regardless of the permissions on the actual file. The latter causes the most difficulty for new users since they mistakenly associate file deletion permissions with the actual file rather than with the directory that contains the file.

Creating and Mounting a File System

Earlier, we described how the COHERENT system consists of a tree of directories; and how that tree branches from the root directory '/'. This is a useful description, and true as far as it goes; but the full situation is a little more complex.

The tree of COHERENT directories in fact consists of any number of *file systems,* each of which exists on its own physical device. A *physical device* may be a partition on your hard disk, a floppy disk, or even a chunk of RAM.

The COHERENT system contains a suite of commands that let you create a new file system on a physical device, and graft (or *mount*) that new file system onto the COHERENT directory tree. The following few sub-sections will walk you through the steps of creating a new file system on a floppy disk and mounting it onto your existing COHERENT directory tree. These descriptions may be a bit too advanced for beginners; but most users will find them to be interesting and helpful.

fdformat: Format a Floppy Disk

The first step in creating our new file system is to format a floppy disk. The command **fdformat** formats a diskette. When a diskette is formatted, COHERENT writes information on each track that makes it possible for the floppy disk to hold a file system.

fdformat uses the following syntax:

```
/etc/fdformat device
```

where *device* is the name of the device to be formatted. To format a high-density, 5.25-inch diskette, use the command:

```
/etc/fdformat /dev/rfha0
```

To format a high-density, 3.5-inch diskette, type:

```
/etc/fdformat /dev/rfva0
```

To format a low-density, 5.25-inch diskette, type:

```
/etc/fdformat /dev/rf9a0
```

For this example, we'll assume that you have a high-density, 5.25-inch floppy disk. Insert into drive 0 (that is, drive A) of your computer, and type the command:

```
/etc/fdformat -v /dev/rfha0
```

The **-v** option to **fdformat** tells it to verify that the disk is sound. This option means that the command will take longer to execute, but in the long run it's worth it as it will ensure that you do not waste time to trying to copy data onto a flawed disk. For details on the command **fdformat**, see its entry in the Lexicon.

When this command has finished executing, leave the floppy disk in drive 0.

mkfs: Create a File System

The command **mkfs** creates a file system on a physical device. This command has the following syntax:

```
/etc/mkfs special proto
```

special names the physical device on which the file system is to be built. *proto* is either a number or a file name. If it is a number, **mkfs** builds a file system of that size in blocks.

For our example, type the command:

```
/etc/mkfs /dev/fha0 2400
```

This command writes a file system onto device **/dev/fha0**, which in this case represents the floppy disk in drive 0 that we just formatted. The number 2400 represents the number of blocks that fits onto such a disk. Please note that the above example is for a 5.25-inch, high-density floppy disk. For directions on how to create a file system on a floppy disk of different size or density, see the Lexicon articles for **floppy disks** or **mkfs**.

If *proto* is not a number, **mkfs** assumes that it is a prototype file. The command **badscan** scans a physical device for bad blocks and writes such a prototype file for you. Prototype files are beyond the scope of this example; but for information on them see the Lexicon entry for **badscan** or the Lexicon entry for **floppy disks**. The latter article summarizes all the ways in which floppy disks are used by the COHERENT system.

mount: Mount a File System

Now that you have formatted your floppy disk and built a file system on it, you can *mount* the newly created file system. *Mounting* grafts this device's file system onto the COHERENT system's directory tree. Thereafter, you can write files onto that device, read them, remove them, or do anything else that you wish with that device and its contents.

mount has the following syntax:

```
/etc/mount device directory
```

device names the physical device whose file system is to be mounted. *directory* names the *base directory* for that file system. The base directory is the directory by which the file system is accessed. For example, directory **/usr** is the base directory for the file system that holds all users' home directories. We'll describe base directories a little further in a few paragraphs.

For purposes of our example, type the following command:

```
/etc/mount /dev/fha0 /f0
```

This mounts the file system on the disk in drive 0 onto base directory **/f0**.

The base directory by convention is a directory in the root directory '/'. You do not have to do this, however. For example, if your user name was **henry** and you wished to mount the file system on the floppy disk in your home directory, you could type:

```
/etc/mount /dev/fha0 /usr/henry/backup
```

This will mount the file system on the floppy disk onto directory **/etc/henry** and name its base directory as **backup**. Note that if directory **backup** already existed in directory **/usr/henry**, its contents will be inaccessible until you unmount the file system on the floppy disk. Unmounting is discussed in the following sub-section.

For more information on mounting a file system, see the Lexicon article **mount**.

Using a Newly Mounted File System

Now that you have created and mounted a file system, you can use it like any other directory. To see how this works, type the following command:

```
cat >/f0/testfile
Here's some text we're writing onto the
newly mounted file system on a floppy disk.
<ctrl-D>
```

Here you can use the **cat** command to write some text into file **testfile**, which lives on the floppy disk you just mounted. To see that this text has been written there, type:

```
cat /f0/textfile
```

You should see the floppy-disk drive whirl briefly, and the following appear on your screen:

```
Here's some text we're writing onto the
newly mounted file system on a floppy disk.
```

You can now use this file system like any other, even though it lives on a floppy disk rather than your hard disk. As you can see, this is an easy way to extend the size of your COHERENT system's file system.

umount: Unmount a File System

Finally, when you have finished working with a file system, you must use the command **umount** to un-mount it. This command prunes the file system on a given physical device from the COHERENT system's directory tree. You will use this command frequently as you use floppy disks.

umount takes one argument: the name of the physical device being unmounted. In our example, the command

```
/etc/umount /dev/fha0
```

unmounts the file system on the high-density, 5.25-inch floppy disk insert into drive 0 (that is, drive A) on your computer.

Under unsophisticated operating systems like MS-DOS, you can insert or remove floppy disks without giving the matter a second though. The COHERENT system, however, uses a complex set of buffers to speed the reading and writing of information to the floppy disk; for this reason, if you simply yank a floppy disk out of its drive, all of the information in the COHERENT system's buffers will be lost. Worse, if you yank out a floppy disk and insert a COHERENT-formatted floppy disk, the COHERENT system will write the data in its buffers onto that new floppy disk — and probably destroy its file system in the process. Unmounting a file system tells the COHERENT system to flush all information in its buffers and write it onto the disk.

To emphasize this point, please read the following carefully:

If you mount a floppy disk, you must use the **umount** *command to unmount it before you remove the disk from its drive. If you do not, data will be destroyed.*

This concludes the discussion of how to mount create a file system, mount it, and use it. See the Lexicon article **floppy disks** for further information on how to do this task.

The following two sub-sections discuss how to check a file system, to ensure its integrity.

fsck: Check a File System

The command **fsck** checks a file system, to ensure its integrity. For example:

```
fsck /dev/root
```

where **/dev/root** is a disk device, checks the file system located on device **/dev/root**.

If possible, you should **umount** the file system before you check it. You cannot **umount** the root file system. If you can't unmount it, be sure that no other users are on the system (i.e., that you are in single-user mode), then reboot the system immediately *without* performing a **sync**. If other users are creating or expanding files while the file systems are being checked, **fsck** will report false errors.

If **fsck** finds any discrepancies, it writes appropriate messages onto the console (that is, the screen directly plugged into your computer). An absence of messages indicates that there are no problems with the file system. The appendix to this manual gives all of **fsck**'s error messages, and suggests how you should respond to each.

COHERENT's boot routines run **fsck** automatically, and will rerun it if necessary to fix problems with the file system. For more information on **fsck**, see its entry in the Lexicon.

Devices, Files, and Drivers

The next few sub-sections introduce the topic of special files and devices. You brushed this topic in the earlier section that described how to format and mount a file system on a floppy disk; the following few sections go into it more systematically. Beginners will probably find that much of this sub-section is mystifying, but experienced users and ambitious beginners probably will find much of value here.

To begin, the COHERENT system is designed to provide device-independent I/O. Devices and files are handled in a consistent way. Each I/O device is represented as a *special file* in directory **/dev**. For example, if your system has a line printer device named **lp**, you can list a file, named **prog** for example, on the printer by saying:

```
cat prog >/dev/lp
```

Another example is to copy the file **prog** with the **cp** command to your terminal:

```
cp prog /dev/tty
```

There are two types of special files represented in **/dev** and when you list **/dev** with **lc** it will separate them.

The first type is a *block special* file. This type includes disks and magnetic tape. These devices are read and written in blocks of 512 bytes, and can be randomly accessed. (As a practical note, note that magnetic tape can be read in a random fashion only by positioning backwards and forwards one record at a time; disks can be read or written in a totally random fashion.)

The I/O to and from block devices is buffered to improve overall system performance. When a program writes a block of data, the data are held in a buffer to be written at a later time. If the same block is read twice in a row, the data for it is still available in memory and do not have to be fetched from the physical device.

A special program named **/etc/update** forces all buffered data to the physical device periodically by calling the command **sync** to protect against losing data in the case of an accident, such as a power failure. If you must bring the system down, you must force the latest data to be written by typing the command **sync**.

Character-Special Files

The second kind of special file is called a *character-special* file. Included in this class are devices that are not block special: terminals, printers, and so on. Disks and tapes can also be treated as character special files. For every block special file for a disk, such as

```
/dev/at0c
```

there is usually a character-special file:

```
/dev/rat0c
```

Character-special files are sometimes called *raw* files, hence the prefix **r** in **rat0c**. A raw file has no buffering or other intermediate processing performed on its information. This difference is an efficient benefit to commands such as **dump** and **fsck**, which do their own buffering.

tty Processing

One special set of devices has other processing — the **tty** or terminal files. A terminal-special file with this special processing is called a *cooked* device. The processing includes handling the **kill**, **erase**, **interrupt**, **quit**, **stop**, **start**, and **end-of-file** characters. Processing can be disabled with the command **stty** so the program deals with the raw device. However, using a raw **tty** device generally has negative effects on performance of the COHERENT system.

A Tour Through the File System

Our introduction to COHERENT's system of files and directories concludes with a tour of the COHERENT file system. Much of this material has been described earlier.

General File System Layout

The base of the file system is the **root** directory, whose name is simply:

```
/
```

Most of the files in the root are directories. To list the files in the **root** directory, type:

```
lc /
```

/bin

Most of the commonly used commands are programs contained in **/bin**, such as the command **lc** used in the above example. Foreign commands, such as MicroEMACS and **kermit**, are placed in directory **/usr/bin**.

The shell does not automatically look in **/bin** for commands, but consults the variable **PATH** to determine where commands are to be found. A typical value for **PATH** is:

```
/bin:/usr/bin:.
```

This tells the shell to look for commands in three places (in this order): **/bin**, **/usr/bin**, and finally **.**, the current directory. The shell does not consult **PATH** if the command contains one or more **/** characters, indicating a complete or partial path specification.

/dev

Devices in the COHERENT system are accessed through files in the directory **/dev**. If there is a line printer available on the system named **lp**, you can print characters from a file named **testdata** by typing the command:

```
cat testdata >/dev/lp
```

All devices on the system are represented in the **/dev** directory. Note that it is not recommended you access devices directly, but use the COHERENT system's utilities that *spool* files to them. This will prevent two users attempting to write material to a device simultaneously, and so garbling the output. For example, to access the line-printer device, use the spooler **lp**. See the Lexicon's entries on **printer** and **device drivers**.

/drv

A unique feature of the COHERENT system is the concept of loadable device drivers. This feature lets COHERENT system programmers write their own device drivers without modifying the rest of the system. Drivers can be unloaded, modified, and reloaded without halting and rebooting the system. Loadable drivers are kept in directory **/drv**. To load a driver, type:

```
/etc/drvld /drv/driver
```

where *driver* is the driver to load. See the Lexicon's entry on **drvld** for more information.

/etc

Several commands that you will use in your role as system administrator are kept in directory **/etc**. These are described in detail elsewhere in this guide. They include commands for system accounting, booting the system, mounting the system, create file systems, and control system time.

Also in **/etc** are several data files used in system administration. These include **/etc/passwd**, the file containing user names, ids, and passwords; news files; and file **/etc/ttys**, which describes the properties of each user terminal attached to the system.

/lib

The COHERENT system provides many useful functions for performing input and output (I/O) and mathematics, for use in your C programs. These and other libraries, along with the phases of the C compiler itself, are kept in directory **/lib**. This directory includes files containing standard system calls, standard I/O, and mathematical routines such as **sin**, **cos**, and **log**.

/usr

The directory **/usr** contains user directories, along with a few system directories.

/usr/adm contains additional information of interest to the system administrator.

/usr/bin contains commands that were not entirely created by Mark Williams Company.

/usr/games contains computer games. **/usr/games/lib/fortunes** holds a set of *bon mots*; the game **fortune** selects one at random and prints it on your screen. A call to this game can be placed in a user's **.profile**, so he will see a new fortune each time that he logs on. To add fortunes of your own, just edit the file **/usr/games/lib/fortunes**.

The directory **/usr/include** contains header files for C programs, such as **stdio.h**. Other header files define formats of files and other important data structures in the system.

/usr/lib contains the macro files **ms** and **man** used the **nroff** text processor; the unit conversion tables for the command **units**; and the file **/usr/lib/crontab** used to hold commands for **cron**. This directory also holds the C libraries.

/usr/man contains manual sections referenced by the commands **man** and **help** commands.

/usr/msgs stores messages displayed by the command **msgs**.

/usr/pub contains public files, such as telephone numbers and a copy of the ASCII table.

/usr/spool contains information for line-printer spooling, and mail that has not yet been delivered.

/u

In some systems, users' directories are placed on a separate device to save space. Because a separate device has a separate file system, the directory on that device is called **/u**.

Files: Conclusion

This concludes this tutorial's discussion of files and directories. The rest of this tutorial introduces COHERENT's suite of commands, and discusses topics of special interest to persons who are administering COHERENT systems.

Introduction to COHERENT Commands

This section introduces COHERENT's commands. The COHERENT system comes with more than 200 commands, which perform a variety of work, from formatting text, to editing files, to performing low-level administration of the system. The commands that manipulate files and directories were introduced in the previous section; there are, however, many other varieties of commands, many of which will be introduced here. To begin, we'll introduce the COHERENT system's master command, the *shell*.

The Shell

When you type commands into the COHERENT system, it appears that you are communicating directly with the computer. This is not exactly true, however. When you type into the COHERENT system, you are actually working with a special COHERENT program, the shell. This program reads, interprets, and executes every command that you type into the system. The shell can also interpret, expand, and otherwise flesh out what you type; this is done to help spare you unnecessary typing, and to permit you to assemble powerful commands with only a few keystrokes.

Please note, in passing, that the COHERENT system comes with two shells: the Korn shell **ksh** and the Bourne shell **sh**. These shells have somewhat different features. The descriptions in this section assume that you are using **sh**, which is COHERENT's default shell.

The shell is so powerful that mastering it is a major accomplishment; however, you can take advantage of much of what the shell offers by learning a few simple commands and procedures.

This section introduces some commands commonly used by COHERENT users. For more information on these or other commands see **help** and **man**. Also, consult the Lexicon.

Please note the following special punctuation characters:

```
* ? [ ] | ; { }
( ) $ = : ` ' " < > << >>
```

These characters have special meaning to the shell, and typing them can cause the shell to behave quite differently from what you may expect. Do not use these characters until you have read the following section, which discusses their use, or until they are presented in examples.

Redirecting Input and Output

Most COHERENT commands write their output to the *standard output* device, which is normally your terminal's screen. For example, **who** prints on your terminal the name of each user currently logged into your COHERENT system:

```
who
```

By using the special character **>**, you can redirect the output of **who** into a file. The command

```
who >whofile
```

writes this information into **whofile**. The operator **>** tells COHERENT to *redirect* the standard output. Later, you can list the information on your terminal using **cat**:

```
cat whofile
```

Once the information is in a file, you can process it in other ways. For example

```
sort whofile
```

sorts the contents of **whofile** and prints the results on your screen. In this way, you can display the users' names on your terminal in alphabetical order.

You can also redirect the *standard input* to accept input from a file rather than from your terminal. To redirect the standard input, use the special character **<** before the name of the file that you want read as the standard input. For example, the command **mail** sends electronic mail to another user; normally, it "mails" what you type on the standard input, but you can use '**<**' to tell it to mail the contents of a file instead.

```
mail fred < whofile
```

mails the contents of **whofile** to user **fred**.

Pipes

The *pipe* is an important feature of the COHERENT system. Pipes allow you to hook several programs together by redirecting the output of one into the input of the next. A pipe is represented by the character '|' in the command line.

Most COHERENT programs are written to act as *filters*. A filter is a program that reads its input one line at a time or one character at a time, performs some transformation upon what it has read, and then writes the transformed data to the standard output device. You can easily perform complex transformations on data by hooking a number of simple filters together with pipes. Consider, for example, the command:

```
who | sort
```

Here, the command **who** generates a list of persons who are logged into the system. The output of **who** is then piped to the program **sort**, which sorts the list of users into alphabetical order and prints them on the standard error device.

The power and flexibility of the COHERENT operating system owes much to the pipe.

Superuser

A special user in the COHERENT system, called the *superuser*, has privileges greater than those of other users. The superuser can read all files (except encrypted files) and execute all programs. You must be logged in as the superuser during certain phases of your work as system administrator.

There are two ways to access the COHERENT system as the superuser. The first is to login under the user name **root**. When the system prompts

```
Coherent 386 login:
```

reply:

```
root
```

This automatically makes you **superuser**. To remind you that you are superuser, the COHERENT system prompts you with **#** instead of the usual **$**.

The second way to acquire the privileges of superuser is to issue the command

```
su
```

when you are logged in as a user other than **root**. You must have privileges to access **root** to do this, and you must know the password for **root**. When you type

```
<ctrl-D>
```

in this mode, COHERENT returns you to your previous identity.

To be the superuser for only one command, use the form of the command

```
su root command
```

command is the command to be executed as superuser. For example, to edit the message of the day file **/etc/motd** if you are not the superuser, type

```
su root me /etc/motd
```

When you finish using MicroEMACS, your original user id will be unchanged.

To limit access to privileged resources, the COHERENT system requires users to enter *passwords* before being granted that privilege. Users may be required to enter passwords before logging in.

If the **root** user has a password, you will be prompted for it. If you do not enter it correctly, the system will tell you

```
Sorry
```

and not allow you to become the **superuser**.

It is normal practice to protect access to superuser status by setting the password. If you are the only user of your COHERENT system, or if you deeply trust all other users, you do not have to do so. However, because the superuser can perform any sort of mayhem on your system, it is advisable to set the password, especially if outsiders can dial into your system via modem.

vsh: The Visual Shell

Some users prefer to work visually rather than type explicit commands on a command line. For these users, COHERENT offers the command **vsh**, its *visual shell*.

vsh does *not* give you a true windowing system, like X or Microsoft Windows; nor does it use a mouse. However, **vsh** does give you a visual desktop for your files and commands. You can use the arrow keys on your terminal or console to select a command or a file; execute commands with one keystroke; program your function keys to execute commands automatically; and in general make COHERENT easier to use.

TUTORIALS

vsh is described in full in its Lexicon entry. However, to get the flavor of **vsh**, try the following exercises:

- Before you begin, make sure that COHERENT knows what kind of terminal you are working at. To check that type the command:

  ```
  echo $TERM
  ```

  If you are working at your computer's console, you should **ansipc** on your screen; whereas if you have logged in from an IBM PC plugged into your computer's serial port, you should see the response **vt100**. If you do not see the correct response, do *not* do the following exercises, because all you will on your screen is a jumble.

- If your terminal's type is set correctly, invoke the visual shell by typing **vsh** on your command line.

- In a moment, **vsh** draws its desktop on your screen. You will see five windows: Two long, narrow windows across the top of your screen; a big window on the left; and two small windows stacked on top of each other on the right. The top narrow window has a number of commands in it; the large window on the left displays all of the files in the current directory. The top file displayed in the large, left window is highlighted.

- The two stacked windows on the right of the screen give information about your system. They give, for example, the name of your system, the number of bytes in the current directory, your login identifier, whether you have mail waiting for you, and other information.

- Press the arrow key ↓. The bar of highlighting moves to the next name displayed in the large, left window. The scroll bar in the large, left window also creeps down a little. Now, press the arrow key ↑. The highlighting bar moves up again.

- Press **E** (for Exit). **vsh** opens a pop-up window and asks if you want to really exit. Press (¢) to accept the underlined option, 'y'. When you do so, **vsh** then exits and returns you to the COHERENT command line.

vsh is a powerful command that makes COHERENT much easier for the average user. See its Lexicon entry for details on its features and how to use it.

Manipulating Text Under COHERENT

The COHERENT system includes a number of commands and utilities with which you can process text. The phrase *process text* means to edit it and prepare it for printing.

MicroEMACS: Text Screen Editor

COHERENT includes a full-featured screen editor, called MicroEMACS. MicroEMACS allows you to divide the screen into sections, called *windows*, and display and edit a different file in each one. It has a full search-and-replace function, allows you to define keyboard macros, and has a large set of commands for killing and moving text.

Also, MicroEMACS has a full help function for C programming. Should you need information about any macro or library function that is included with COHERENT, all you need to do is move the text cursor over that word and press a special combination of keys; MicroEMACS will then open a window and display information about that macro or function.

For a list of the MicroEMACS commands, see the Lexicon entry for **me**, the MicroEMACS command. A following section of this manual gives a full tutorial on MicroEMACS. In the meantime, however, you can begin to use MicroEMACS by learning a half-dozen or so commands.

To invoke MicroEMACS, type the command

```
me hello.c
```

at the COHERENT prompt. This invokes MicroEMACS to edit a file called **hello.c**. Now, type the following text, as it is shown here. If you make a mistake, simply backspace over it and type it correctly; the backspace key will wrap around lines:

```
main()
{
        printf("hello, world\n");
}
```

When you have finished, *save* the file by typing **<ctrl-X><ctrl-S>** (that is, hold down the control key and type 'X', then hold down the control key and type 'S'). MicroEMACS will tell you how many lines of text it just saved. Exit from the editor by typing **<ctrl-X><ctrl-C>**.

Now, re-invoke MicroEMACS by typing

```
me hello.c
```

The text of the file you just typed is now displayed on the screen. Try changing the word **hello** to **Hello**, as follows: First, type **<ctrl-N>** That moves you to the *next* line. (The command **<ctrl-P>** would move you to the *previous* line, if there were one.) Now, type the command **<ctrl-F>**. As you can see, the cursor moved *forward* one space. Continue to type **<ctrl-F>** until the cursor is located over the letter 'h' in **hello**. If you overshoot the character, move the cursor *backwards* by typing **<ctrl-B>**.

When the cursor is correctly positioned, delete the 'h' by typing the *delete* command **<ctrl-D>**; then type a capital 'H' to take its place.

With these few commands, you can load files into memory, edit them, create new files, save them to disk, and exit. This just gives you a sample of what MicroEMACS can do, but it is enough so that you can begin to do real work.

Now, again *save* the file by typing **<ctrl-X><ctrl-S>**, and exit from MicroEMACS by typing **<ctrl-X><ctrl-C>**.

Just as a reminder, the following table gives the MicroEMACS commands presented above:

<ctrl-N>	Move cursor to the *next* line
<ctrl-P>	Move cursor to the *previous* line
<ctrl-F>	Move cursor *forward* one character
<ctrl-B>	Move cursor *backward* one character
<ctrl-D>	*Delete* a character
<ctrl-X><ctrl-S>	*Save* the edited file
<ctrl-X><ctrl-C>	Exit from MicroEMACS
<ctrl-Z>	Save a file and exit

Note that on some terminals, the arrow keys will not work. Note, too, that some remote terminals may have trouble using **<ctrl-S>**, if they use XON/XOFF to control flow. In this case, use **<ctrl-Z>** instead.

For more information, see the tutorial for MicroEMACS included with in this manual.

pr, prps, lp: Print Files

The command **lp** prints files for you, making sure that your request does not conflict with other uses of the printer. To print a file, type the command

```
lp file
```

substituting the name of the file to be printed for "file". If you don't name a file on the command line, **lp** prints what it receives from the standard input. Thus, you can use **lp** in pipes; this allows you to print immediately matter that you type on your keyboard.

lp will take your file and try to print it on any printer you have plugged into your computer's parallel port. If you do not have a printer plugged in, or if it is not turned on, **lp** will hold onto your files until the printer becomes ready; it will wait days, if necessary, until the printer becomes available.

lp is also intelligent enough to handle requests from several different users: if more than one user wants to print a file, **lp** will print them one at a time. In this way, the COHERENT system lets several users share one printer.

lp does nothing to the file other than print it. This means that it does not attach page heading to a file, nor does it break up the file into page-sized chunks. Another command, **pr**, does this for you. It paginates the standard input, giving a header with date, file name, page number, and line numbers. The paginated output appears on the standard output.

To print a paginated file on the line printer, type:

```
pr file | lp
```

Note the use of the pipe '|', which passes the output of **pr** as input to **lp**.

The command **prps** is like **pr**; however, it writes a file in the PostScript page-description language, suitable for printing on a PostScript printer. If you have PostScript on your system, you should use **prps** instead of **pr** to paginate text for printing.

COHERENT has many more features for printing files than can be covered here. For more details on **lp** and on how to print text, see the Lexicon entry for **printer**.

nroff, troff: Text Formatters

The commands **nroff** and **troff** format text for display or printing. They are, in fact, text-formatting languages: you type commands into your text file, and **nroff** or **troff** interprets the commands to format the text in the manner that you want.

nroff and **troff** differ in the style of formatting that they perform. **nroff** formats text into monospaced font, like that on an ordinary typewriter, Its output is suitable for display on the screen. **troff** formats text into proportionally spaced fonts, like those seen on this page. Its output is suitable for printing on a laser printer or other sophisticated typesetting device. The commands for **nroff** and **troff** closely resemble each other. The following descriptions will assume that you are using **nroff**, but they apply to **troff** as well.

nroff's programming language is quite complex and sophisticated. This manual includes a tutorial that introduces **nroff**'s language. You can, however, use **nroff** to perform simple formatting tasks by using the **ms** macro package. The following describes some of the more commonly used **nroff** commands.

To see how **nroff** works, type the following script:

```
cat >script.r
.ds CF "Print on Bottom of Each Page"
Here is some text.
Here is some more text.
.PP
The above command set a new paragraph.
Yet more text.
.SH
Here is a Section Heading
.PP
More text.
\fBThis is printed in bold face.\fR
This printed in Roman.
\fIThis is printed in italics or underlined.\fR
.PP
Here's some more text.
Here's yet more text.
And more text yet.
<ctrl-D>
```

Now, format and display the text with the following command:

```
nroff -ms script.r | more
```

You will see the text formatted for your screen. The string **Print on Bottom of Each Page** appears at the bottom of the display. The following describes the **nroff** commands with which this formatting was performed.

nroff's commands are introduced in either of two ways: by a period '.' in the *first* column of a line; or by a backslash '\' occurring anywhere in a line. The following reviews this script in detail.

.ds CF This defines the text to appear on the bottom of each page. If the text is more than one word long, it must be enclosed within quotation marks.

.PP Begin a new paragraph. **nroff** skips one line and indents the following line by five spaces (one-half inch).

.SH Print a section heading. **nroff** skips one line and prints in boldface the line of text that follows this command.

\fB Print the following text in **boldface**.

\fR Print the following text in Roman.

\fI Print the following text in *italics*.

With these few commands, you can perform simple formatting of your text.

To print the formatted text, use the command **lp**. For example, to print **script.r** on a line printer, use the command:

```
nroff -ms script.r | lp
```

This discussion is sufficient to get you started, but it just scratches the surface of what you can do with **nroff** and **troff**. See their respective entries in the Lexicon for details of what these commands can do. See the tutorial for **nroff** that appears later in this manual for a thorough introduction to the formatting language used by these commands.

Miscellaneous Commands

COHERENT includes numerous commands that perform miscellaneous tasks. These include some of the most useful, and entertaining, commands in the COHERENT system.

who: Who Is on the System

To find who is logged into the system, use the COHERENT command **who**. This command lists who is logged into the COHERENT system, one name per line. You will see your own user name there as well.

If you sit down at a terminal that is not in use, but at which someone has already logged in, the following command tells you who is logged in:

```
who am i
```

COHERENT replies with the name of the user logged in at that terminal.

write: Electronic Dialogue

The command **write** lets you carry on a "conversation" with another user. The conversation continues until you or the other user type **<ctrl-D>** on his terminal.

For example, user **fred** can begin a conversation with user **anne** by typing:

```
write anne
```

On **anne**'s terminal, the message

```
Message from fred...
```

will appear. To establish the other half of the communication, **anne** should then say

```
write fred
```

and a similar notification appears on **fred**'s terminal.

At this point, both users simply type lines on their terminal and **write** sends the message to the other user. To avoid typing at the same time, each user should end a "speech" by typing a line that has the single letter

```
o
```

to signify "over", or "go ahead". When the other user sends you this, you know it is your turn to "talk", and vice versa.

When your communication is finished, you should type

```
oo
<ctrl-D>
```

Here, **oo** means "over and out", and the **<ctrl-D>** terminates the **write** command. Note that **o** and **oo** are polite conventions, and are not necessary to using **write**.

mail: Send an Electronic Letter

You can send electronic mail to another user on your COHERENT system by using the command **mail**. This command works whether or not that person is logged into the system at the time you type your message. The message is stored in an electronic "mailbox", and the user will be notified that a message is waiting for him the next time he logs into your system.

Before you can use **mail** on your system, you must run the program **uuinstall**. This program will ask you some questions about how you have configured your COHERENT system, and will write files of information that **mail** and the communications protocol UUCP need to deliver your mail. For detailed directions on how to run **uuinstall**, see the section *Installing UUCP* in the UUCP tutorial that appears later in this manual.

TUTORIALS

Among other things, this program will ask you to name your "site" and your "domain". Without going into too much detail at this point, the site is *nom de plume* by which your machine is known to other COHERENT or UNIX systems. Site names generally are not computer-ese; **conan**, **terminator**, **lepanto**, **chelm**, and **smiles** are all examples of site names. If you don't intend to communicate with other systems, use your first name as the site name. The domain is the name by which a group of related machines are together known. If you and a number of other local COHERENT systems wish to be known together, you can establish a domain and register it with the network. Domain names, too, should be descriptive. If you don't intend to use a domain, set the domain name to **UUCP**.

To mail a message to user **anne**, just type:

```
mail anne
```

mail immediately prompts you for a title for your message:

```
Subject:
```

You can type the message's subject, which will be used to title the message, or you can just press (¢).

Once you have titled your message, type the body of the message. You can conclude your message in any of three ways: you can type **<ctrl-D>**, type a period '.' at the beginning of a line, or a question mark '?' at the beginning of a line. The first two methods end the message immediately; the last method, however, invokes an editor, and lets you edit the message further before sending it on to the intended recipient. Environmental variable **EDITOR**, if defined, selects the editor to be used.

For example, to send your message to user **anne**, you might do the following. First, invoke **mail**:

```
mail anne
```

Next, give your message a title:

```
Subject:  I'll be working late
```

Finally, type the body of the message:

```
I'll be working late.  I hope to get home before Catherine
and George go to bed.  Please remind Ivan and Marian to do
their homework.  Marian should remember to practice her
violin.
<ctrl-D>
```

If you wish, you can first type your message into a file and then mail it. For example:

```
cat >hb.msg
All come to the birthday party at four
next to the pump room.
<ctrl-D>
```

To mail the message to user **jill**, type:

```
mail jill <hb.msg
```

You can send a mail message to several users at one time by listing each user's name on the command line. For example, the command

```
mail jill jack ted barb < hb.party
```

mails the contents of file **hb.party** to **jill**, **jack**, **ted**, and **barb**. To illustrate the use of the mail command, send yourself a mail message. Type the following; substitute your user name for "you" in the mail command:

```
mail you
Subject:  test the COHERENT mail system
This is a note to
myself to test
mail.
.
```

If someone has sent you mail, the COHERENT system will tell you:

```
You have mail.
```

when you log in.

To receive mail, type the **mail** command with no parameters:

```
mail
```

If you have no mail, COHERENT will tell you:

```
No mail.
```

If you do have mail, the system will print each message on your terminal, along with the user name of the sender, and the date and time that the message was mailed.

After each message, the **mail** program types a question mark **?** and waits for your reply. You can type any of the following commands in reply to the prompt:

d Delete the message.

<Return>
Proceed to the next message.

s *file* Save, or copy, the message into *file*.

q Quit — exit from **mail** and return to the shell.

You will know that you are finished with all of your messages when **mail** sends you a **?** without typing anything before it.

mail can also send messages to other COHERENT or UNIX systems via the UUCP utility. See the accompanying tutorial on UUCP to see how you can set up COHERENT to do this.

msgs: Cumulative Message Board

The message of the day disappears when a new message is inserted. If a user does not log in for several days, the message of the day may no longer be there. For items that you want everyone to see, such as hours of operation or new operating procedures, you should use **msgs** instead of **motd**.

msgs helps users get all important messages, even if they don't log in every day. The system remembers which users have seen each message. After a user logs in, invoking **msgs** will show the number, date, and author of each message written since the user last logged in. Therefore it is easy for the user to stay up to date with the system-wide messages.

To add a message to the file, simply mail the message to **msgs**. To title the message, write it as the first line in the message, after the "Subject:" prompt from **mail**.

The home directory for **msgs** will grow over time, as more and more messages accumulate. Also, if a new user is enrolled on your COHERENT system, he may have to wade through several hundred messages when he first logs in. Therefore, you should purge the home directory for **msgs** every now and again; you may wish to throw away the announcements of office parties three Christmases ago, and save important information on diskette.

msgs keeps track of what messages each user has read by recording the number of the last message read in the file **$HOME/.msgsrc**. When each user logs on, his version of **.msgsrc** is inspected to determine the last message seen. If messages were added after that, **msgs** prints the ones the user wants to see, and then updates **.msgsrc**.

grep: Find Patterns in Text Files

The command **grep** lets you find lines that contain a *pattern* within one or more files. Patterns are sometimes called *regular expressions*.

To illustrate **grep**, create file **doc1** by typing:

```
cat >doc1
a few lines
of text.
<ctrl-D>
```

Then the command

```
grep text doc1
```

prints the second line of file **doc1**:

```
of text.
```

The first parameter to **grep** is the *pattern* for which you are looking; the rest of the arguments are the names of files

to be examined. **text** is the pattern and **doc1** is the file.

To find if a particular user is on the system, pipe **who** into **grep**:

```
who | grep you
```

(Substitute the user name in question for **you**.) Try it with your user name. The pattern is **you**, but no file name is specified. **grep** reads input from the standard input, which in this example is connected to the output of the **who** command.

You can specify several files to be searched; simply put the additional file names after the first:

```
grep pattern doc1 doc2
```

Or, you can search all files in the current directory for the pattern with

```
grep pattern *
```

The asterisk will be interpreted to mean all files, and grep will look for *pattern* in each.

The search pattern can be a *pattern*. Patterns are fully discussed in the tutorial for **ed**.

You can also locate lines that do *not* contain given patterns by using the **grep** option **-v**.

```
grep -v bugs prog1 prog2
```

This command finds and prints all lines in files **prog1** and **prog2** that do not contain the pattern **bugs**.

date: Print the Date

The COHERENT system keeps track of the time and date. To find the date and time, use the command:

```
date
```

COHERENT responds with the day of the week, the month day and year, and the time of day.

Internally, the COHERENT system records the date and time as the number of seconds since January 1, 1970, 00:00:00 Greenwich Mean Time (GMT). This means that files created in one time zone and referenced in another time zone will bear the correct time. The time and date printed out is converted from the internal form to the local time.

passwd: Change Your Password

You should change your password from time to time, to ensure that no unauthorized person can gain access to your files (or to the system as a whole).

It is easy to change passwords on the COHERENT system: just type the command **passwd**. **passwd** first asks you for your current password (if you have one), and then asks you to enter your new password twice. Entering the new password twice helps ensure that the system gets the password as you want it. If you do not type it the same way both times, COHERENT will say:

```
Password not changed.
```

You must then begin again with the command **passwd**.

Be sure the password is something that you can remember. It is recommended that the password be at least six characters long. Do not write it down, but memorize it. You can use a four-letter password, but if you do, you should mix upper-case and lower-case letters to make it more difficult for outsiders to guess.

stty: Change Terminal Behavior

Because a wide variety of terminals can be used with the COHERENT system, you must pass some information to the COHERENT system so it can handle your terminal correctly.

The command **stty** describes the information COHERENT currently has for you; you can then use **stty** with arguments to change how COHERENT handles your terminal.

For example, COHERENT normally echoes each character you type, as you type it. However, if your terminal also echoes what you type, you will see double characters. To prevent this, issue the command:

```
stty -echo
```

The program **login** uses this feature when you type your password, to help keep it secret from anyone who is kibbitzing at your desk.

To set the echo feature again, type:

```
stty echo
```

When you first log in, the system presumes that your terminal does not directly handle the **tab** character, so when COHERENT sends a **tab** to your terminal it simulates it with spaces. If your terminal does handle tabs, issue the command:

```
stty tabs
```

The COHERENT system will no longer substitute spaces for tabs. To go back to substitution,

```
stty -tabs
```

The **<erase>** character lets you delete the previously typed character. The **<kill>** character lets you delete the line that you have been typing but have not yet finished. By default, COHERENT sets these to, respectively, **<ctrl-H>** and **<ctrl-U>**. To change them to, respectively, **<ctrl-E>** and **<ctrl-K>**, use the **stty** command as follows:

```
stty erase ^E kill ^K
```

The carat '^' tells **stty** that you want to specify a control character.

To reset erase and kill to the default values at login, the command

```
stty ek
```

suffices. This is equivalent to:

```
stty erase ^H kill ^U
```

To see what your current terminal parameter settings are, type

```
stty
```

with no arguments.

Scheduling Commands For Regular Execution

The command **cron** is a valuable tool for using your COHERENT system. With it, you can instruct COHERENT to execute commands at various times of the day or night, even if nobody is logged into the system.

To specify a command to be executed at some later time, simply enter one line of information in a "cron" file. (Where the cron file lives will described below.) For example, assume that you want to greet user **norm**, if he is logged into the system on Monday morning. You can do this by sending him a message at 8:13 on Monday. Use MicroEMACS to add the following line to the cron file:

```
13 8 * * 1     msg norm%You are sure in early!
```

The numbers and * at the beginning specify the time:

```
13 8 * * 1
```

The **13** means "13 minutes past the hour". (**cron** numbers the minutes zero through 59.) The **8** means "8 AM". (**cron** numbers the hours of the day zero through 23, with zero indicating 12 AM.) The positions containing * normally specify the day and month. The two * characters mean "any day" and "any month". Finally, the **1** means "day 1 of the week," which is Monday. (**cron** numbers the days of the week zero through six, with zero indicating Sunday.) The breakdown of this command is shown as follows:

minute	13
hour	8
day of month	* — all days
month	* — all months
day of week	1 — Monday

Because each entry in the cron file must be on one line, the symbol **%** represents the beginning of the input string. If the information is too long for one line, type a backslash character before you press (¢) to end the line. The backslash tells **cron** to ignore the **<Return>** character.

With this information in the file, **cron** executes the command

```
msg norm
Am Monday!
```

TUTORIALS

at 8:13 every Monday morning.

cron expects time to be in the 24-hour clock, so 1 PM is represented as **13** hours. If you need to print a literal percent sign '%', precede it with a backslash:

```
\%
```

The times for **cron** commands can be even more complex than the numbers and * shown above.

You can express a range for any of the five parts of a time by separating two numbers with a hyphen. For example, to send user **marianne** a humorous message on week days, use the command:

```
59 11 * * 1-5   /usr/games/fortune | msg marianne
```

To list a choice of times, separate single numbers or ranges with commas but no spaces. To send notification about a meeting on Monday, Wednesday, and Friday at 3 PM, use:

```
0 15 * * 1,3,5  echo Meeting at 3:30 ... | mail fred anne joe
```

The time specification

```
0 15 * * 1,3,5
```

represents the time 1500 (3 PM) on every Monday, Wednesday, and Friday.

mail and **msg** are just some examples of commands that can be used with **cron**; many others can be used. For example, **cron** is commonly used to execute UUCP commands late at night, when telephone rates are low. See the Lexicon article on **cron** for more information about this command. If you wish to schedule commands to be run but not on a regular basis, use command **at**. See its Lexicon article for further details.

As was mentioned above, you must edit a cron file for the **cron** daemon to execute a command automatically. COHERENT uses a complex scheme of cron files. If the file **/usr/lib/crontab** exists, the **cron** daemon will read it — and only it. However, if **/usr/lib/crontab** does not exist, the **cron** daemon will look in directory **/usr/spool/cron/crontabs** for the set of cron files located there. Each user can have his own cron file. All commands in a user's cron file are executed with the suite of permissions granted to that user; thus, a user cannot use **cron** to delete files that do not belong to him. If a user wishes to create or update his own cron file, use the command **crontab**; see its entry in the Lexicon for details.

Please note that each flavor of cron file uses the syntax described above.

Managing Processes

A *process* is a command that is undergoing execution. Because COHERENT is a multi-tasking operating system, numerous processes can be undergoing execution at the same time. The following commands let you monitor and, within limits, affect the operation of the processes your COHERENT system is executing.

ps: List Active Processes

Each process in the system is assigned a number called the *process id*, or *PID*. Each user logged into the system has one or more processes. Except in special circumstances, the first process that he has is the shell, or command-line interpreter. The commands he types are run by the shell.

The shell normally waits for a command to terminate before it begins to process the next command. However, if you use the '&' operator, the shell creates simultaneous processes: that is, while it executes one command it will let you type another. Thus, you can execute two or more commands simultaneously.

You can examine the processes associated with your login, or all processes in the system, with the command **ps**. Type:

```
ps
```

The result will resemble:

```
TTY       PID  COMMAND
color0     36  ksh
color0   4004  mail
color0   4005  me
color0   4009  sh
color0   4010  ps
```

The first column names the terminal you are running on, in this case the virtual console **color0**. This identifier is taken from the file **/etc/ttys**, with the prefix **tty** removed from name. The **tty** identifier is also printed by the

command **who**. The second column lists the corresponding process identifier (PID). The third column names each command and gives its parameters, if any. **ksh** represents the shell process, and **ps** represents the **ps** command itself.

To see all the processes, type:

```
ps -a
```

The result will resemble:

```
TTY       PID  COMMAND
null        1  init
color3     33  ksh
color2     34  ksh
color1     35  ksh
color0     36  ksh
com3l      37  login
color3   3629  sh
color3   3630  kermit
color3   3631  kermit
color0   4004  mail
color0   4005  me
color0   4011  sh
color0   4012  ps
```

This display will, of course, differ quite a bit from system to system and from minute to minute.

For a full description of all options to **ps**, see its entry in the Lexicon.

kill: Signal Processes

Occasions will arise when the system administrator must log other users out of the system. For example, you may need to bring the system down quickly; or perhaps a user forgot to log out before leaving the terminal and did not see your broadcast message requesting that all users log out.

The command **kill**, when used by the superuser, terminates processes. To log out a user whose shell has process number 300, use the command:

```
kill -9 300
```

You must be logged in as **root** or use the command **su** to **kill** a process that belongs to another user. Each user can kill all processes that he owns, including his own shell process (which automatically logs him out).

kill has other uses as well — see the Lexicon's entry for **kill** for more information.

Programming Under COHERENT

The COHERENT system provides a number of languages in which you can write programs.

The shells included with COHERENT — **sh**, the Bourne shell, and **ksh**, the Korn shell — not only process commands, but are powerful programming languages in their own right. For details on how to program in these languages, see their respective entries in the Lexicon; and see the tutorial *Introducing sh, the Bourne Shell*, which follows in this manual.

COHERENT includes a full-featured assembler, with which you can assemble your assembly-language programs. Assembly language is sometimes required for operations that require you to work very closely with the operating system or hardware. For more information on the COHERENT assembler, see the Lexicon entry for **as**.

Most programming that cannot be executed efficiently by a shell language is done in C, the language in which the COHERENT system was written. The COHERENT system comes with a full-featured C compiler, with which you can compile the program you write in that language. If you are new to C, the tutorial *The C Language*, which follows in this manual, will introduce you to it. The following sub-sections briefly describe the tools available under COHERENT with which you can write, compile, and debug your C programs.

Basic Steps in COHERENT Programming

The steps that are necessary to generate a program are:

1. Create the program source file
2. Compile the source program, correcting any errors
3. Test and debug the program
4. Run the program

If you have compilation errors in step 2, or program errors in step 3 or 4, return to step 1.

Use your favorite editor to build and change the source program, the **cc** command to compile the source program and produce an object program, and **db** to help debug the program. Although the C compiler provides a macro facility, other languages do not. Therefore, if the source program uses macros, you can use **m4** to expand the macros.

This section covers each of these steps and provides some example programs.

Create the Program Source

The first step is to use MicroEMACS, **vi**, **ed**, or some other editor to create the program's source file. Details on the use of **ed** and MicroEMACS are covered in their respective tutorials, which follow in this manual. Each editor's commands are summarized in its Lexicon article.

For the first program, try a simple program that prints a short message on your terminal. For the sake of simplicity, we'll enter text using **cat** instead of invoking an editor. To build the program, type the following:

```
cat > small.c
main ()
{
        printf ("The COHERENT operating system\n");
}
<ctrl-D>
```

The first line invokes the concatenation program **cat** to enter the program's source code. The **<ctrl-D>** signals that you have finished entering text.

The program itself begins with the special word **main** which defines a function and must appear in every C program. The parentheses, here with nothing between them, enclose any arguments that are passed to the function. They are required even if there are no arguments. The body of the program appears between the braces **{** and **}**.

The function **printf** is part of the standard library of C programs. It prints formatted information on the terminal. In this case it will produce the string enclosed between quotation marks. The special character string

```
\n
```

means "newline". Two lines of output to the terminal can be produced by

```
"line 1\nline 2\n"
```

as an argument to **printf**. This appears in the output as:

```
line 1
line 2
```

For a fuller introduction to the C language, see the tutorial on the *The C Language*, which follows in this manual.

cc: Compile the Program

The command **cc** compiles C programs. It executes all the parts of the C compiler and the associated linker **ld**. The linker combines pieces of programs and includes necessary elements from the library, such as **printf()**. The linker is occasionally called from the command line, but only for more complex problems than you are trying here. To compile our test program, type the command

```
cc small.c
```

If the compiler detects any errors, it prints a message on the terminal, along with the line number that contains the error. You can use this line number to find the error with your editor and fix it. You can now use the program by simply typing:

```
small
```

The tutorial on *The C Language* describes **cc** in greater detail; also see its entry in the Lexicon for a full summary of its many capabilities.

m4: Macro Processing

To extend the capabilities of all languages, the COHERENT system provides a macro processor, called **m4**.

Program source for all languages consists of character strings. Macro processors perform string replacement, whereby a string in the input file may be replaced by another string. **m4** provides parameter substitution, as well as testing values of currently available strings and conditional processing. **m4** is unique in that you can rearrange large sections of the input text by using the macros. For more information on **m4**, see the tutorial *Introduction to the m4 Macro Processor*, which follows in this manual.

make: Build Larger Programs

All the examples of programs thus far have been self-contained. As programs grow larger, it is usual to divide the source program into smaller files. This simplifies editing, speeds compilation, increases modularity, and lets several different programs share common functions.

Thus, in developing the larger program, you may have several source files in your directory, possibly a header file or two, and the object files that result from compilation. From these are built the executable file that runs when you type its name.

To change or fix the program, you must edit the source programs or header files in question with **ed**, recompile the required source, and relink all the modules. But, with a change that affects several modules, it can be tricky to remember exactly which modules need recompilation, and it can be time-consuming to recompile all modules.

COHERENT provides a command called **make**, which solves this problem. **make** examines the time a file was last modified, and the time of modification of files that it depends upon, and performs the necessary compilation or other processing. (COHERENT file system directories contain the time that each file was created or modified.)

The tutorial *The make Programming Discipline*, which follows in this manual, fully introduces this powerful and useful program.

db: Debug the Program

The first and most critical step to debugging programs is to not put bugs in them! The methods of structured analysis, design, and programming, or the method of stepwise refinement can substantially reduce the number of errors in a program.

One can also place **printf** statements at strategic points throughout the program to display logic flow and key data values. These display statements should be designed so that they can be turned off for normal operation without removing them from the program.

On occasion, however, you may find that it is necessary to debug at the machine level. If you must, COHERENT's **db** will make it possible to do so.

db provides tools that make the machine program instructions visible in the most natural notation. That is, instructions are displayed in a fashion that resembles assembly language, numbers can be displayed in hexadecimal, octal, or decimal as needed, and strings of characters displayed in familiar graphic form. **db** can also patch a program to be run again, as well as to control the execution of a program with breakpoints and one step at a time.

Briefly, to use **db** on a program like our sample **small** above, use the command:

```
db small
```

Now you can inspect and display instructions and data in the system, control execution, and even change the instructions in the program if you are bold enough.

To examine a data segment location in the program, simply type the address of the location. **db** knows about symbols in the program, so if you want to examine the location corresponding to **main**, type:

```
main
```

db types out the value in hexadecimal or octal (depending upon which is appropriate for your machine).

You can expand the display command to print many locations at one time, and choose the format of printout. To print five locations interpreted as instructions, type

```
main,5?i
```

where the format character **i** follows the question mark indicating format, and 5 is the count of locations to be

printed. To exit **db**, type

```
:q
```

For a complete list of the format that **db** recognizes, and other details about **db**, see its entry in the Lexicon.

Administering the COHERENT System

The COHERENT system can be used by many people at the same time. One person must coordinate its use, like a key operator does for an office copier. This person is called the *system administrator*, and he sees to it that the COHERENT system runs smoothly every day. The administrator can also customize the COHERENT system to the needs of an individual installation.

Although you may be the only person to use your COHERENT system, many of the ideas discussed here are important for making your system work at its best. Please spend a few minutes reading this manual to familiarize yourself with the elementary concepts of COHERENT system maintenance.

Adding a New User

Each user allowed to use your COHERENT system must have a *user name* and a *user id*; the user may also have a *password*. The user name is usually the user's initials or a nickname. The *user id* is an integer number used to identify the user internally to the system. As system administrator, you will assign both of these for each user. This section tells you how.

To log in to the system, a user must have an entry in the *password* file **/etc/passwd**. The password file contains each user's name, id, and password if any. As system administrator, you will maintain this file.

Likewise, each group of users is assigned a *group name*, as well as a *group id*. Groups are not necessary to use the COHERENT system, but some installations prefer to set up groups by project or department.

It is simple to add a new user to the system. The command **newusr** takes care of all the details, and makes an entry in the password file. You must be logged in as **root**. For example, to create an entry for a user named Henry, log in as **root**, and then issue the command:

```
/etc/newusr henry "Henry Smith" /usr
```

This creates an entry in **/etc/passwd** for **henry**, creates his home directory in the **/usr** file system, creates all appropriate files for him (such as his **.profile** and his mailbox), and sets all permissions correctly.

System Security

One of the most important tasks in running your COHERENT system is maintaining its security. Basically, security means two things: keeping outsiders from logging into your system, and keeping your system's users from doing untoward things. This section describes some steps you can take to ensure that your system is secure.

Passwords

Passwords provide the first level of COHERENT system security.

For systems with passwords, each user with a password must type his password as part of the login process. If he enters the password incorrectly, he cannot log in.

Your system's administrator can assign a password when she creates a user's log-in account, as described above. If you do not assign a password, anyone will be able to log in as that user.

In any system with passwords, it is especially important to assign a password to the **root**, or *superuser*. If the superuser does not require a password, any user can log in as **root** and automatically have access to the powerful tools that control the operation of the system.

Any user with a password can restrict access to his files. Once you assign him his password, he can change it with the command **passwd**. However, because of higher privileges, **root** can always access everyone's files.

The passwords are kept in file **/etc/passwd**, with the rest of the user login information. Passwords are encrypted, so reading **/etc/passwd** will not reveal passwords.

File Protection

The second level of COHERENT system security is *file protection*. A user can set each of three categories of protection for each of his files. A standard protection, or *access permission*, is given to each file when it is created.

The three categories of permissions are for the user himself, for other users in his group, and for all other users. To see the levels of protection of your files, type the command

```
ls -l
```

For more details on the meaning of each column in this printout, see the Lexicon entry for the change-mode command **chmod**.

Encryption

The command **crypt** provides a third level of system security. It lets a user encode and decode information in a file. The superuser has access to every file in the system; so to protect sensitive information even from his prying eyes, a user can disguise it with encryption. Sensitive system information, such as passwords, are also encrypted for security purposes; and the **mail** command lets users send encrypted mail to each other. For details about encryption, see the entry on **crypt** in the Lexicon.

Dumping and Saving Files

You should regularly copy your files on floppy disk, to protect your valuable files against a catastrophic system failure. The Lexicon article **backups** describes in detail how to do this.

System Accounting

The COHERENT system provides two types of computer time *accounting* to help you track the use of the system. Three commands control the accounting and provide reports at various levels of detail.

Note that system accounting adds overhead to your system, because your system has to do more work to record everything it does, and because the accounting files can quickly grow to unmanageable sizes. System accounting is useful for COHERENT systems that are being used by multiple users who must account for (i.e., pay for) their use of the system, or in other circumstances where it is important to note each user's activity. For most systems that support a handful of users, system accounting simply isn't worth the bother.

If, however, you decide that you need system accounting, read on.

ac: Login Accounting

Whenever a user logs into the COHERENT system, it records the user's name, the terminal number, and the date and time of the login. It also records when he logs out.

You can use this information to compute the time each user, or all users, were logged into the system. The command **ac** prints the total of all login times recorded in the accounting file. An example of the result is

```
Total:   8357:00
```

You can ask for a summary of total login times for each day by typing:

```
ac -d
```

An example result would be:

```
Friday November 13:
        Total:   53:08
Saturday November 14:
        Total:   75:36
Sunday November 15:
        Total:   73:15
```

Finally, you can summarize the times for individual users with the command:

```
ac -p jack ted fred
```

This will show the total login times for these users:

```
fred            1100:42
jack            910:41
ted             641:58
Total:          2653:21
```

Also,

```
ac -pd
```

gives the time for each user, for each day that he logged in.

TUTORIALS

Login accounting is not automatically operational. The login information is collected only if the file **/usr/adm/wtmp** exists.

To start login accounting if it is not working, type the command

```
>/usr/adm/wtmp
```

while logged in as **root**. This creates the file **/usr/adm/wtmp** if it does not exist (and destroys existing information if it does) and thereby enables login accounting.

To turn off login accounting while it is running, you can type:

```
rm /usr/adm/wtmp
```

After you activate login accounting, you should purge **/usr/adm/wtmp** periodically as it grows continuously, and on an active system will eventually consume much disk space. To purge the current information but leave accounting turned on, type:

```
>/usr/adm/wtmp
```

sa: Processing Accounting

While login accounting tells you how much time a user spends logged into the system, it does not tell you the individual commands used. *Process accounting* does so. Under COHERENT, each execution of each command constitutes a separate process. (COHERENT's ability to maintain a list of processes and swap each in and out of memory until all are executed, is what gives COHERENT its multi-tasking capability.) Process accounting records system time, user time, and real time for each command executed by each user on the system. The command **sa** reports this information for you, using a format that you set.

sa has several options, to generate different reports. When used with no options, **sa** lists the number of times each call is made, the total CPU time, and the total real time used by the command, ordered by decreasing CPU time. This is a summary by command; the following gives an example:

```
             #CALL   CPU     REAL
sh           61      1       832
ld           5       1       7
ar           5       0       1
ranlib       3       0       1
p            16      0       11
dld          2       0       1
lc           19      0       1
cc           4       0       8
atrun        43      0       1
find         1       0       0
ed           1       0       2
             ...
```

This report has been truncated to save space. The listing will depend on what commands are used in your system, and the characteristics of your hardware. To summarize by user, use the **-m** option:

```
sa -m
```

The option **-l** separates CPU time expended by users from that expended by the system. This command

```
sa -l
```

produces:

```
            #CALL   USER    SYSREAL
sh          61      0       1832
ld          5       0       07
ar          5       0       01
ranlib      3       0       01
p           16      0       011
dld         2       0       01
lc          19      0       01
cc          4       0       08
atrun       43      0       01
find        1       0       00
ed          1       0       02
cat         4       0       01
rm          3       0       00
            ...
```

This report has been truncated to save space. To list the user name and the command name, use **sa** with the option **-u**. No times or counts are given. The command:

```
sa -u
```

produces output of the form:

```
tj              p
tj              lc
tj              find
tj              pr
bin             lc
tj              spin
tj              sh
bin             cc
...
```

This report has been truncated and edited to save space. In practice, it is longer. The **-u** option overrides other options.

Process accounting is on only if you turn it on. To turn on process accounting, type the command:

```
/etc/accton /usr/adm/acct
```

while logged in as **root**. The file **/usr/adm/acct** holds the raw accounting information.

To turn off process accounting, use the same command with no file name:

```
/etc/accton
```

If accounting is not on when you type this command, you will get an error message. No information is gathered when accounting is turned off.

When process accounting is in use, the file **/usr/adm/usracct** grows with each user command issued. You should regularly condense or remove the information, to keep the file from devouring all free space on your disk. To condense the information, invoke **sa** with the **-s** option. You must turn off accounting while condensing information.

The information summarized by user will appear in **/usr/adm/usracct**, and information saved by command is placed in **/usr/adm/savacct**. These summarized files are used in future requests to **sa**. After condensing, you can turn accounting back on.

Additional options give flexibility to the report. See the entry for **sa** in the Lexicon for additional details on these options.

Conclusion

The following sections of this manual give tutorials to teach you how to use many of COHERENT's tools and commands. The Lexicon contains brief synopses of all commands, library routines, system calls, and macros available under the COHERENT system. It also includes many technical references and definitions, to help you with terminology throughout this manual.

Introducing sh, the Bourne Shell

sh is the command that invokes the Bourne shell, which is the COHERENT system's default command interpreter. The Bourne shell interpets commands, and much more. It is, in effect, both a programming language and an interpreter.

At least one writer has noted that the shell is the original "fourth-generation language" — that is, a powerful programming language that is straightforward enough to be programmed by non-programmers. You will find that taking a little time to master the rudiments of the shell programming language will pay enormous benefits in making best use of your COHERENT system.

Simple Commands

The shell command language is built around simple commands. For example, the following command lists all files in the current directory:

```
lc
```

You can combine several simple commands on one line by separating them with semicolons:

```
who;du;mail
```

The shell executes the commands in sequence as if they had been typed:

```
who
du
mail
```

In both of these examples, **du** does not begin execution until **who** is finished, and **mail** does not begin until **du** is done.

Special Characters

The shell treats the following characters specially; if you want to use them without their special meaning, you must precede them with the backslash character ****, or enclose them within quotation marks:

```
*  ?  [  ]  |  ||  ;  {  }  (  )  &  &&
$  =  :  `  '  "  <  >  <<  >>
```

The function of these characters will be explained later in this section. To use one of these characters in a command, for example '?', type:

```
echo \?
```

In addition, the shell treats the following words in a special way when they appear as the first word of a command:

```
if          then        elif        else        fi

case        esac

do          done

for         in

until       while       break

test
```

Running Commands in the Background

The shell can execute commands simultaneously as well as sequentially. This means that while the shell is executing one command, it lets you type and execute another command. Under the shell, the number of commands you can execute at the same time is limited mainly by the amount of memory and disk space on your system.

If a command is followed by the special character '&', the shell begins to execute it immediately, and prompts you to enter another command. For example, if you need to **sort** a large file but want to continue with other commands while the sort is executing, you can type:

```
sort >bigfile.sorted bigfile.unsorted &
ed prog
```

This allows you to edit file **prog** while your computer quietly executes the sort in the background.

When you run a command with **&**, the shell types the *process id* of the command started in background. When the COHERENT system runs a command, it assigns that command a *process id*, which is a number that uniquely identifies that command to COHERENT. Normally, there is no need to be concerned about these numbers. However, when you run commands in the background, the shell tells you the id of the background process so you can keep track of its execution.

The command

```
ps
```

lists the processes you are currently running. If you have no background jobs, the response is:

```
TTY PID
30: 362 -sh
30: 399 ps
```

The first column shows the number that COHERENT has assigned to your terminal. This is the same terminal number printed out by **who**. The second column shows the process id; the third column shows the program or command executing. The characters **-sh** in the third column means the login shell. There are two processes because the shell is running the **ps** command as a separate process.

Once you have started a background command, **ps** shows you the process entry, which has the process id that the shell typed out for you.

If you need the results from a background job, you can wait for it to finish by issuing the command:

```
wait
```

The shell will then accept no further commands until all your background jobs are finished. If there are no background jobs, there will be no delay.

Scripts

Many of the commands that you use in COHERENT are *programs*, such as **ed**. Others, like the **man** command, are *scripts*, or files that merely contain calls to other commands. You can write scripts on your own, simply by using a text editor to type into a file the commands you wish to execute. If you frequently use a set of commands, you can save yourself from having to type them over and over by simply typing them once into a script.

For example, suppose that you wish to check periodically the amount of disk space that you have used, the amount of disk space still available, and see who is using the system. You can write a script to do all of this automatically. Create the script **good.am** by typing the following commands:

```
ed
a
du
df
who | sort
mail
.
w good.am
q
```

From now on, to execute the above-listed commands, you need only type:

```
sh good.am
```

where **sh** is a command that means: read commands from a file, in this case **good.am**. If you can issue a command from your terminal, you can also execute it from within a script.

You can make a command file directly executable by using the command **chmod**. For example, the command

```
chmod +x good.am
```

lets you execute the script **good.am** by typing

```
good.am
```

and leaving off the **sh**. Once you have done the **chmod** command, you can still issue the commands by typing:

```
sh good.am
```

as well as use **ed** or MicroEMACS to change the contents of the script.

Notice that the commands called by a script may themselves be scripts. This is illustrated by the following script, **second.sh**:

```
ed
a
good.am
lc
.
w second.sh
q
```

Thus, typing:

```
sh second.sh
```

calls the script **good.am**, and also calls the command **lc**.

.profile: Login Shell Script

When you log into the system and before you are issued your first prompt, COHERENT checks your home directory for a file named **.profile**; if it is present, the shell executes the commands it contains.

This enables you to have COHERENT execute commands as soon as you log in. Check if your installation provides one for you by doing an **lc** (be sure that your current directory is the home directory). If the file is there, print it by saying:

```
cat .profile
```

Some of the commands may be of the form:

```
PATH=':/bin:/usr/bin'
```

This sort of command will be discussed below.

Substitutions

Scripts of the form shown above are processed by the COHERENT shell without change. However, the COHERENT shell increases the power of commands by performing three kinds of substitutions within commands before it executes them.

First, it replaces special characters in commands with file names from the current or other directories. This allows you to issue a single command that processes several files.

Second, you can give a script *arguments*, much like arguments that are passed to a Pascal, Algol, or C procedure. This lets you target the action of the script to a specific file name specified when you call it.

Third, the output of one command can be "piped" into another command to serve as its input.

We will use the command **echo** to illustrate these kinds of substitution. Remember that substitutions take place for all commands in the same way that they do for **echo**.

File Name Substitution

File names are often used as command parameters. That is, you will often tell a command to do something to one or more files. By using special shell characters, you can substitute file names in commands. These special characters describe file name *patterns* for the shell to look for in the directory. When the shell finds the file names, it replaces the pattern with them.

The asterisk * matches any number of any characters in file names. Thus,

```
echo *
```

echoes all the file names in the current directory, whereas

```
echo f*
```

gives all file names that begin with the letter **f**, and

```
echo a*z
```

lists all names that begin with **a** and end with **z**.

To illustrate more clearly, create two files by typing

```
cat >zz1
<ctrl-D>
cat >zz2
<ctrl-D>
```

Then the **echo** command

```
echo zz*
```

produces the output:

```
zz1 zz2
```

Thus, by using a single *, you can substitute several file names into a command. In other words, the command

```
echo zz*
```

is equivalent to

```
echo zz1 zz2
```

If no file names fit the pattern, the special characters are not changed, but left in the command exactly as you typed them. To illustrate, type the command

```
rm zz*
echo zz*
```

The first command will remove all files whose names begin with **zz**, and is therefore equivalent to:

```
rm zz1 zz2
```

The **echo** command that follows, however, echoes

```
zz*
```

because no files begin with **zz**; they were just removed.

Enclosing command words within apostrophes prevents the shell from matching file names with the enclosed characters. In the unlikely event that you have a file whose name is

```
zz*
```

that you want to remove, use the command

```
rm 'zz*'
```

The * is enclosed within apostrophes, and therefore is not changed by the shell.

Another special character **?** match any one letter. To see how this works, create empty files **file1**, **file2**, and **file33** by typing:

```
>file1
>file2
>file33
```

The command

```
echo file?
```

replies

```
file1 file2
```

because **?** does not match **33**.

You can use brackets **[** and **]** to indicate a choice of single characters in a pattern:

```
echo file[12]
```

This command replies:

TUTORIALS

```
file1 file2
```

To match a range of characters, separate the beginning and end of the range with a hyphen. The command

```
echo [a-m]*
```

prints any file name beginning with a lower-case letter from the first half of the alphabet, and is exactly equivalent to:

```
echo [abcdefghijklm]*
```

When such patterns find several file names, they are inserted in alphabetical order.

Because the character **/** is important in path names, the shell does not match it with ***** or **?** in patterns. The slash must be matched explicitly; that is, it is matched only by a **/** itself. Therefore, to find all the files in the **/usr** directories with the name **notes**, type:

```
echo /usr/*/notes
```

The asterisk matches all the subdirectories of **/usr** that contain a file named **notes**.

In addition, a leading period in a file name must be matched explicitly. If you have a file in your current directory with the name **.profile**, the command

```
echo *file
```

does not match it.

These patterns can appear anywhere within a command or a command file.

Parameter Substitution

Each shell script can have up to nine *positional parameters*. This lets you write scripts that can be used for many circumstances. Recall that command parameters follow the command itself and are separated by tabs or spaces. An example of a command reference with two parameters is:

```
show first second
```

where **first** and **second** are the parameters.

To substitute the positional parameters in the script, use the character **$** followed by the decimal number of the parameter. Consider the following example. First, create two sample files:

```
cat > first
line 1
line two
line 3
<ctrl-D>
cat > second
line 1
line 2
line 3
<ctrl-D>
```

Then, issue the commands

```
cat first
cat second
diff first second
```

Inspect the output carefully. The command **diff** compares two files and prints all lines that differ. In this case, it prints **line two** and **line 2**.

Now, build the script **show**, which uses parameter substitution:

```
ed show
a
cat $1
cat $2
diff $1 $2
.
wq
chmod +x show
```

To demonstrate the effet of **show**, type:

```
show first second
```

Inspect the output and compare it with the output you received earlier.

If you issue the **show** command with fewer than the required number of parameters, the shell substitutes an empty string in its place. For example, using the command

```
show first
```

is equivalent to

```
cat first
cat
diff first
```

where the null string has been substituted for **$2**.

The example above shows the parameter references separated from each other by a space. In some uses, you may wish to prefix a substituted parameter to a name or a number. When more than one digit follows a **$**, The shell picks up the first digit as the number of the parameter. To illustrate, build the shell file **pos**:

```
ed
a
echo $167
.
w pos
q
chmod +x pos
```

Then call the script with

```
pos five
```

and the result will be:

```
five67
```

Shell Variable Substitution

In addition to positional parameters, the shell provides *variables*. You can assign values to variables, test them, and substitute them in commands.

The variable name can be built from letters, numbers, and the underscore character; for example:

```
high_tension
a
directory
167
```

Note that keywords must not be single digits, because the shell then treats them as positional parameters. Be aware that the shell treats upper-case and lower-case letters differently in variable names.

An assignment statement gives a value to a shell variable:

```
a=welcome
```

You can inspect their value with the **echo** command:

```
echo $a
```

The shell substitutes the value of the variable **a** in the **echo** command, which then appears as

```
echo welcome
```

COHERENT responds to this command by printing:

```
welcome
```

Don't forget the **$** when referring to the value.

Notice that the shell looks for special characters in any command that it sees — this includes the *space* character. To avoid problems, enclose the value to be assigned in apostrophes:

```
phrase='several words long'
```

There are several uses for variables. One is to hold a long string that you expect to type repeatedly as part of a command. If you are editing files in a subdirectory like

```
/usr/wisdom/source/widget
```

you can abbreviate if you set a variable **pw** to:

```
pw='/usr/wisdom/source/widget'
```

Then simply using **$pw** in a command

```
echo $pw
```

substitutes the long path name.

Another use of shell variables is as keyword parameters to commands. These then can be used the same way as positional parameters. To see how this works, create another script resembling **show**:

```
ed
a
cat $one
cat $two
diff $one $two
.
w show2
q
chmod +x show2
```

To use **show2**, issue:

```
one=first two=second show2
```

This is equivalent in effect to:

```
cat first
cat second
diff first second
```

Unlike positional parameters, keyword parameters may be several characters in length. If you want some text to follow immediately a keyword parameter, enclose the keyword parameter in braces. To illustrate this, build a command file called **brace**, as follows:

```
ed
a
echo 'with brace:' ${a}bc
echo 'without brace:' $abc
.
w brace
q
chmod +x brace
```

Call the command file with **a** set:

```
a=567 brace
```

The result is:

```
with brace: 567bc
without brace:
```

When used in this way, the keyword parameters must be assigned before the command and on the same line as

the command. In this case, the assignment of keyword parameters does not affect the variable after the command is executed. For example, if you type:

```
one=ordinal
one=first two=second show2
echo 'value of one is ' $one
```

echo produces:

```
value of one is ordinal
```

Variables set other than on the line of a command are not normally accessible to a script. To illustrate, build a parameter display script:

```
ed
a
echo 1 $1 2 $2 p1 $p1 p2 $p2
.
w pars
q
chmod +x pars
```

This will be used to show the behavior of parameters. The parameters to **echo** without a **$** help to read the output. To pass positional parameters, type:

```
pars ay bee
```

The output is:

```
1 ay 2 bee p1 p2
```

To pass keyword parameters, type:

```
p1=start p2=begin pars
```

The result is:

```
1 2 p1 start p2 begin
```

To illustrate that the setting of **p1** and **p2** did not "stick", type:

```
echo $p1 $p2 'to show'
```

echo replies:

```
to show
```

This indicates that **p1** and **p2** are not set.

Illustrating that variables set separately from a command are not seen by the command, type:

```
p1=outside1 p2=outside2
pars
```

This replies:

```
1 2 p1 p2
```

By using the **export** command, however, such variables can be made available to commands. The commands

```
export p1 p2
p1='see me' p2=hello
pars
```

produce:

```
1 2 p1 see me p2 hello
```

This indicates that after the **export** of **p1** and **p2**, they are available to other commands. Once a variable has appeared in an **export** command, its value can be changed without a need to **export** it again.

Command Substitution

By enclosing a command between ` characters, you can feed its output onto the command line of another command. For example

```
echo `ls`
```

echoes the output of the **ls** command.

Special Shell Variables

When you log into the COHERENT system, it sets the shell variable **HOME** to your *home* or default directory path. If your user name is **henry**, then the command

```
echo $HOME
```

on most systems prints:

```
/usr/henry
```

The change directory command **cd** sets the working directory to the path found in **HOME** if no argument is given.

The shell normally prompts you with **$** for commands, and with **>** if more information is needed. These two prompts are taken by the shell from the variables **PS1** and **PS2**. You can change these if you want different prompts, for example

```
PS1="Fred's Software Palace: "
PS2='!'
```

To have these take effect each time you log in, put the assignment statements in your **.profile** file.

The shell variable **PATH** lists the path names of directories that contain commands. To show the contents of **PATH**, type:

```
echo $PATH
```

It typically will show:

```
:/bin:/usr/bin
```

This means that the shell looks for a command first in the current directory, then in **/bin**, and, if not found there, then in **/usr/bin**. The path names are separated by ':'. This means that an empty string precedes the first ':', the current directory. Another common setting for **PATH** is:

```
:..:/bin:/usr/bin
```

This means that the shell seeks commands first in the current directory, then in '..' (the parent directory of the current directory), then in **/bin**, and finally in **/usr/bin**.

dot . : Read Commands

Similar to the command **sh** is the **.** command. The command

```
. cfil
```

causes the shell to read and execute commands from **cfil**.

This differs from the **sh** command in several respects. First, there's no way to pass parameters to **cfil** with the '.' command. Second, the **sh** command executes another shell to read the commands, whereas '.' simply reads the commands directly. Finally, all the string variables and parameters are accessible by **cfil**.

The command file **good.am** created earlier can be executed with:

```
. good.am
```

This has the same effect. Similarly, the '.' can itself be used within a command file:

```
ed
a
. good.am
lc
.
w third.sh
q
```

Then, the command

```
. third.sh
```

has the same result as the command:

```
sh third.sh
```

Values Returned by Commands

Most COHERENT commands return a value that indicates success or failure. For example, if **grep** cannot find your file, it issues a diagnostic message and returns a value that tells the shell that something went wrong. You can examine this value by typing the command:

```
echo $?
```

This tells you the value returned by the last command executed. Zero indicates success (true), whereas a non-zero value indicates failure (false). Note that this convention is the opposite of that in the C language (a fact that has led to confusion on occasion).

You can use the value returned by a command to affect decisions about executing other commands.

test: Condition Testing

For most commands, the return value is a side-effect of their operation. However, the **test** command's only task is to return a value. This command can test many conditions, and return a value to indicate whether the requested condition is true or false.

The command

```
test -f file01
```

returns true (zero) if **file01** exists and is not a directory. To check if a file is a directory, use:

```
test -d file01
```

test can also test strings. This is useful when you are using parameter substitution. To illustrate, build the following command:

```
ed
a
test $1 = $2
echo 'test 1 & 2 for equal:' $?
test $1 != $2
echo 'test 1 & 2 for not equal:' $?
.
w test.ed
q
chmod +x test.ed
```

Because the '=' is a parameter, be sure to surround it with space characters.

This command file tests its two parameters for equality. Try the commands:

```
test.ed one two
test.ed one one
```

The **test** command has many other options; see the Lexicon entry for **test** for details.

Executing Commands Conditionally

Type the following commands to create two files:

```
cat >file1
line one
line two
line three
<ctrl-D>
cat >file2
line one
two is different
line three
<ctrl-D>
```

Now, compare the files and print the return value:

```
cmp -s file1 file2
echo $?
```

The command **cmp** compares two files byte-by-byte; the **-s** option tells **cmp** merely to indicate whether the files were the same. The command

```
echo $?
```

prints **1** (false) because the files are not the same.

To process a second command based on the result returned by the first, type:

```
cmp -s file1 file2 || cat file2
```

The characters **||** signify that the following command **cat** should be executed if the **cmp** command returns a non-zero value, which it will for this example.

The two characters **&&** execute the command that follows them only if the preceding command returns true (zero).

To see how this works, create a third file with the command:

```
cp file1 file3
```

Type the command:

```
cmp -s file1 file3 && rm file3
```

This command removes **file3** if **cmp** indicates that **file1** and **file3** are identical. Because **cmp** is preceded by the copy command **cp**, the files **file1** and **file3** are identical, and so **file3** is removed.

Control Flow

Because the shell is a programming language as well as a program, it provides constructs for conditional execution and loops. These are **for**, **if**, **while**, **until**, and **case**. Also, a subshell can be executed within '(' and ')'.

for: Execute a Loop

The **for** construct processes a set of commands once for each element in a list of items.

To illustrate **for**, type the following commands to COHERENT:

```
for i in a b c
do echo $i
done
```

The items **a**, **b**, and **c** form the list of value that the variable **i** assumes. The shell executes **echo** with **i** assuming each value in turn. The result of these commands is:

```
a
b
c
```

Notice that after you type the line containing **for**, COHERENT prompts with a different character **>** (on most COHERENT systems). The shell does this to remind you that you must type more information. After you type the line containing **done**, the prompt again becomes **$**.

The **for** command is usually used within a script. Also, you can leave off the list of value to the index variable; when you do this, the shell by default uses the arguments typed on the script's command line as the values for the index variable. To illustrate, type:

```
ed
a
for i
do echo $i
echo '---'
done
.
w script.for
q
chmod +x script.for
```

The statement **for i** is equivalent to:

```
for i in $*
```

where **$*** means "all positional parameters". Notice that two commands are repeated for each value of **i**. Now, call **script.for** with the following command line:

```
script.for 1 2 3 4 test
```

The result is:

```
1
---
2
---
3
---
4
---
test
---
```

if: Execute Conditionally

if tests the result of a command and conditionally executes other commands based upon that result. It can be used instead of **&&** and **||**, as shown above. To demonstrate this, first type the command:

```
cp file1 file3
```

This creates **file3** (because we deleted it on the previous page). Then type:

```
if cmp -s file1 file2
then cat file3
fi
```

This means that the shell executes

```
cat file3
```

if **cmp** returns zero (true).

To get the same result as given by the previously illustrated command:

```
cmp -s file1 file3 && rm file3
```

with the **if** statement, also use **else**:

```
if cmp -s file1 file3
then
else rm file3
fi
```

The commands between **else** and **fi** are executed if the result of the command following the **if** is false or non-zero. Note that there is no command following *then*.

The **elif** statement lets you test several conditions with one **if** statement and act on the one that is true. In general terms,

```
if command1
then action1
elif command2
then action2
elif command3
then action3
else action4
fi
```

The items labeled *command* and *action* are both commands or lists of commands.

First, the shell executes **command1**. If the result is true, it performs **action1**. If the result from **command1** is not true, the shell then executes **command2**. If its result is true, then it performs **action2**. This process continues so long as none of the commands return a true result. If none of the command results are true, the action following the **else** is executed.

TUTORIALS

To illustrate **elif**, create a shell script that list on your terminal only one of the three file-name arguments. Use the command

```
test -f name
```

which returns true if *name* is an existing non-directory file.

```
ed
a
if test -f $1
then cat $1
elif test -f $2
then cat $2
elif test -f $3
then cat $3
else echo 'None are files'
fi
.
w cat.1
q
chmod +x cat.1
```

Now, let's exercise **cat.1**. Type:

```
cat.1 file1 file2 file3
cat.1 file3 file2 file1
cat.1 foo bar baz
```

Examine the results.

while: Execute a Loop

Another looping or repetitive shell statement is the **while** statement. The commands

```
while command1
do command2
done
```

first performs *command1*. If its result is true, *command2* is executed, and *command1* is again executed. This process continues until *command1* returns false (non-zero).

until: Another Looping Construct

The construct **until** resembles **while**. For example, the commands:

```
until command1
do command2
done
```

execute *command2* until *command1* returns true (zero).

case: Serial Conditional Execution

The **case** statement resembles the **if** statement in that it offers a multiple choice. To illustrate, type the following script, which lets you choose one of several ways to list the contents of a directory:

```
ed
a
case $1 in
     1) ls -l;;
     2) ls;;
     3) lc;;
     *) echo unknown parameter $1;;
esac
.
w dir
q
chmod +x dir
```

The words **case** and **esac** bracket the entire **case** statement. The effect of the command

```
dir 2
```

is equivalent to:

```
ls
```

Each choice within the **case** statement is indicated by a string followed by **)**:

```
2)
```

indicates what is to be executed if argument **$1** has the value **2**.

The strings that select the choices may be patterns. The choice '*)' signifies that a match can be made on any string. Notice that this resembles the use of * to substitute any file name. An expression of the form

```
[1-9])
```

in a **case** statement matches any digit from 1 through 9. A list of alternatives can be presented by separating the choices with a vertical bar:

```
a|b|c) command
```

Each command or command list in the case choice must be terminated by a double semicolon **;;**.

Summary

The shell is a command programming language that handles simple commands as well as complex commands that can iterate as well as make decisions. Three kinds of substitution are provided to increase the power of your commands.

For more information about the shell, see the tutorial for the shell that follows in this manual. For more information about a given command, see its entry in the Lexicon.

Note, too, that the COHERENT system also includes the Korn shell **ksh**. This is a superset of the Bourne shell described here, and has many features that you may find useful. For information about this shell, see the Lexicon entry for **ksh**.

Introduction to MicroEMACS

This section introduces MicroEMACS, the interactive screen editor for COHERENT.

What is MicroEMACS?

MicroEMACS is an interactive screen editor. An *editor* lets you type text into your computer, name it, store it, and recall it later for editing. *Interactive* means that MicroEMACS accepts an editing command, executes it, displays the results for you immediately, then waits for your next command. *Screen* means that you can use nearly the entire screen of your terminal as a writing surface: you can move your cursor up, down, and around your screen to create or change text, much as you move your pen up, down, and around a piece of paper.

These features, plus the others that will be described in the course of this tutorial, make MicroEMACS powerful yet easy to use. You can use MicroEMACS to create or change computer programs or any type of text file.

This version of MicroEMACS was developed by Mark Williams Company from the public-domain program written by David G. Conroy. This tutorial is based on the descriptions in his essay *MicroEMACS: Reasonable Display Editing in Little Computers*. MicroEMACS is derived from the mainframe display editor EMACS, created by Richard Stallman.

For a summary of MicroEMACS and its commands, see the entry for **me** in the Lexicon.

Keystrokes: <ctrl>, <esc>

The MicroEMACS commands use **control** characters and **meta** characters. Control characters use the *control* key, which is marked **Control** or **ctrl** on your keyboard. Meta characters use the *escape* key, which is marked **Esc**.

Control works like the *shift* key: you hold it down *while* you strike the other key. This tutorial represent it with a hyphen; for example, pressing the control key and the letter 'X' key simultaneously will be shown as follows:

```
<ctrl-X>
```

The **esc** key, on the other hand, works like an ordinary character. You strike it first, *then* strike the letter character you want. This tutorial does *not* represent the *Escape* codes with a hyphen; for example, it represents **escape X** as:

```
<esc>X
```

Becoming Acquainted with MicroEMACS

Now you are ready for a few simple exercises that will help you get a feel for how MicroEMACS works.

To begin, type the following command to COHERENT:

```
me sample
```

Within a few seconds, your screen will have been cleared of writing, the cursor will be positioned in the upper left-hand corner of the screen, and a command line will appear at the bottom of your screen.

Now type the following text. If you make a mistake, just backspace over it and retype the text. Press the carriage return or enter key after each line:

```
main()
{
        printf("Hello, world!\n");
}
```

Notice how the text appeared on the screen character by character as you typed it, much as it would appear on a piece of paper if you were using a typewriter.

Now, type **<ctrl-X><ctrl-S>**; that is, type **<ctrl-X>**, and then type **<ctrl-S>**. It does not matter whether you type capital or lower-case letters. Notice that this message has appeared at the bottom of your screen:

```
[Wrote 4 lines]
```

This command has permanently stored, or *saved*, what you typed into a file named **sample**.

Type the next few commands, which demonstrate some of the tasks that MicroEMACS can perform for you. These commands will be explained in full in the sections that follow; for now, try them to get a feel for how MicroEMACS works.

Type **<esc><**. Be sure that you type a less-than symbol '<', which on most keyboards is located just above the comma. Notice that the cursor has returned to the upper left-hand corner of the screen. Type **<esc>F**. The cursor has jumped forward by one word, and is now on the left parenthesis.

Type **<ctrl-N>**. Notice that the cursor has jumped to the next line, and is now just to the right of the left brace '{'.

Type **<ctrl-A>**. The cursor has jumped to the *beginning* of the second line of your text.

Type **<ctrl-N>** again. Now the cursor is at the beginning of the third line of the program, the **printf** statement.

Now, type **<ctrl-K>**. The third line of text has disappeared, leaving an empty space. Type **<ctrl-K>** again. The empty space where the third line of text had been has now disappeared.

Type **<esc>>**. Be sure to type a greater-than symbol '>', which on most keyboards is just above the period. The cursor has jumped to the space just below the last line of text. Now type **<ctrl-Y>**. The text that you erased a moment ago has reappeared, but in a new position on the screen.

By now, you should be feeling more at ease with typing MicroEMACS's *control* and *escape* codes. The following sections will explain what these commands mean. For now, exit from MicroEMACS by typing **<ctrl-X><ctrl-C>**, and when the message

```
Quit [y/n]?
```

appears type **y** and then **<return>**. This will return you to the shell.

Beginning a Document

This tutorial practices on file **example1.c**, **example2.c**, and **example3.c**. They are stored in the directory **/usr/src/example**. Before beginning, type the following commands to copy these files into the current directory and change their permissions:

```
cp /usr/src/sample/example?.c .
chmod +w example?.c
```

Now, type the following command to invoke MicroEMACS:

```
me example1.c
```

In a moment, the following text will appear on your screen:

```
/*
 * This is a simple C program that computes the results
 * of three different rates of inflation over the
 * span of ten years.  Use this text file to learn
 * how to use MicroEMACS commands
 * to make creating and editing text files quick,
 * efficient and easy.
 */
#include <stdio.h>
main()
{
      int i;                    /* count ten years */
      float w1, w2, w3;         /* three inflated quantities */
      char  *msg = " %2d\t%f %f %f\n";/* printf string */
      i = 0;
      w1 = 1.0;
      w2 = 1.0;
      w3 = 1.0;
      for (i = 1; i<= 10; i++) {
            w1 *= 1.07;         /* apply inflation */
            w2 *= 1.08;
            w3 *= 1.10;
            printf (msg, i, w1, w2, w3);
      }
}
```

When you invoke MicroEMACS, it *copies* that file into memory. Your cursor also moved to the upper left-hand

corner of the screen. At the bottom of the screen appears the *status line*, as follows:

```
-- Coherent MicroEMACS -- example1.c -- File: example1.c --
```

The word to the left, MicroEMACS, is the name of the editor. The word in the center, **example1.c**, is the name of the *buffer* that you are using. (We will describe later just what a buffer is and how you use it.) The name to the right is the name of the text file that you are editing.

Moving the Cursor

Now that you have read a text file into memory, you are ready to edit it. The first step is to learn to move the cursor.

Try these commands for yourself as we described them. That way, you will quickly acquire a feel for handling MicroEMACS's commands.

Moving the Cursor Forward

This first set of commands moves the cursor forward:

<ctrl-F>	Move forward one space
<esc>F	Move forward one word
<ctrl-E>	Move to end of line

To see how these commands work, do the following: Type the *forward* command **<ctrl-F>**. As before, it does not matter whether the letter '**F**' is upper case or lower case. The cursor has moved one space to the right, and now is over the character '*' in the first line.

Type **<esc>F**. The cursor has moved one *word* to the right, and is now over the space after the word **This**. MicroEMACS considers only alphanumeric characters when it moves from word to word. Therefore, the cursor moved from under the * to the space after the word **This**, rather than to the space after the *. Now type the *end of line* command **<ctrl-E>**. The cursor has jumped to the end of the line and is now just to the right of the **s** of the word **results**.

Moving the Cursor Backwards

The following summarizes the commands for moving the cursor backwards:

<ctrl-B>	Move back one space
<esc>B	Move back one word
<ctrl-A>	Move to beginning of line

To see how these work, first type the *backward* command **<ctrl-B>**. As you can see, the cursor has moved one space to the left, and now is over the letter **e** of the word **three**. Type **<esc>B**. The cursor has moved one *word* to the left and now is over the **t** in **three**. Type **<esc>B** again, and the cursor will be positioned on the **o** in **of**.

Type the *beginning of line* command **<ctrl-A>**. The cursor jumps to the beginnning of the line, and once again is resting over the '**/**' character in the first line.

From Line to Line

<ctrl-P>	Move to previous line
<ctrl-N>	Move to next line

These two commands move the cursor up and down the screen. Type the *next line* command **<ctrl-N>**. The cursor jumps to the space before the '*' in the next line. Type the *end of line* command **<ctrl-E>**, and the cursor moves to the end of the second line to the right of the period.

Continue to type **<ctrl-N>** until the cursor reaches the bottom of the screen. As you reached the first line in your text, the cursor jumped from its position at the right of the period on the second line to just right of the brace on the last line of the file. When you move your cursor up or down the screen, MicroEMACS tries to keep it at the same position within each line. If the line to which you are moving the cursor is not long enough to have a character at that position, MicroEMACS moves the cursor to the end of the line.

Now, practice moving the cursor back up the screen. Type the *previous line* command **<ctrl-P>**. When the cursor jumped to the previous line, it retained its position at the end of the line. MicroEMACS remembers the cursor's position on the line, and returns the cursor there when it jumps to a line long enough to have a character in that position.

Continue pressing **<ctrl-P>**. The cursor will move up the screen until it reaches the top of your text.

Repetitive Motion

Some computers repeat a command automatically if you *hold down* the control key and the character key. Try holding down **<ctrl-N>** for a moment, and see if it repeats automatically. If it does, that will speed moving your cursor around the screen, because you will not have to type the same command repeatedly.

Moving Up and Down by a Screenful of Text

The next two cursor movement commands allow you to roll forward or backwards by one screenful of text.

<ctrl-V>	Move forward one screen
<esc>V	Move back one screen

If you are editing a file with MicroEMACS that is too big to be displayed on your screen all at once, MicroEMACS displays the file in screen-sized portions (on most terminals, 22 lines at a time). The *view* commands **<ctrl-V>** and **<esc>V** allow you to roll up or down one screenful of text at a time.

Type **<ctrl-V>**. Your screen now contains only the last three lines of the file. This is because you have rolled forward by the equivalent of one screenful of text, or 22 lines.

Now, type **<esc>V**. Notice that your text rolls back onto the screen, and your cursor is positioned in the upper left-hand corner of the screen, over the character '**/**' in the first line.

Moving to Beginning or End of Text

These last two cursor movement commands allow you to jump immediately to the beginning or end of your text.

<esc><	Move to beginning of text
<esc>>	Move to end of text

The *end of text* command **<esc>>** moves the cursor to the end of your text. Type **<esc>>**. Be sure to type a greater-than symbol '>'. Your cursor has jumped to the end of your text.

The *beginning of text* command **<esc><** will move the cursor back to the beginning of your text. Type **<esc><**. Be sure to type a less-than symbol '<'. The cursor has jumped back to the upper left-hand corner of your screen.

These commands move you immediately to the beginning or the end of your text, regardless of whether the text is one page or 20 pages long.

Saving Text and Quitting

If you do not wish to continue working at this time, you should *save* your text, and then *quit*.

It is good practice to save your text file every so often while you are working on it. If an accident occurs, such as a power failure, you will not lose all of your work. You can save your text with the *save* command **<ctrl-X><ctrl-S>**. Type **<ctrl-X><ctrl-S>** that is, first type **<ctrl-X>**, then type **<ctrl-S>**. If you had modified this file, the following message would appear:

```
[Wrote 23 lines]
```

The text file would have been saved to your computer's disk. (MicroEMACS sends you messages from time to time. The messages enclosed in square brackets '[' ']' are for your information, and do not necessarily mean that something is wrong.) To exit from MicroEMACS, type the *quit* command **<ctrl-X><ctrl-C>**. This will return you to the shell.

Killing and Deleting

Now that you know how to move the cursor, you are ready to edit your text.

To return to MicroEMACS, type the command:

```
me example1.c
```

Within a moment, **example1.c** will be restored to your screen.

By now, you probably have noticed that MicroEMACS is always ready to insert material into your text. Unless you use the **<ctrl>** or **<esc>** keys, MicroEMACS assumes that whatever you type is text and inserts it onto your screen where your cursor is positioned.

The simplest way to erase text is simply to position the cursor to the right of the text you want to erase and backspace over it. MicroEMACS, however, also has a set of commands that allow you to erase text easily. These commands, *kill* and *delete*, behave differently; the distinction is important, and will be explained in a moment.

Deleting Vs. Killing

When MicroEMACS *deletes* text, it is erased completely and disappears forever. However, when MicroEMACS *kills* text, the text is copied into a temporary storage area in memory. This storage area is overwritten when you move the cursor and then kill additional text. Until then, however, the killed text is saved. This aspect of killing allows you to restore text that you killed accidentally, and it also allows you to move or copy portions of text from one position to another.

MicroEMACS is designed so that when it erases text, it does so beginning at the *left edge* of the cursor. This left edge is called the *current position*.

You should imagine that an invisible vertical bar separates the cursor from the character immediately to its left. As you enter the various kill and delete commands, this vertical bar moves to the right or the left with the cursor, and erases the characters it touches.

Erasing Text to the Right

The first two commands to be presented erase text to the *right*.

<ctrl-D>	Delete one character to the right
<del>	Delete one character to the right
<esc>D	Kill one word to the right

The keystrokes **<del>** and **<ctrl-D>** both delete one *character* to the right of the current position. **<esc>D** deletes one *word* to the right of the current position.

To try these commands, type **<ctrl-D>** or **<del>**. MicroEMACS erases the character '**/**' in the first line, and shifted the rest of the line one space to the left.

Now, type **<esc>D**. MicroEMACS erases the '*' character and the word **This**, and shifts the line six spaces to the left. The cursor is positioned at the *space* before the word **is**. Type **<esc>D** again. The word **is** vanishes along with the *space* that preceded it, and the line shifts *four* spaces to the left.

Remember that **<ctrl-D>** *deletes* text, but **<esc>D** *kills* text.

Erasing Text to the Left

You can erase text to the *left* with the following commands:

<backspace>	Delete one character to the left
<esc><backspace>	Kill one word to the left

To see how to erase text to the left, first type the *end of line* command **<ctrl-E>**; this will move the cursor to the right of the word **three** on the first line of text. Now, type **<backspace>**. The second **e** of the word **three** has vanished.

To erase the *word* that lies to the left of the cursor, type **<esc><backspace>**: that is, type **<esc>** and then type **<backspace>**. MicroEMACS defines a word as a string delimited by white space. For example, if you type **<esc><backspace>**, the rest of the word **three** vanishes, and the cursor moves to the white space that lies to the left of that word. If the cursor is at the beginning of a line, then this command kills the last word on the previous line of text.

Please note that erasing text with **<backspace>** *deletes* the text; that is, the text is thrown away and gone forever. Erasing text with **<esc><backspace>**, however, *kills* text; which means that the text is stored internally and can be retrieved. The distinction between *deleting* text and *killing* is described in detail below.

Erasing Lines of Text

Finally, the following command erases a line of text:

<ctrl-K>	Kill from cursor to end of line

This command kills a line of text, from the line beginning from immediately to the left of the cursor to the end of the line.

To see how this works, move the cursor to the beginning of line 2. Now, strike **<ctrl-K>**. All of line 2 has vanished and been replaced with an empty space. Strike **<ctrl-K>** again. The empty space has vanished, and the cursor is now positioned at the beginning of what used to be line 3, in the space before * **Use**.

Yanking Back (Restoring) Text

The following command allows you restore material that you have killed:

<ctrl-Y>	Yank back (restore) killed text

Remember that when you kill text, MicroEMACS temporarily stores it elsewhere. You can return this material to the screen by using the *yank back* command **<ctrl-Y>**. Type **<ctrl-Y>**. All of line 2 has returned; the cursor, however, remains at the beginning of line 3.

Quitting

When you are finished, do not save the text. If you do so, the undamaged copy of the text that you made earlier will be replaced with the present mangled copy. Rather, use the *quit* command **<ctrl-X><ctrl-C>**. Type **<ctrl-X><ctrl-C>**. On the bottom of your screen, MicroEMACS responds:

```
Quit [y/n]?
```

Reply by typing **y** and a carriage return. If you type **n**, MicroEMACS will return you to where you were in the text. MicroEMACS will now return you to the shell.

Block Killing and Moving Text

As noted above, text that is killed is stored temporarily within memory. You can yank killed text back onto your screen, and not necessarily in the spot where it was originally killed. This feature allows you to move text from one position to another.

Moving One Line of Text

You can kill and move one line of text with the following commands:

<ctrl-K>	Kill text to end of line
<ctrl-Y>	Yank back text

To test these commands, invoke MicroEMACS for the file **example1.c** by typing the following command:

```
me example1.c
```

When MicroEMACS appears, the cursor will be positioned in the upper left-hand corner of the screen.

To move the first line of text, begin by typing the *kill* command **<ctrl-K>** twice. Now, press **<esc>>** to move the cursor to the bottom of text. Finally, yank back the line by typing **<ctrl-Y>**. The line that reads

```
/* This is a simple C program that computes the results
```

is now at the bottom of your text.

Your cursor has moved to the point on your screen that is *after* the line you yanked back.

Multiple Copying of Killed Text

When text is yanked back onto your screen, it is *not* deleted from memory. Rather, it is simply copied back onto the screen. This means that killed text can be reinserted into the text more than once. To see how this is done, return to the top of the text by typing **<esc><**. Then type **<ctrl-Y>**. The line you just killed now appears as both the first and last line of the file.

The killed text will not be erased from its temporary storage until you move the cursor and then kill additional text. If you kill several lines or portions of lines in a row, all of the killed text will be stored in the buffer; if you are not careful, you may yank back a jumble of accumulated text.

Kill and Move a Block of Text

If you wish to kill and move more than one line of text at a time, use the following commands:

<ctrl-@>	Set mark
<esc>.	Set mark
<ctrl-W>	Kill block of text
<ctrl-Y>	Yank back text

If you wish to kill a block of text, you can either type the *kill* command **<ctrl-K>** repeatedly to kill the block one line at a time, or you can use the *block kill* command **<ctrl-W>**. To use this command, you must first set a *mark* on the screen, an invisible character that acts as a signal to the computer. The mark can be set with either **<esc>.** or **<ctrl-@>**.

Once the mark is set, you must move your cursor to the other end of the block of text you wish to kill, and then strike **<ctrl-W>**. The block of text will be erased, and will be ready to be yanked back elsewhere.

Try this out on **example1.c**. Type **<esc><** to move the cursor to the upper left-hand corner of the screen. Then type the *set mark* command **<ctrl-@>**. MicroEMACS will respond with the message

```
[Mark set]
```

at the bottom of your screen. Now, move the cursor down six lines, and type **<ctrl-W>**. Note how the block of text you marked out has disappeared.

Move the cursor to the bottom of your text. Type **<ctrl-Y>**. The killed block of text has now been reinserted.

When you yank back text, be sure to position the cursor at the *exact* point where you want the text to be yanked back. This will ensure that the text will be yanked back in the proper place. To try this out, move your cursor up six lines. Be careful that the cursor is at the *beginning* of the line. Now, type **<ctrl-Y>** again. The text reappeared *above* where the cursor was positioned, and the cursor has not moved from its position at the beginning of the line which is not what would have happened had you positioned it in the middle or at the end of a line.

Although the text you are working with has only 23 lines, you can move much larger portions of text using only these three commands. Remember, too, that you can use this technique to duplicate large portions of text at several positions to save yourself considerable time in typing and reduce the number of possible typographical errors.

Capitalization and Other Tools

The next commands perform a number of tasks to help with your editing. Before you begin this section, destroy the old text on your screen with the *quit* command **<ctrl-X><ctrl-C>**, and read into MicroEMACS a fresh copy of the program, as you did earlier.

Capitalization and Lowercasing

The following MicroEMACS commands automatically capitalize a word (that is, make the first letter of a word upper case), or make an entire word upper case or lower case.

<esc>C	Capitalize a word
<esc>L	Lowercase from cursor to end of word
<esc>U	Uppercase from cursor to end of word

To try these commands, do the following: First, move the cursor to the letter **d** of the word *different* on line 2. Type the *capitalize* command **<esc>C**. The word is now capitalized, and the cursor is now positioned at the space after the word. Move the cursor forward so that it is over the letter **t** in **rates**. Press **<esc>C** again. The word changes to **raTes**. When you press **<esc>C**, MicroEMACS capitalizes the *first* letter the cursor meets.

MicroEMACS can also change a word to all upper case or all lower case. (There is very little need for a command that will change only the first character of an upper-case word to lower case, so it is not included.)

Type **<esc>B** to move the cursor so that it is again to the left of the word **Different**. It does not matter if the cursor is directly over the **D** or at the space to its left; as you will see, this means that you can capitalize or lowercase a number of words in a row without having to move the cursor.

Type the *uppercase* command **<esc>U**. The word is now spelled **DIFFERENT**, and the cursor has jumped to the space after the word.

Again, move the cursor to the left of the word **DIFFERENT**. Type the *lowercase* command **<esc>L**. The word has changed back to **different**. Now, move the cursor to the space at the beginning of line 3 by typing **<ctrl-N>** then **<ctrl-A>**. Type **<esc>L** once again. The character "*' is not affected by the command, but the letter **U** is now lower case. **<esc>L** not only shifts a word that is all upper case to lower case: it can also un-capitalize a word.

The *uppercase* and *lowercase* commands stop at the first punctuation mark they meet *after* the first letter they find. This means that, for example, to change the case of a word with an apostrophe in it you must type the appropriate command twice.

Transpose Characters

MicroEMACS allows you to reverse the position of two characters, or *transpose* them, with the *transpose* command **<ctrl-T>**.

Type **<ctrl-T>**. MicroEMACS transposes the character that is under the cursor with the character immediately to its *left*. In this example,

```
* use this
```

in line 3 now appears:

```
* us ethis
```

The space and the letter **e** have been transposed. Type **<ctrl-T>** again. The characters have returned to their original order.

Screen Redraw

<ctrl-L> Redraw screen

Occasionally, while you are working on a text another COHERENT user will write or mail you a message. COHERENT will write the message directly on your screen, which scrambles your screen. A message sent from another user or a message from the COHERENT system is *not* recorded into your text; however, you may wish to erase the message and continue editing. The *redraw screen* command **<ctrl-L>** will redraw your screen to the way it was before it was scrambled.

Type **<ctrl-L>**. Notice how the screen flickers and the text is rewritten. Had your screen been spoiled by extraneous material, that material would have been erased and the original text rewritten.

The **<ctrl-L>** command also has another use: it can move the line on which the cursor is positioned to the center of the screen. If you have a file that contains more than one screenful of text and you wish to have that particular line in the center of the screen, position the cursor on that line and type **<ctrl-U><ctrl-L>**. You will see a prompt that says

```
Arg: 4
```

The meaning of this prompt is explained below; for now, ignore it and press **<return>**. Immediately, MicroEMACS redraws the screen and places the line you selected in the center of the screen.

Return Indent

<ctrl-J> Return and indent

You may often be faced with a situation in which, for the sake of programming style, you need to indent many lines of text: before every line you must tab the correct number of times before typing the text. These *block indents* can be a time-consuming typing chore. The MicroEMACS **<ctrl-J>** command makes this task easier. **<ctrl-J>** moves the cursor to the next line on the screen and automatically positions the cursor at the previous line's level of indentation.

To see how this works, first move the cursor to the line that reads

```
w3 *= 1.10:
```

Press **<ctrl-E>**, to move the cursor to the end of the line. Now, type **<ctrl-J>**.

As you can see, a new line opens up and the cursor is indented the same amount as the previous line. Type

```
/* Here is an example of auto-indentation */
```

This line of text begins directly under the previous line.

Word Wrap

<ctrl-X>F Set word wrap

Although you have not yet had much opportunity to use it, MicroEMACS will automatically wrap text that you are typing. Word-wrapping is controlled with the *word wrap* command **<ctrl-X>F**. To see how the word wrap command works, first exit from MicroEMACS by typing **<ctrl-X><ctrl-C>**; then reinvoke MicroEMACS by typing

```
me cucumber
```

When MicroEMACS re-appears, type the following text; however, do *not* type any carriage returns:

```
A cucumber should be
well sliced, and dressed
with pepper and vinegar,
and then thrown out, as
good for nothing.
```

When you reached the edge of your screen, a dollar sign was printed and you were allowed to continue typing. MicroEMACS accepted the characters you typed, but it placed them at a location beyond the right edge of your screen.

Now, move to the beginning of the next line and type **<ctrl-U>**. MicroEMACS will reply with the message:

```
Arg: 4
```

Type **30**. The line at the bottom of your screen now appears as follows:

```
Arg: 30
```

(The use of the *argument* command **<ctrl-U>** will be explained in a few minutes.) Now type the *word-wrap* command **<ctrl-X>F**. MicroEMACS will now say at the bottom of your screen:

```
[Wrap at column 30]
```

This sequence of commands has set the word-wrap function, and told it to wrap to the next line all words that extend beyond the 30th column on your screen.

The *word wrap* feature automatically moves your cursor to the beginning of the next line once you type past a preset border on your screen. When you first enter MicroEMACS, that limit is automatically set at the first column, which in effect means that word wrap has been turned off.

When you type prose for a report or a letter of some sort, you probably will want to set the border at the 65th column, so that the printed text will fit neatly onto a sheet of paper. If you are using MicroEMACS to type in a program, however, you probably will want to leave word wrap off, so you do not accidentally introduce carriage returns into your code.

To test word wrapping, type the above text again, without using the carriage return key. When you finish, it should appear as follows:

```
A cucumber should be well
sliced, and dressed with
pepper and vinegar, and then
thrown out, as good for
nothing.
```

MicroEMACS automatically moved your cursor to the next line when you typed a space character after the 30th column on your screen.

If you wish to fix the border at some special point on your screen but do not wish to go through the tedium of figuring out how many columns from the left it is, simply position the cursor where you want the border to be, type **<ctrl-X>F**, and then type a carriage return. When **<ctrl-X>F** is typed without being preceded by a **<ctrl-U>** command, it sets the word-wrap border at the point your cursor happens to be positioned. When you do this, MicroEMACS will then print a message at the bottom of your terminal that tells you where the word-wrap border is now set.

To re-word wrap the text between the cursor and the mark, type **<ctrl-X>H**.

If you wish to turn off the word wrap feature again, simply set the word wrap border to one.

Search and Reverse Search

When you edit a large text, you may wish to change particular words or phrases. To do this, you can roll through the text and read each line to find them; or you can have MicroEMACS find them for you. Before you continue, close the present file by typing **<ctrl-X> <ctrl-C>**; then reinvoke the editor to edit the file **example1.c**, as you did before. The following sections perform some exercises with this file.

Search Forward

<ctrl-S>	Search forward incrementally
<esc>S	Search forward with prompt

As you can see from the display, MicroEMACS has two ways to search forward: incrementally, and with a prompt.

An *incremental* search is one in which the search is performed as you type the characters. To see how this works, first type the *beginning of text* command **<esc><** to move the cursor to the upper left-hand corner of your screen. Now, type the *incremental search* command **<ctrl-S>**. MicroEMACS will respond by prompting with the message

```
i-search forward []
```

at the bottom of the screen.

We will now search for the pointer ***msg**. Type the letters ***msg** one at a time, starting with *. The cursor has jumped to the first place that a * was found: at the second character of the first line. The cursor moves forward in the text file and the message at the bottom of the screen changes to reflect what you have typed.

Now type **m**. The cursor has jumped ahead to the letter **s** in ***msg**. Type **s**. The cursor has jumped ahead to the letter **g** in ***msg**. Finally, type **g**. The cursor is over the space after the token ***msg**. Finally, type **<esc>** to end the string. MicroEMACS replies with the message

```
[Done]
```

which indicates that the search is finished.

If you attempt an incremental search for a word that is not in the file, MicroEMACS will find as many of the letters as it can, and then give you an error message. For example, if you tried to search incrementally for the word ***msgs**, MicroEMACS would move the cursor to the phrase ***msg**; when you typed 's', it would tell you

```
failing i-search forward: *msgs
```

With the *prompt search*, however, you type in the word all at once. To see how this works, type **<esc><**, to return to the top of the file. Now, type the *prompt search* command **<esc>S**. MicroEMACS responds by prompting with the message

```
Search [*msgs]:
```

at the bottom of the screen. The word ***msgs** is shown because that was the last word for which you searched, and so it is kept in the search buffer.

Type in the words **editing text**, then press the carriage return. Notice that the cursor has jumped to the period after the word **text** in the next to last line of your text. MicroEMACS searched for the words **editing text**, found them, and moved the cursor to them.

If the word you were searching for was not in your text, or at least was not in the portion that lies between your cursor and the end of the text, MicroEMACS would not have moved the cursor, and would have displayed the message

```
Not found
```

at the bottom of your screen.

Reverse Search

<ctrl-R>	Search backwards incrementally
<esc>R	Search backwards with prompt

The search commands, useful as they are, can only search forward through your text. To search backwards, use the reverse search commands **<ctrl-R>** and **<esc>R**. These work exactly the same as their forward-searching counterparts, except that they search toward the beginning of the file rather than toward the end.

TUTORIALS

For example, type **<esc>R**. MicroEMACS replies with the message

```
Reverse search [editing text]:
```

at the bottom of your screen. The words in square brackets are the words you entered earlier for the *search* command; MicroEMACS remembered them. If you wanted to search for **editing text** again, you would just press the carriage return. For now, however, type the word **program** and press the carriage return.

Notice that the cursor has jumped so that it is under the letter **p** of the word **program** in line 1. When you search forward, the cursor moves to the *space after* the word for which you are searching, whereas when you reverse search the cursor moves to the *first letter* of the word for which you are searching.

Cancel a Command

<ctrl-G> Cancel a search command

As you have noticed, the commands to move the cursor or to delete or kill text all execute immediately. Although this speeds your editing, it also means that if you type a command by mistake, it executes before you can stop it.

The *search* and *reverse search* commands, however, wait for you to respond to their prompts before they execute. If you type **<esc>S** or **<esc>R** by accident, MicroEMACS will interrupt your editing and wait for you to initate a search that you do not want to perform. You can evade this problem, however, with the *cancel* command **<ctrl-G>**. This command tells MicroEMACS to ignore the previous command.

To see how this command works, type **<esc>R**. When the prompt appears at the bottom of your screen, type **<ctrl-G>**. Three things happen: your terminal beeps, the characters **^G** appear at the bottom of your screen, and the cursor returns to where it was before you first typed **<esc>R**. The **<esc>R** command has been cancelled, and you are free to continue editing.

If you cancel an *incremental search* command, **<ctrl-S>** or **<esc-S>**, the cursor returns to where it was before you began the search. For example, type **<esc><** to return to the top of the file. Now type **<ctrl-S>** to begin an incremental search, and type **m**. When the cursor moves to the **m** in **simple**, type **<ctrl-G>**. Your cursor returns to the top of the file, where you began the search.

Search and Replace

<esc>% Search and replace

MicroEMACS also gives you a powerful function that allows you to search for a string and replace it with a keystroke. You can do this by executing the *search and replace* command **<esc>%**.

To see how this works, move to the top of the text file by typing **<esc><**; then type **<esc>%**. You will see the following message at the bottom of your screen:

```
Old string [m]
```

As an exercise, type **msg**, and then press **<return>**. MicroEMACS will then ask:

```
New string:
```

Type **message**, and press the carriage return. As you can see, the cursor jumps to the first occurrence of the string **msg**, and prints the following message at the bottom of your screen:

```
Query replace: [msg] -> [message]
```

MicroEMACS is asking if it should proceed with the replacement. Type a carriage return: this displays the options that are available to you at the bottom of your screen:

```
<SP>[,] replace, [.] rep-end, [n] dont, [!] repl rest <C-G> quit
```

The options are as follows:

Typing a space or a comma executes the replacement, and moves the cursor to the next occurrence of the old string; in this case, it replaces **msg** with **message**, and moves the cursor to the next occurrence of **msg**.

Typing a period '.' replaces this one occurrence of the old string and ends the search and replace procedure. In this example, typing a period replaces this one occurrence of **msg** with **message** and ends the procedure.

Typing the letter 'n' tells MicroEMACS *not* to replace this instance of the old string, but move to the next occurrence of the old string. In this case, typing 'n' does *not* replace **msg** with **message**, and the cursor jumps to the next place where **msg** occurs.

Typing an exclamation point '!' tells MicroEMACS to replace all instances of the old string with the new string automatically, without checking with you any further. In this example, typing '!' replaces all instances of **msg** with **message** without further queries from MicroEMACS. When you finish searching and replacing, MicroEMACS displays a message that tells how many replacements it made.

Typing **<ctrl-G>** aborts the search and replace procedure.

Saving Text and Exiting

This set of basic editing commands allows you to save your text and exit from the MicroEMACS program. They are as follows:

<ctrl-X><ctrl-S>	Save text
<ctrl-X><ctrl-W>	Write text to a new file
<ctrl-Z>	Save text and exit
<ctrl-X><ctrl-C>	Exit without saving text

You have used two of these commands already: the *save* command **<ctrl-X><ctrl-S>** and the *quit* command **<ctrl-X><ctrl-C>**, which respectively allow you to save text or to exit from MicroEMACS without saving text. (Commands that begin with **<ctrl-X>** are called *extended* commands; they are used frequently in the commands described later in this tutorial.)

Write Text to a New File

<ctrl-X> <ctrl-W>	Write text to a new file

If you wish, you can copy the text you are currently editing to a text file other than the one from which you originally read the text. Do this with the *write* command **<ctrl-X><ctrl-W>**.

To test this command, type **<ctrl-X><ctrl-W>**. MicroEMACS displays the following message on the bottom of your screen:

```
Write file:
```

MicroEMACS is asking for the name of the file into which you wish to write the text. Type **sample**. MicroEMACS replies:

```
[Wrote 23 lines]
```

The 23 lines of your text have been copied to a new file called **sample**. The status line at the bottom of your screen has changed to read as follows:

```
-- Coherent MicroEMACS -- example1.c -- File: sample --------------
```

The significance of the change in file name will be discussed in the second half of this tutorial.

Before you copy text into a new file, be sure that you have not selected a file name that is already being used. If you do, MicroEMACS will erase whatever is stored under that file name, and the text created with MicroEMACS will be stored in its place.

Save Text and Exit

Finally, the *store* command **<ctrl-Z>** will save your text *and* move you out of the MicroEMACS editor. To see how this works, watch the bottom line of your terminal carefully and type **<ctrl-Z>**. MicroEMACS has saved your text, and now you can issue commands directly to the shell.

Advanced Editing

The second half of this tutorial introduces the advanced features of MicroEMACS.

The techniques described here will help you execute complex editing tasks with minimal trouble. You will be able to edit more than one text at a time, display more than one file on your screen at a time, enter a long or complicated phrase repeatedly with only one keystroke, and give commands to COHERENT without having to exit from MicroEMACS.

Before beginning, however, you must prepare a new text file. Type the following command:

```
me example2.c
```

In a moment, **example2.c** will appear on your screen, as follows:

```
/* Use this program to get better acquainted
 * with the MicroEMACS interactive screen editor.
 * You can use this text to learn some of the
 * more advanced editing features of MicroEMACS.
 */

#include <stdio.h>
main()
{
        FILE *fp;
        int ch;
        int filename[20];

        printf("Enter file name: ");
        gets(filename);

        if ((fp =fopen(filename,"r")) !=NULL) {
                while ((ch = fgetc(fp)) != EOF)
                        fputc(ch, stdout);
        } else
                printf("Cannot open %s.\n", filename);
        fclose(fp);
}
```

Arguments

Most of the commands already described in this tutorial can be used with *arguments*. An argument is a subcommand that tells MicroEMACS to execute a command a given number of times. With MicroEMACS, arguments are introduced by typing **<ctrl-U>**.

Arguments: Default Values

By itself, **<ctrl-U>** sets the argument at *four*. To illustrate this, first type the *next line* command **<ctrl-N>**. By itself, this command moves the cursor down one line, from being over the '**/**' at the beginning of line 1, to being over the *space* at the beginning of line 2.

Now, type **<ctrl-U>**. MicroEMACS replies with the message:

```
Arg: 4
```

Now type **<ctrl-N>**. The cursor jumps down *four* lines, from the beginning of line 2 to the letter **m** of the word **main** at the beginning of line 6.

Type **<ctrl-U>**. The line at the bottom of the screen again shows that the value of the argument is four. Type **<ctrl-U>** again. Now the line at the bottom of the screen reads:

```
Arg: 16
```

Type **<ctrl-U>** once more. The line at the bottom of the screen now reads:

```
Arg: 64
```

Each time you type **<ctrl-U>**, the value of the argument is *multiplied* by four. Type the *forward* command **<ctrl-F>**. The cursor has jumped ahead 64 characters, and is now over the **i** of the word **file** in the *printf* statement in line 11.

Selecting Values

Naturally, an argument does not have to be a power of four. You can set the argument to whatever number you wish, simply by typing **<ctrl-U>** and then typing the number you want.

For example, type **<ctrl-U>**, and then type **3**. The line at the bottom of the screen now reads:

```
Arg: 3
```

Type the *delete* command **<esc>D**. MicroEMACS has deleted three words to the right.

You can use arguments to increase the power of any *cursor movement* command, or any *kill* or *delete* command. The sole exception is **<ctrl-W>**, the *block kill* command.

Deleting With Arguments: An Exception

Killing and *deleting* were described in the first part of this tutorial. They were said to differ in that text that was killed was stored in a special area of the computer and could be yanked back, whereas text that was deleted was erased outright. However, there is one exception to this rule: any text that is deleted using an argument can also be yanked back.

To see how this works, first type the *begin text* command **<esc><** to move the cursor to the upper left-hand corner of the screen. Then, type **<ctrl-U> 5 <ctrl-D>**. The word **Use** has disappeared. Move the cursor to the right until it is between the words **better** and **acquainted**, then type **<ctrl-Y>**. The word **Use** has been moved within the line (although the spaces around it have not been moved). This function is very handy, and should greatly speed your editing.

Remember, too, that unless you move the cursor between one set of deletions and another, the computer's storage area will not be erased, and you may yank back a jumble of text.

Buffers and Files

Before beginning this section, replace the edited copy of the text on your screen with a fresh copy. Type the *quit* command **<ctrl-X><ctrl-C>** to exit from MicroEMACS without saving the text; then return to MicroEMACS to edit the file **example2.c**, as you did earlier.

Now, look at the status line at the bottom of your screen. It should appear as follows:

```
-- Coherent MicroEMACS -- example2.c -- File: example2.c --------------
```

As noted in the first half of this tutorial, the name on the left of the command line is that of the program. The name in the middle is the name of the *buffer* with which you are now working, and the name to the right is the name of the *file* from which you read the text.

Definitions

A *file* is a mass of text that has been given a name and has been permanently stored on your disk. A *buffer* is a portion of the computer's memory that has been set aside for you to use, which may be given a name, and into which you can put text temporarily. You can place text into the buffer either by typing it at your keyboard or by *copying* it from a file.

Unlike a file, a buffer is not permanent: if your computer were to stop working (because you turned the power off, for example), a file would not be affected, but a buffer would be erased.

You must *name* your files because you work with many different files, and you must have some way to tell them apart. Likewise, MicroEMACS allows you to *name* your buffers, because MicroEMACS allows you to work with more than one buffer at a time.

File and Buffer Commands

MicroEMACS gives you a number of commands for handling files and buffers. These include the following:

<ctrl-X><ctrl-W>	Write text to file
<ctrl-X><ctrl-F>	Rename file
<ctrl-X><ctrl-R>	Replace buffer with named file
<ctrl-X><ctrl-V>	Switch buffer or create a new buffer
<ctrl-X>K	Delete a buffer
<ctrl-X><ctrl-B>	Display the status of each buffer

Write and Rename Commands

The *write* command **<ctrl-X><ctrl-W>** was introduced earlier when the commands for saving text and exiting were discussed. To review, **<ctrl-X><ctrl-W>** changes the name of the file into which the text is saved, and then copies the text into that file.

Type **<ctrl-X><ctrl-W>**. MicroEMACS responds by printing

```
Write file:
```

on the last line of your screen.

Type **junkfile**, then **<return>**. Two things happen: First, MicroEMACS writes the message

```
[Wrote 21 lines]
```

at the bottom of your screen. Second, the name of the file shown on the status line changes from **example2.c** to **junkfile**. MicroEMACS is reminding you that your text is now being saved into the file **junkfile**.

The *file rename* command **<ctrl-X><ctrl-F>** allows you rename the file to which you are saving text, *without* automatically writing the text to it. Type **<ctrl-X><ctrl-F>**. MicroEMACS will reply with the prompt:

```
Name:
```

Type **example2.c** and **<return>**. MicroEMACS does *not* send you a message that lines were written to the file; however, the name of the file shown on the status line has changed from **junkfile** back to **example2.c**.

Replace Text in a Buffer

The *replace* command **<ctrl-X><ctrl-R>** allows you to replace the text in your buffer with the text from another file.

Suppose, for example, that you had edited **example2.c** and saved it, and now wished to edit **example1.c**. You could exit from MicroEMACS, then re-invoke MicroEMACS for the file **example1.c**, but this is cumbersome. A more efficient way is to simply replace the **example2.c** in your buffer with **example1.c**.

Type **<ctrl-X><ctrl-R>**. MicroEMACS replies with the prompt:

```
Read file:
```

Type **example1.c**. Notice that **example2.c** has rolled away and been replaced with **example1.c**. Now, check the status line. Notice that although the name of the *buffer* is still **example2.c**, the name of the *file* has changed to **example1.c**. You can now edit **example1.c**; when you save the edited text, MicroEMACS will copy it back into the file **example1.c** unless, of course, you again choose to rename the file.

Visiting Another Buffer

The last command of this set, the *visit* command **<ctrl-X><ctrl-V>**, allows you to create more than one buffer at a time, to jump from one buffer to another, and move text between buffers. This powerful command has numerous features.

Before beginning, however, straighten up your buffer by replacing **example1.c** with **example2.c**. Type the *replace* command **<ctrl-X><ctrl-R>**; when MicroEMACS replies by asking

```
Read file:
```

at the bottom of your screen, type **example2.c**.

You should now have the file **example2.c** read into the buffer named **example2.c**.

Now, type the *visit* command **<ctrl-X><ctrl-V>**. MicroEMACS replies with the prompt

```
Visit file:
```

at the bottom of the screen. Now type **example1.c**. Several things happen. **example2.c** rolls off the screen and is replaced with **example1.c**; the status line changes to show that both the buffer name and the file name are now **example1.c**; and the message

```
[Read 23 lines]
```

appears at the bottom of the screen.

This does *not* mean that your previous buffer has been erased, as it would have been had you used the *replace* command **<ctrl-X><ctrl-R>**. MicroEMACS is still keeping **example2.c** "alive" in a buffer and it is available for editing; however, it is not being shown on your screen at the present moment.

Type **<ctrl-X><ctrl-V>** again, and when the prompt appears, type **example2.c**. **example1.c** scrolls off your screen and is replaced by **example2.c**, and the message

```
[Old buffer]
```

appears at the bottom of your screen. You have just jumped from one buffer to another.

Move Text From One Buffer to Another

The *visit* command **<ctrl-X><ctrl-V>** not only allows you to jump from one buffer to another: it allows you to *move text* from one buffer to another as well. The following example shows how you can do this.

First, kill the first line of **example2.c** by typing the *kill* command **<ctrl-K>** twice. This removes both the line of text *and* the space that it occupied. If you did not remove the space as well the line itself, no new line would be created for the text when you yank it back. Next, type **<ctrl-X><ctrl-V>**. When the prompt

```
Visit file:
```

appears at the bottom of your screen, type **example1.c**. When **example1.c** has rolled onto your screen, type the *yank back* command **<ctrl-Y>**. The line you killed in **example2.c** has now been moved into **example1.c**.

Checking Buffer Status

The number of buffers you can use at any one time is limited only by the size of your computer. You should create only as many buffers as you need to use immediately; this will help the computer run efficiently.

To help you keep track of your buffers, MicroEMACS has the *buffer status* command **<ctrl-X><ctrl-B>**. Type **<ctrl-X><ctrl-B>**. The status line moves up to the middle of the screen, and the bottom half of your screen is replaced with the following display:

```
C    Size Lines   Buffer         File
-    ---- -----   ------         ----
*     655    24   example1.c     example1.c
*     403    20   example2.c     example2.c
```

This display is called the *buffer status window*. The use of windows will be discussed more fully in the following section.

The letter **C** over the leftmost column stands for **Changed**. An asterisk indicates that that buffer has been changed since it was last saved, whereas a space means that the buffer has not been changed. **Size** indicates the buffer's size, in number of characters; **Buffer** lists the buffer name, and **File** lists the file name.

Now, kill the second line of **example1.c** by typing the *kill* command **<ctrl-K>**. Then type **<ctrl-X><ctrl-B>** once again. The size of the buffer **example1.c** shrinks from 657 characters to 595 to reflect the decrease in the size of the buffer.

To make this display disappear, type the *one window* command **<ctrl-X>1**. This command will be discussed in full in the next section.

Renaming a Buffer

One more point must be covered with the *visit* command. COHERENT does not allow you to have more than one file with the same name. For the same reason, MicroEMACS does not allow you to have more than one *buffer* with the same name.

Ordinarily, when you visit a file that is not already in a buffer, MicroEMACS creates a new buffer and gives it the same name as the file you are visiting. However, if for some reason you already have a buffer with the same name as the file you wish to visit, MicroEMACS stops and asks you to give a new, different name to the buffer it is creating.

For example, suppose that you wanted to visit a new *file* named **sample**, but you already had a *buffer* named **sample**. MicroEMACS would stop and give you this prompt at the bottom of the screen:

```
Buffer name:
```

You would type in a name for this new buffer. This name could not duplicate the name of any existing buffer. MicroEMACS would then read the file **sample** into the newly named buffer.

Delete a Buffer

If you wish to delete a buffer, simply type the *delete buffer* command **<ctrl-X>K**. This command allows you to delete only a buffer that is hidden, not one that is being displayed.

Type **<ctrl-X>K**. MicroEMACS will give you the prompt:

```
Kill buffer:
```

Type **example2.c**. Because you have changed the buffer, MicroEMACS asks:

```
Discard changes [y/n]?
```

Type **y**, then **<return>**. Now, type the *buffer status* command **<ctrl-X><ctrl-B>**. The buffer status window no longer shows the buffer **example2.c**. Although the prompt refers to *killing* a buffer, the buffer is in fact *deleted* and cannot be yanked back.

Windows

Before beginning this section, it will be necessary to create a new text file. Exit from MicroEMACS by typing the *quit* command **<ctrl-X><ctrl-C>**; then reinvoke MicroEMACS for the text file **example1.c** as you did earlier.

Now, copy **example2.c** into a buffer by typing the **visit** command **<ctrl-X><ctrl-V>**. When the message

```
Visit file:
```

appears at the bottom of your screen, type **example2.c**. MicroEMACS reads **example2.c** into a buffer, and shows the message

```
[Read 21 lines]
```

at the bottom of your screen.

Finally, copy a new text, called **example3.c**, into a buffer. (You copied this file into the current directory when you began this tutorial.) Type **<ctrl-X><ctrl-V>** again. When MicroEMACS asks which file to visit, type **example3.c**. The message

```
[Read 123 lines]
```

appears at the bottom of your screen.

The first screenful of text appears as follows:

```
/*
 * Factor prints out the prime factorization of numbers.
 * If there are any arguments, then it factors these.  If
 * there are no arguments, then it reads stdin until
 * either EOF or the number zero or a non-numeric
 * non-white-space character.  Since factor does all of
 * its calculations in double format, the largest number
 * which can be handled is quite large.
 */
#include <stdio.h>
#include <math.h>
#include <ctype.h>

#define NUL '\0'
#define ERROR 0x10 /* largest input base */
#define MAXNUM 200 /* max number of chars in number */

main(argc, argv)
int argc;
register char *argv[];

-- Coherent MicroEMACS -- example3.c -- File: example3.c --------------
```

At this point, **example3.c** is on your screen, and **example1.c** and **example2.c** are hidden.

You could edit first one text and then another, while remembering just how things stood with the texts that were hidden; but it would be much easier if you could display all three texts on your screen simultaneously. MicroEMACS allows you to do just that by using *windows*.

Creating Windows and Moving Between Them

A *window* is a portion of your screen that can be manipulated independent of the rest of the screen. The following commands let you create windows and move between them:

<ctrl-X>2	Create a window
<ctrl-X>1	Delete extra windows
<ctrl-X>N	Move to next window
<ctrl-X>P	Move to previous window

The best way to grasp how a window works is to create one and work with it. To begin, type the *create a window* command **<ctrl-X>2**.

Your screen is now divided into two parts, an upper and a lower. The same text is in each part, and the command lines give **example3.c** for the buffer and file names. Also, note that you still have only one cursor, which is in the upper left-hand corner of the screen.

The next step is to move from one window to another. Type the *next window* command **<ctrl-X>N**. Your cursor has now jumped to the upper left-hand corner of the *lower* window.

Type the *previous window* command **<ctrl-X>P**. Your cursor has returned to the upper left-hand corner of the top window.

Now, type **<ctrl-X>2** again. The window on the top of your screen is now divided into two windows, for a total of three on your screen. Type **<ctrl-X>2** again. The window at the top of your screen has again divided into two windows, for a total of four.

It is possible to have as many as 11 windows on your screen at one time, although each window will show only the control line and one or two lines of text. Neither **<ctrl-X>2** nor **<ctrl-X>1** can be used with arguments.

Now, type the *one window* command **<ctrl-X>1**. All of the extra windows have been eliminated, or *closed*.

Enlarging and Shrinking Windows

When MicroEMACS creates a window, it divides into half the window in which the cursor is positioned. You do not have to leave the windows at the size MicroEMACS creates them, however. If you wish, you may adjust the relative size of each window on your screen, using the *enlarge window* and *shrink window* commands:

<ctrl-X>Z	Enlarge window
<ctrl-X><ctrl-Z>	Shrink window

To see how these work, first type **<ctrl-X>2** twice. Your screen is now divided into three windows: two in the top half of your screen, and the third in the bottom half.

Now, type the *enlarge window* command **<ctrl-X>Z**. The window at the top of your screen is now one line bigger: it has borrowed a line from the window below it. Type **<ctrl-X>Z** again. Once again, the top window has borrowed a line from the middle window.

Now, type the *next window* command **<ctrl-X>N** to move your cursor into the middle window. Again, type the *enlarge window* command **<ctrl-X>Z**. The middle window has borrowed a line from the bottom window, and is now one line larger.

The *enlarge window* command **<ctrl-X>Z** allows you to enlarge the window your cursor is in by borrowing lines from another window, provided that you do not shrink that other window out of existence. Every window must have at least two lines in it: one command line and one line of text.

The *shrink window* command **<ctrl-X><ctrl-Z>** allows you to decrease the size of a window. Type **<ctrl-X><ctrl-Z>**. The present window is now one line smaller, and the lower window is one line larger because the line borrowed earlier has been returned.

The *enlarge window* and *shrink window* commands can also be used with arguments introduced with **<ctrl-U>**. However, remember that MicroEMACS will not accept an argument that would shrink another window out of existence.

Displaying Text Within a Window

Displaying text within the limited area of a window can present special problems. The *view* commands **<ctrl-V>** and **<esc>V** roll window-sized portions of text up or down, but you may become disoriented when a window shows only four or five lines of text at a time. Therefore, three special commands are available for displaying text within a window:

<ctrl-X><ctrl-N>	Scroll down
<ctrl-X><ctrl-P>	Scroll up
<esc>!	Move within window

Two commands allow you to move your text by one line at a time, or *scroll* it: the *scroll up* command **<ctrl-X><ctrl-N>**, and the *scroll down* command **<ctrl-X><ctrl-P>**.

Type **<ctrl-X><ctrl-N>**. The line at the top of your window has vanished, a new line has appeared at the bottom of your window, and the cursor is now at the beginning of what had been the second line of your window.

Now type **<ctrl-X><ctrl-P>**. The line at the top that had vanished earlier has now returned, the cursor is at the beginning of it, and the line at the bottom of the window has vanished. These commands allow you to move forward in your text slowly so that you do not become disoriented.

Both of these commands can be used with arguments introduced by **<ctrl-U>**.

The third special movement command is the *move within window* command **<esc>!**. This command moves the line your cursor is on to the top of the window.

To try this out, move the cursor down three lines by typing **<ctrl-U>3<ctrl-N>**; now type **<esc>!**. (Be sure to type an exclamation point '!', not a numeral one '1', or your window will vanish. The command **<esc>1** is explained below.) The line to which you had moved the cursor is now the first line in the window, and three new lines have scrolled up from the bottom of the window. You will find this command to be very useful as you become more experienced at using windows.

All three special movement commands can also be used when your screen has no extra windows, although you will not need them as much.

One Buffer

Now that you have been introduced to the commands for manipulating windows, you can begin to use windows to speed your editing.

To begin with, scroll up the window you are in until you reach the top line of your text. You can do this either by typing the *scroll up* command **<ctrl-X><ctrl-P>** several times, or by typing **<esc><**.

Kill the first line of text with the *kill* command **<ctrl-K>**. The first line of text has vanished from all three windows. Now, type **<ctrl-Y>** to yank back the text you just killed. The line has reappeared in all three windows.

The main advantage to displaying one buffer with more than one window is that each window can display a different portion of the text. This can be quite helpful if you are editing or moving a large text.

To demonstrate this, do the following: First, move to the end of the text in your present window by typing the *end of text* command **<esc>>**, then typing the *previous line* command **<ctrl-P>** four times. Now, kill the last four lines.

You could move the killed lines to the beginning of your text by typing the *beginning of text* command **<esc><**; however, it is more convenient simply to type the *next window* command **<ctrl-X>N**, which moves you to the beginning of the text as displayed in the next window. MicroEMACS remembers a different cursor position for each window.

Now yank back the four killed lines by typing **<ctrl-Y>**. You can simultaneously observe that the lines have been removed from the end of your text and that they have been restored at the beginning.

Multiple Buffers

Windows are especially helpful when they display more than one text. Remember that at present you are working with *three* buffers, named **example1.c**, **example2.c**, and **example3.c**, although your screen is displaying only the text **example3.c**. To display a different text in a window, use the *switch buffer* command **<ctrl-X>B**.

Type **<ctrl-X>B**. When MicroEMACS asks

```
Use buffer:
```

at the bottom of the screen, type **example1.c**. The text in your present window is replaced with **example1.c**. The command line in that window changes, too, to reflect the fact that the buffer and the file names are now **example1.c**.

Moving and Copying Text Among Buffers

It is now very easy to copy text among buffers. To see how this is done, first kill the first line of **example1.c** by typing the **<ctrl-K>** command twice. Yank back the line immediately by typing **<ctrl-Y>**. Remember, the line you killed has *not* been erased from its special storage area, and may be yanked back any number of times.

Now, move to the previous window by typing **<ctrl-X>P**, then yank back the killed line by typing **<ctrl-Y>**. This technique can also be used with the *block kill* command **<ctrl-W>** to move large amounts of text from one buffer to another.

Checking Buffer Status

The *buffer status command* **<ctrl-X><ctrl-B>** can be used when you are already displaying more than one window on your screen.

When you want to remove the buffer status window, use either the *one window* command **<ctrl-X>1**, or move your cursor into the buffer status window using the *next window* command **<ctrl-X>N** and replace it with another buffer by typing the *switch buffer* command **<ctrl-X>B**.

Saving Text From Windows

The final step is to save the text from your windows and buffers. Close the lower two windows with the *one window* command **<ctrl-X>1**. Remember, when you close a window, the text that it displayed is still kept in a buffer that is *hidden* from your screen. For now, do *not* save any of these altered texts.

When you use the *save* command **<ctrl-X><ctrl-S>**, only the text in the window in which the cursor is positioned is written to its file. If only one window is displayed on the screen, the *save* command will save only its text.

If you made changes to the text in another buffer, such as moving portions of it to another buffer, MicroEMACS would ask

```
Quit [y/n]:
```

If you answer '**n**', MicroEMACS will *save* the contents of the buffer you are currently displaying by writing them to your disk, but it will ignore the contents of other buffers, and your cursor will be returned to its previous position in the text. If you answer '**y**', MicroEMACS again will save the contents of the current buffer and ignore the other buffers, but you will exit from MicroEMACS and return to the shell. Exit from MicroEMACS by typing the *quit* command **<ctrl-X><ctrl-C>**.

Keyboard Macros

A *keyboard macro* is a set of MicroEMACS commands that are stored in memory and given a name. After you create a keyboard macro, you can execute it again and again just by typing its name. If you are revising a big file, you will find that keyboard macros are one of the most useful features in MicroEMACS, and one that you will use often.

The following table summarizes MicroEMACS's keyboard-macro commands:

<ctrl-X>(	Open a keyboard macro
<ctrl-X>)	Close a keyboard macro
<ctrl-X>E	Execute a keyboard macro
<ctrl-X>M	Rename a keyboard macro
<ctrl-X>I	Bind current macro as initialization macro

Creating a Keyboard Macro

To begin to create a macro, type the *begin macro* command **<ctrl-X>(**. Be sure to type an open parenthesis '(', not a numeral '9'. MicroEMACS will reply with the message

```
[Start macro]
```

Type the following phrase:

```
MAXNUM
```

Then type the *end macro* command **<ctrl-X>)**. Be sure you type a close parenthesis ')', not a numeral '0'. MicroEMACS will reply with the message

```
[End macro]
```

Move your cursor down two lines and execute the macro by typing the *execute macro* command **<ctrl-X>E**. The phrase you typed into the macro has been inserted into your text.

If you give these commands in the wrong order, MicroEMACS warns you that you are making a mistake. For example, if you open a keyboard macro by typing **<ctrl-X>(**, and then attempt to open another keyboard macro by again typing **<ctrl-X>(**, MicroEMACS will say:

```
Not now
```

Should you accidentally open a keyboard macro, or enter the wrong commands into it, you can cancel the entire macro simply by typing **<ctrl-G>**.

Execute a Macro Repeatedly

As with most MicroEMACS commands, you can use a keyboard macro with an argument to execute it repeatedly. For example, if you have defined a keyboard macro, then typing

<ctrl-U><ctrl-X>E

executes that macro four times. (Remember, four is the default value for **<ctrl-U>**.)

As described above, **<ctrl-U>** normally is used with a positive number, to indicate how often MicroEMACS should execute a given command or macro. With keyboard macros, however, you can use a special value for **<ctrl-U>**: -1. This tells MicroEMACS to repeatedly execute a keyboard macro until it fails.

For example, consider that you define the following keyboard macro:

<ctrl-S> foo **<ctrl-K>**

This macro searches for the string "foo", then kills the rest of line that that string is on. Now, when you type the command

<ctrl-U> -1 **<ctrl-X>E**

executes this macro until MicroEMACS can no longer find the string "foo"; it then quits.

Obviously, you should define your macro carefully before you execute it with this -1 option to **<ctrl-U>**; otherwise, you can commit tremendous mayhem on your file with one keystroke.

Replacing a Macro

To replace this macro with another, go through the same process. Type **<ctrl-X>(**. Then type the *buffer status* command **<ctrl-X><ctrl-B>**, and type **<ctrl-X>)**. Remove the buffer status window by typing the *one window* command **<ctrl-X>1**.

Now execute your keyboard macro by typing the **execute macro** command **<ctrl-X>E**. The *buffer status* command has executed once more.

Renaming a Macro

Many times during a long editing session, you will find that you use one keyboard macro, then use a second keyboard macro, then find that you need the first macro again. In previous releases of MicroEMACS, the only way to do this work was to type the first macro, replace it with the second macro, then retype the first macro when you need it again. The present edition of MicroEMACS, however, lets you define any number of keyboard macros, and save them by giving each one a unique "name" that is, its own unique keyboard binding.

To rename a keyboard macro that you have already created, use the *rename macro* command **<ctrl-X>M**. To see how this works, do the following: (1) Type **<ctl-X>(** to open the keyboard macro. (2) Type **<esc>S** xyz **<ctrl-U> <ctrl-D>** to fill the macro with something. (3) Finally, type **<ctrl-X>)** to close the macro.

Now, type **<ctrl-X>M**, to rename the macro. MicroEMACS will reply with the prompt:

```
enter keybinding for macro
```

Type **<esc>L**. This tells MicroEMACS to take the keyboard macro you created and link it to the keystrokes **<esc>L**.

Now, whenever you type **<esc>L**, MicroEMACS will execute **<esc>**s xyz **<ctrl-U> <ctrl-D>**. You can now define another keyboard macro without wiping out the one you have renamed. There is no theoretical limit to the number of keyboard macros you can create, although there are practical limits imposed by the amount of memory available to MicroEMACS.

Renaming Macros: A Few Caveats

Please note that if you name a keyboard macro with a keystroke that is already defined, MicroEMACS will no longer be able to access that keystroke's functionality.

For example, if instead of naming your new macro **<esc>L** you named it **<ctrl-A>**, then every time you typed **<ctrl-A>** MicroEMACS would execute **<esc>S** xyz **<ctrl-U> <ctrl-D>** and you would no longer be able to jump to the beginning of a line (which **<ctrl-A>** normally does).

The only exceptions are **<ctrl-X>**, **<esc>**, and the **<ctrl-X>R** command (described below), which MicroEMACS will not let you reassign. Obviously, you should be very careful when you assign a name to a keyboard macro, or you could easily clobber much of the editor's functionality.

Note, too, that MicroEMACS lets you define reflexive keybindings, but these never work. For example, if you named the above example macro **<ctrl-D>** instead of **<esc>L**, then every time you typed **<ctrl-D>** MicroEMACS would try to execute a macro that included **<ctrl-D>** in it. Obviously, this can tie MicroEMACS into knots in no time. Again, please be very careful when you assign names to keyboard macros.

The commands **<ctrl-X>S** and **<ctrl-X>L** let you save all named keyboard macros into a file, and restore them during a later editing session. These commands are described in the next section.

Setting the Initialization Macro

MicroEMACS allows one macro to be specified which will be executed every time MicroEMACS is invoked. This "initialization macro" can be set using the key sequence **<ctrl-X>I** and causes MicroEMACS to "bind" the currently defined macro to the initialization macro.

Flexible Key Bindings

As you have noticed by now, MicroEMACS works through standard *key bindings:* that is, one keystroke or combination of keystrokes tells MicroEMACS to perform a particular task. For example, typing **<ctrl-A>** tells MicroEMACS to move the cursor to the beginning of the line; typing **<ctrl-E>** tells MicroEMACS to move the cursor to the end of the line; and so on.

MicroEMACS allows you to change its key bindings, so you can bind a given keystroke or combination of keystrokes to a task other than the default one documented in this tutorial. In this way, you can reconfigure MicroEMACS so that it resembles another editor with which you are more familiar.

To perform this magic, MicroEMACS uses two tables for keybindings: a *default table* that is loaded at compile time and never changes, and a *dynamic table* that you can modify with MicroEMACS's keybinding commands.

The following table summarizes MicroEMACS's commands for flexible keybindings:

<ctrl-X>R	Replace one binding with another
<ctrl-X>X	Rebind metakeys
<ctrl-X>S	Save flexible bindings and macros into file
<ctrl-X>L	Load flexible bindings and macros from file

Changing a Keybinding

The *replace binding* command **<ctrl-X>R** replaces one binding with another. For example, if you wished to replace the *beginning of line* command **<ctrl-A>** with **<esc>Z**, you would do the following:

1. Type **<ctrl-X>R** to invoke the rebinding command.

2. When you see the prompt

```
Enter old keybinding
```

type the keybinding you wish to change in this case, **<ctrl-A>**.

3 When you then see the prompt

```
Enter new keybinding
```

type the keybinding to which you wish to change it in this case, **<esc>Z**.

Note that you cannot rebind the command **<ctrl-X>R** itself; otherwise, you would paint yourself into a corner. Also, note that if you rebind a command to itself (that is, if you type the same keybinding in response to both prompts), then that keybinding is bound to the old meaning of the keybindings, should there be any.

Rebinding Metakeys

MicroEMACS's keybindings depend on several pre-defined metakeys. A *metakey* is a keystroke that introduces a further set of commands. MicroEMACS's default keybindings use two metakeys: **<ctrl-X>** and **<esc>**. Other editors use other keystrokes as metakeys. If you wish to rebind a metakey, use the *rebind metakey* command **<ctrl-X>X**. This command prompts you to bind up to three metakeys, and the argument key **<ctrl-U>**.

For example, suppose that you wish to change the metakey **<ctrl-X>** to **<ctrl-Q>**. To begin, type the command **<ctrl-X>X**. You will see the prompt

```
Enter prefix character 1 or space
```

"Prefix character 1" is **<ctrl-X>** in the default bindings. Type **<ctrl-Q>**. You will then see the prompt:

```
Enter prefix character 2 or space
```

"Prefix character 2" is **<esc>** in the default bindings. Since you do not want to change it, type **<space>**. You will then see the prompt:

```
Enter prefix character 3 or space
```

There is no "prefix character 3" in the default bindings, but you can set a third one for your keybindings if you wish. Since (for the sake of this example) you do not wish to set one, type **<space>**. Finally, you will see the prompt:

```
Enter repeat code or space
```

The "repeat code" executes a command repeatedly; in this tutorial, it is often called the "argument key" or "argument command". Since (in this example) you do not wish to change it, type **<space>**.

Now that you have reset the **<ctrl-X>** metakey, you must now type **<ctrl-Q>** every time in place of **<ctrl-X>** throughout all of the MicroEMACS commands. For example, if you wished to change the metakey back from **<ctrl-Q>** to **<ctrl-X>**, you would have to type **<ctrl-Q>X** to invoke the *rebind metakey* command.

Note that because **<ctrl-Q>** already is bound in the MicroEMACS keybindings, when you rebind it the command to which it was bound is no longer available to you. However, if you un-rebind the key, then it automatically is rebound to its old command. In the above example, **<ctrl-Q>** is bound to the *insert literal character* command, which lets you insert control characters into your file. When you rebound the **<ctrl-X>** metakey to **<ctrl-Q>**, then the *insert literal character* command was no longer available to you. However, when you re-rebound the **<ctrl-Q>** metakey to **<ctrl-X>**, then **<ctrl-Q>** was automatically rebound to the *insert literal character* command.

To change the first prefix character back to **<ctrl-X>**, type **<ctrl-Q>X**, then enter **<ctrl-X>** when you see the prompt for prefix character 1. This restores the original metacharacter. Note, however, that the original function of **<ctrl-Q>** (which is to let you embed control characters within a file) is no longer available to you: MicroEMACS "forgot" its original function when you made **<ctrl-Q>** into a prefix character.

Save and Restore Keybindings

MicroEMACS lets you save your rebound keybindings into a file, and reload them during another editing session. To save your keybindings into a file, type the *save keybindings* command **<ctrl-X>S**. Try it. You will see the prompt:

```
Store bindings file:
```

Type the name of a file. MicroEMACS then writes its keybindings into that file. This command saves all named keyboard macros that you have created. It also saves other aspects of the MicroEMACS environment that you have set; for example, if you have turn on word-wrapping, that fact will be saved.

To restore a set of keybindings, use the *restore keybindings* command **<ctrl-X>L**. Try it. You will see the prompt:

```
Load bindings file:
```

Type the name of the file in which you saved MicroEMACS' keybindings and all named keyboard macros. MicroEMACS will then load them into memory for you.

These commands let you prepare several sets of customized keybindings and macros. You can customize keybindings to suit your preference, or create custom sets of macros to suit any number of specialized editing tasks.

By default, MicroEMACS checks for the existence of file **$HOME/.emacs.rc** and executes it if found. You can generate a copy of **.emacs.rc** using the save-keybindings command **<ctrl-X>S**. The MicroEMACS command-line option **-f** lets you specify an alternate file of keybindings macros. After it load **.emacs.rc**, MicroEMACS then executes the initialization macro, if one exists. For example, if you wish to use the set of keybindings saved in file **keybind** to edit file **textfile**, then you would type the following:

```
me -f keybind textfile
```

As you can see, MicroEMACS's system of keyboard macros and flexible key bindings help make it an extremely flexible and powerful editor.

Sending Commands to COHERENT

The only remaining commands you need to learn are the *program interrupt* commands **<ctrl-X>! and <ctrl-C>**. These commands allow you to interrupt your editing, give a command directly to the shell, and then resume editing without affecting your text in any way.

The command **<ctrl-X>!** allows you to send *one* command line (one command, or several commands plus separators) to the operating system. To see how this command works, type **<ctrl-X>!**. The prompt **!** has appeared at the bottom of your screen. Type **lc**. Observe that the directory's table of contents scrolls across your screen, followed by the message **[end]**. To return to your editing, simply type a carriage return. The *interrupt* command **<ctrl-C>** suspends editing indefinitely, and allows you to send an unlimited number of commands to the operating system. To see how this works, type **<ctrl-C>**. After a moment, the COHERENT system's prompt will appear at the bottom of your screen. Type **time**. The COHERENT system replies by printing the time and date. To resume editing, then simply type **<ctrl-D>**.

If you wish, you can suspend MicroEMACS's operation, tell the COHERENT system to invoke another copy of the MicroEMACS program, edit a file, then return to your previous editing. To see how this is done, type **<ctrl-C>**. When the prompt appears at the bottom of your screen, type

```
me example1.c
```

It doesn't matter that you are already editing **example1.c**. MicroEMACS will simply copy the **example1.c** file into a new buffer and let you work as if the other MicroEMACS program you just interrupted never existed.

Exit from this second MicroEMACS program by typing the *quit* command **<ctrl-X><ctrl-C>**. Then type **<ctrl-D>**. Your original MicroEMACS program has now been resumed. However, none of the changes you made in the secondary MicroEMACS program will be seen here.

It is not a good idea to use multiple MicroEMACS programs to edit the same program: it is too easy to become confused as to which edits were made to which version.

The only time this is advisable is if you wish to test to see how a certain edit would affect your text: you can create a new MicroEMACS program, test the command, and then destroy the altered buffer and return to your original editing program without having to worry that you might make errors that are difficult to correct.

Now type **<ctrl-X><ctrl-C>** to exit.

Compiling and Debugging Through MicroEMACS

MicroEMACS can be used with the compilation command **cc** to give you a reliable system for debugging new programs.

Often, when you're writing a new program, you face the situation in which you try to compile, but the compiler produces error messages and aborts the compilation. You must then invoke your editor, change the program, close the editor, and try the compilation over again. This cycle of compilation editing recompilation can be quite bothersome.

To remove some of the drudgery from compiling, the **cc** command has the *automatic*, or MicroEMACS option, **-A**. When you compile with this option, the MicroEMACS screen editor will be invoked automatically if any errors occur. The error or errors generated during compilation will be displayed in one window, and your text in the other, with the cursor set at the number of the line that the compiler indicated had the error.

Try the following example. Use MicroEMACS to enter the following program, which you should call **error.c**:

```
main() {
        printf("Hello, world!\n")
}
```

The semicolon was left off of the **printf()** statement, which is an error. Now, save the file and exit from MicroEMACS; then try compiling **error.c** with the following **cc** command:

```
cc -A error.c
```

You should see no messages from the compiler because they are all being diverted into a buffer to be used by MicroEMACS. Then MicroEMACS will appear automatically. In one window you should see the message:

```
3: error.c : missing ';'
```

and in the other you should see your source code for **error.c**, with the cursor set on line 3.

If you had more than one error, typing **<ctrl-X>>** would move you to the next line with an error in it; typing **<ctrl-X><** would return you to the previous error. With some errors, such as those for missing braces or semicolons, the compiler cannot always tell exactly which line the error occurred on, but it will almost always point to a line that is near the source of the error.

Now, correct the error by typing a semicolon at the end of line 2. Close the file by typing **<ctrl-Z>**. **cc** will be invoked again automatically.

cc will continue to compile your program either until the program compiles without error, or until you exit from MicroEMACS by typing **<ctrl-U>** followed by **<ctrl-X><ctrl-C>**.

The MicroEMACS Help Facility

MicroEMACS has a built-in help function. With it, you can ask for information either for a word that you type in, or for a word over which the cursor is positioned. The MicroEMACS help file contains the bindings for all library functions and macros included with COHERENT.

For example, consider that you are preparing a C program and want more information about the function **fopen**. Type **<ctrl-X>?**. At the bottom of the screen will appear the prompt

```
Topic:
```

Type **fopen**. MicroEMACS will search its help file, find its entry for **fopen**, then open a window and print the following:

```
fopen - Open a stream for standard I/O
#include <stdio.h>
FILE *fopen (name, type) char *name, *type;
```

If you wish, you can kill the information in the help window and yank it into your program to ensure that you prepare the function call correctly.

Consider, however, that you are checking a program written earlier, and you wish to check the call to **fopen**. Simply move the cursor until it is positioned over one of the letters in **fopen**, then type **<esc>?**. MicroEMACS will open its help window, and show the same information it did above.

To erase the help window, type **<esc>1**.

Where To Go From Here

For a complete summary of MicroEMACS's commands, see the entry for **me** in the Lexicon. The COHERENT system includes three other editors: the stream editor **sed**, the popular screen editor **vi**, and the interactive line editor **ed**. Each can help you accomplish editing tasks that may not be well suited for MicroEMACS. For more information on these editors, see their tutorials or check their entries in the Lexicon.

Introduction to the ed Line Editor

This tutorial introduces the interactive editor **ed**. It is intended both for readers who want a tutorial introduction to **ed**, and those who want to use specific sections as a reference.

Related tutorials include those for **sed**, the stream editor, and for **me**, the MicroEMACS screen editor. This tutorial assumes that you already understand the basics of using the COHERENT system, such as what a file is, what it means to edit text, and how to issue commands to the operating system. If you not yet know your way around the COHERENT system, we suggest that you first study the *Using the COHERENT System*, which appears in the front of this manual. It covers the basics of using COHERENT and introduces many useful programs.

Why You Need an Editor

A significant feature of computers is the capacity to store, retrieve, and operate upon information. A computer can store many different kinds of information: programs, computer commands and instructions, data for programs, financial information, electronic mail, or natural-language text (e.g., French, English) destined for a manuscript or book.

ed is a program with which you can enter and edit text on your computer. You can use **ed** to create or change computer programs, natural-language manuscripts, files of commands, or any other file that consists of text that you can read.

ed is designed to be easy to use, and requires little training to get started. The fundamental commands are simple, but have enough flexibility to perform complex tasks.

Learning To Use the Editor

Practice on your part will help you learn quickly. The following sections contain examples that illustrate each topic discussed. We strongly recommend that you type each example presented as you encounter it in the text. Even if you understand the concept presented, performing the example reinforces the lesson, and you will learn more quickly how to use **ed**.

In addition to reading the text and doing the examples as you encounter them in the text, try your own variations on the commands, and branch out on your own. Try things that you suspect might work, but are not shown as examples.

General Topics

This section presents the background information you will need to understand how **ed** works.

To help illustrate the discussion to follow, log into your COHERENT system and type the following commands:

```
ed
a
this is a sample
ed session
.
w test
q
```

This example calls **ed**, then uses the **a** command to add lines to the text kept in memory. The period signals the end of the additions. The **w** command writes the lines of text to file **test**, and the command **q** tells **ed** to return to COHERENT. You will notice that after you type the **w** command, **ed** will respond with

```
28
```

which is the number of characters in the file.

Thus, to enter **ed**, simply type

```
ed
```

and to exit, type

```
q
```

You can also exit by typing **<ctrl-D>**: that is, hold down the **control** key on your keyboard, and at the same time strike the **D** key.

Notice that you are issuing two different kinds of commands in the above example: the command **ed** is given to the COHERENT shell, to invoke the editor; the rest of the commands are given to the editor. After **ed** is given the **q** command, it exits, and following commands are processed by COHERENT.

ed, Files, and Text

ed works with one file at a time. With **ed**, you can create a file, add to a file, or change a file previously created.

As you use **ed** to create or change files, you will type both *text* and controlling *commands* into the editor. Text is, of course, the matter that you are creating or changing. Commands, on the other hand, tell **ed** what you want it to do. As you will see shortly, there is a simple way to tell **ed** whether what you are typing is text or commands.

ed has about two dozen commands. Almost every one is only one letter long. Although they may seem terse, they are easy to learn. You will appreciate the brevity of the commands once you begin to use **ed** regularly.

You must end each command to **ed** by striking the **<return>** key. This key is present on all terminals. However, the labeling of the key may vary. It may be called **newline**, **linefeed**, **enter**, or **eol**, and is larger than any key on the keyboard except for the space bar. This key will be called the **<return>** key in the remainder of this document.

Creating a File

The example shown above created a file. Here is another example of file creation — here, creating a file called **twoline**:

```
ed
a
Two line Example,
thank you.
.
w twoline
q
```

The letter **a** tells **ed** to add lines to the file. You are creating a new file with this example; and when **ed** creates a new file, it is initially empty. The **w** command writes the lines you have added to file **twoline**. The command **q** tells the editor that you are finished, whereupon it returns to COHERENT. You can use the COHERENT command **cat** to list the contents of the new file:

```
cat twoline
```

the reply will be:

```
Two line Example,
thank you.
```

Each command used here will be described in detail in later sections.

Changing an Existing File

Suppose that a manuscript file of yours needs a few spelling corrections. **ed** will help you make them. To begin, simply name the file to correct when you issue the COHERENT command:

```
ed filename
```

where *filename* stands for the name of the file that you wish to edit. For example, the following adds a line to the file *twoline*, which we just created:

```
ed twoline
$a
This is the third line of the file.
.
w
q
```

Listing the file with **cat** gives:

```
Two line Example,
thank you.
This is the third line of the file.
```

TUTORIALS

The command **$a** tells **ed** to add one or more lines at the end of the file.

Correcting the spelling of a misspelled word is easy with **ed**. You can rearrange groups of words in a manuscript, and you can move or copy larger portions of text, such as a paragraph, from one spot to another.

Working on Lines

ed uses the *line* as the basic unit of information; for this reason, it is called a *line-oriented* editor. A line is defined as a group of characters followed by an end-of-line character, which is invisible. When you type out a file on your terminal, each line in the file will be shown on your terminal as one line. The commands for **ed** are based upon lines. When you add material to a file, you will be adding lines. If you remove or change items, you will do so to groups of lines.

ed knows each line by its number. A line's number, in turn, indicates its position within the file: the first line is number 1, the second line is number 2, and so on.

ed remembers the line you worked on most recently. This can help shorten the commands you type, as well as reduce the need for you to remember line numbers. The line most recently worked on is called the *current* line. **ed** commands use a shorthand symbol for the current line: the period '**.**'.

Another shorthand symbol used in **ed** commands is **$**, which represents the number of the last line in the file.

Many of the **ed** commands operate on more than one line at a time. Groups of lines are denoted by a range of line numbers, which appears as a prefix to the command.

Error Messages

If you type a command to **ed** incorrectly, **ed** respond with:

```
?
```

This indicates that it has detected an error. Many times, this error will be evident to you when you review the command that you just typed.

If you do not see what the error is, you can get a more lengthy description by typing to **ed**:

```
?
```

It will reply with an error message.

Basic Editing Techniques

This section discusses in more detail the elementary techniques and commands that you need to use **ed**. With the material presented in this section, you will be able to do most basic editing tasks.

Again, it is recommended that you type each example. This will help you understand each example, as well as remember the technique it demonstrates.

Creating a New File

To begin, let us presume that you need to create an entirely new file named **first**. Perhaps you only want one line in the file, and it is to read

```
This is my first example
```

These are the steps that you will need to go through to create this file.

The first step is to invoke the **ed** program. To do this, simply type

```
ed
```

Remember that you must end each line of commands or text line by pressing the **<return>** key, because **ed** will not act upon it until you do. Thus, you invoke the editor by typing **ed** and a **<return>**. Notice that these two characters must be lower case.

ed is now ready for commands. The first command that you will use is the append command **a**. This tells **ed** to add lines to the text in memory, which will later be written to the file. The number of lines that **ed** can hold in memory depends upon the amount of memory in your computer. For editing very large files, you should use **sed**, the COHERENT stream editor, which is described in its own tutorial.

ed will continue to add lines until you type a line that contains *only* a period. While it is adding lines, **ed** does not recognize commands.

After you issue the **a** command, you can type the lines to be included, concluding with a line that consists only of a period. This special line signals **ed** that you want to stop appending lines. The information that you have typed so far is:

```
ed
a
This is my first example
.
```

Next, you must tell **ed** to write the edited text into a file. Do so by issuing the write command **w**, plus the name of the file that is to hold the edited text. For example, if you wish to store this example in a file named **first**, issue the command:

```
w first
```

ed will write the file and tell you how many characters were written, in this case 25.

Finally, to quit the editor issue the quit command:

```
q
```

The commands you type after this will be interpreted and acted upon by COHERENT.

Now, review the example in its entirety. First you invoked **ed** by typing **ed** at the COHERENT prompt. Then you issued the add command **a** to add lines to the file. added lines with the **a** command, and finished the adding by typing a line that consists only of a period. You then wrote the editing text into a file by issuing the write command **w**, and finally you exited from **ed** by issuing the quit command **q**. The complete example is:

```
ed
a
This is my first example
.
w first
q
```

ed replied to the **w** command by printing the number of characters it wrote into the file. After you typed **q** COHERENT prompted you for a command again.

Changing a File

Suppose that you wish to change the file that you have just created: you want to add two more lines to the file so that the original line will be sandwiched between the new lines. You want the file to contain:

```
Example two, added last
This is my first example
Example two, added first
```

You will do this with **ed** using two new commands.

Again, you start by telling COHERENT to run **ed**. This time, however, you must type the name of the file that you are changing after the characters **ed**:

```
ed first
```

ed will remember this file name for later use with the **w** command.

ed reads the file in preparation for editing, and tells you the number of characters that it read in, again 25.

After reading the file, **ed** automatically sets the current line to the last line read in.

Now, add the third line shown in the second example by entering:

```
a
Example two, added first
.
```

This resembles the first example. In that case, however, the file had no information, whereas now it does. How did **ed** know where to add the lines?

TUTORIALS

The **a** command adds lines after the *current line*. When **ed** reads a file, it initially sets the current line to the last line read in; therefore, the **a** command added the new line after the last line.

The current line is used implicitly or explicitly by most commands, so it is helpful to know where it is. In general, the current line is left at the last line **ed** has processed. If you lose track of the current line, you can ask **ed** to tell you where it is, as you will see shortly.

To add the very first line to the second example, you will use yet another command, the insert command **i**. This command is identical to the **a** command, except that it inserts lines *before* the current line rather than after it.

Another word about the current line. After an **a** command finishes, the current line is the last line added. Thus, after the addition of "Example two, added first" above, the current line is now the last line in the file. So, if you were to do the **i** command immediately, you would be adding lines just before the last line, which is not what you want to do.

Nearly every **ed** command is flexible enough to allow you to specify the line upon which the command is to operate. Now you can complete the second example:

```
1i
Example two, added last
.
```

The numeral **1** before the **i** tells **ed** to insert lines before the first line in the file. The line-number prefix is used frequently, and applies to nearly every command.

Now, to finish the second example and save it into the same file, type:

```
w
q
```

Note that the file name was left off the **w** command. **ed** remembers the name of the file that you began with, and uses that name if none is used with the **w** command. Therefore, the edited text is written back into file **first**. Note, too, that the previous contents of the file **first** are lost when you write the new file **first**. Alternatively, you can type:

```
w second
```

This leaves the contents of **first** unchanged and creates a new file called **second**.

In case you forget, **ed** can tell you the name of the file with which you began. Simply type the command:

```
f
```

If you had used **f** any time while working on this second example, **ed** would have replied:

```
first
```

Remember to use the **q** command to leave **ed** and return to COHERENT.

Printing Lines

As you use **ed** to edit a file, you will find it most useful to print sections of the file on your terminal. This helps you see what you have done (and sometimes what you have not done), and helps you pinpoint where you wish to make changes.

The print command **p** prints the current line unless you specify a line number.

Continuing with the example begun above, when you type the commands

```
ed first
p
```

ed replies by printing

```
Example two, added first
```

which is the last line in the file named **first** from the previous example.

Again, like the commands **i** and **a**, if you want **ed** to print a line other than the current one, just prefix the **p** command with a line number. Thus, if you want to print the second line in the file, type:

```
2p
```

ed will reply with:

```
This is my first example
```

If you wish to print more than one line of a file, you can tell **ed** to print a *range* of line numbers: type the numbers of the first and last lines you wish to see, separated by a comma. For example, to print all three lines in the second example, type:

```
1,3p
```

ed responds by printing all lines. This same principle applies to other commands. The print command can also appear after other commands such as **s** or **d**, which are discussed later in this section.

Abbreviating Line Numbers

ed recognizes some shorthand descriptions for certain line numbers. The number of the last line can be represented by the dollar sign **$**. Thus, the command

```
1,$p
```

prints every line in the file. The advantage of this shorthand is that the command as typed works for any file, regardless of its size. This construct of **1,$p** is used often enough that it has an abbreviation of its own:

```
*p
```

The number of the current line can also be abbreviated by using the period or dot in the place of a line number. To print all lines from the beginning of the file through the current line, type:

```
1,.p
```

To print all lines from the current line through the end of the file, type:

```
.,$p
```

The special symbol **&** prints one screenful of text. Simply type:

```
&
```

This is equivalent to:

```
.,.+22p
```

If there are fewer than 23 lines between the current line and the end of the file, it is equivalent to

```
.,$p
```

All forms of the **p** command change the current line to the last line printed. The command

```
.,$p
```

after printing changes the current line to the last line of the file.

How Many Lines?

You can easily see the current line with **p**. Type:

```
p
```

This tells **ed** to print the current line. On your terminal, try the command:

```
.p
```

You will see that it does the same thing as **p**.

To discover how large your file is, just type:

```
=
```

ed will reply by typing the number of lines in the file.

To find the number of the current line, use the **dot equals** command:

```
.=
```

ed responds with the number of the current line.

TUTORIALS

Removing Lines

Editing means removing lines of text, as well as adding them. To illustrate how **ed** lets you remove lines of text, create another example file with **ed**:

```
ed
a
This is the first line.
The second line is good.
However, line three is bad.
line four wishes to go away.
line 5 similarly wants to be forgotten,
as does line 6.
the next to last line stays.
as does the last line in the file.
.
w example3
q
```

This creates a file named **example3**.

Now, you can practice removing lines that you no longer want. Begin editing the file by typing:

```
ed example3
```

Now, print the contents of the file by typing:

```
1,$p
```

Our first task is to delete lines 3 through 6. First, delete line 3, then print the entire file again.

```
3d
1,$p
```

and **ed** will respond with

```
This is the first line.
The second line is good.
line four wishes to go away.
line 5 similarly wants to be forgotten,
as does line 6.
the next to last line stays.
as does the last line in the file.
```

Notice that the original file's third line is no longer there. Line 3 is now what used to be line 4. Remember that the line numbers *always* begin with 1 for the first line of the file and progress consecutively even after the file has been changed. Thus, deleting a line will change the line number of each line from the deleted line to the the last line in the file.

You still need to remove three more lines. You can do this with one command:

```
3,5d
```

Again, type ***p** to print the contents of the file:

```
This is the first line.
The second line is good.
the next to last line stays.
as does the last line in the file.
```

Finally, write the updated file and quit:

```
w
q
```

This illustrates how to delete lines, both singly and in a group.

Abandoning Changes

Sometimes, you may make a mistake; rather than damage your file with badly edited text, you may wish to abandon what you have done and begin all over again. You can do so by using the **q** command in a different fashion than is shown above.

If you tell **ed** to **q** before you tell it to write the file with **w**, you abandon any changes made since beginning editing. However, to prevent you from accidentally selecting this option, **ed** checks to see if you have made any changes to the file; and if you have, it responds with a question mark '**?**'. To tell **ed** that you know what you are doing and really do wish to abandon the edited file, reply with a second **q**. **ed** will then quit and return you to COHERENT.

You can avoid the question mark prompt by typing the upper-case **Q** rather than lower-case **q**: **ed** will exit without regard to unsaved changes. You can also exit from **ed** by typing the end-of-file key **<ctrl-D>**.

Substituting Text Within a Line

If you type a line incorrectly, or later wish to rearrange some words or symbols within it, you know enough about **ed** now to do so. You only need to delete the line with the delete command **d** and re-type the line with the insert command **i**. To see how this is done, prepare the file **example4**, as follows:

```
ed
a
Software technology today has
adbanced to the point that large
software projects unherd of in
earlier times are undertaken and
.
w example4
q
```

This example has two misspelled words. We will correct each of them using different **ed** features.

The first method will be the direct way that you probably can anticipate. Give the following commands to the editor exactly as shown:

```
ed example4
2d
i
advanced to the point that large
.
```

These commands use the delete command **d** to delete the second line, and then uses the insert command **i** to insert the correct new line in its place.

Use the command

```
*p
```

to verify that the file now contains:

```
Software technology today has
advanced to the point that large
software projects unherd of in
earlier times are undertaken and
```

You can also use a second method to change the spelling of a word. This is the substitute command **s**. This command is very powerful, and probably is used more frequently than any other **ed** command.

The substitute command **s** is more complex than commands we have discussed so far, in that it has more elements, as follows: First is a line number or optional range of line numbers. Then comes the letter **s**, to invoke the substitute command itself. Third comes two *patterns* or *strings*, which are set off from the rest of the command and from each other with the **slash** character. For example:

```
1,$s/pattern1/pattern2/
```

Here, *pattern1* represents the string that you want **ed** to replace, and *pattern2* is the string that **ed** is to substitute in place of *pattern1*. Note that three slashes separate the two patterns from the **s**, from each other, and from the end of the line. These slashes must always be present.

With this command, you can correct the second spelling error in the example4:

```
3s/herd/heard/
p
```

ed replies:

```
software projects unheard of in
```

Note that these two command lines can be condensed to one:

```
3s/herd/heard/p
```

The meaning of these commands is: on the third line of the file, change **herd** to **heard** and, when finished, print the entire line. Without the **p** command, **ed** will change the line as you direct, but will not show you the new line. It is a good idea to print lines that you substitute in this manner until you gain in confidence with **ed**. Some **ed** experts always print the lines after substitution.

Type

```
.
w sample.text
q
```

to stop entering text, then save the newly typed text into file **sample.text** and exit ("quit") from **ed**.

After these two changes, the file looks like this:

```
Software technology today has
advanced to the point that large
software projects unheard of in
earlier times are undertaken and
```

Although the above example substitutes one word for another, note that the **s** command can replace any consecutive group of characters with any other: it may be one word, several words (including the space characters that separate them), or a fragment of a word.

Because **ed** looks for patterns rather words, you should keep in mind that it may find the wrong pattern. For example, assume that the current line in a file is

```
let not rain fall on a parade
```

and instead you want to say:

```
let not rain fall on the parade
```

You command **ed** to:

```
s/a/the/p
```

and are shocked to discover that the result is:

```
let not rthein fall on a parade
```

A better command to give **ed** would have been a substitute command that substituted the letter **a** preceded and followed by a space:

```
s/ a / the /p
```

Another correct way to do this task is to indicate within the substitution command which of several possible matches within the line is to be substituted. In our example, it is actually the third **a** that we are trying to match, so we could have used the special form of the command

```
s3/a/the/p
```

to get **ed** to select the one we wanted.

Undoing Substitutions

If you did change **a** to **the** inappropriately, you can retract the substitution by issuing the undo command

```
u
```

before you move on to another current line.

To illustrate this, enter this example:

```
ed
a
let not rain fall on a parade
.
w undo
q
```

Now, perform the substitution with

```
ed undo
s/a/the/p
```

which will result in:

```
let not rthein fall on a parade
```

To retract the substitution, simply type:

```
u
p
```

This undoes the substitution and prints the result.

Note that the undo command undoes the substitution only on the current line. Remember that if your substitution command operated over a range of lines, when it finishes the current line is the last one upon which the substitution was made. Thus, if you made an inappropriate substitution over a range of lines, the undo command will fix only the last line.

Global Substitutions

As you saw with the above examples, the **s** command substitutes only the *first* occurrence of the requested pattern on a given line.

A different form of the substitute command finds every occurrence of the indicated string on a line. Simply add the letter **g** for *global* after the third slash in the substitute command, and **ed** finds and changes every one:

```
s/pattern1/pattern2/g
```

So, if the current line contains a phrase:

```
a rose is a rose is a rose
```

and we tell **ed** to substitute

```
s/a/the/g
```

the line is changed to:

```
the rose is the rose is the rose
```

Again, be careful that your command does not inadvertently match all or part of a word that you wish to keep untouched.

Special Characters

In its first two parts, the substitute command uses some special punctuation characters. They will be discussed below in detail. However, you should be aware of these characters and avoid them until you progress to the advanced section, for unless used properly, they will give you undesired results. The characters are:

```
[ ^ $ * . \ &
```

They are used in **ed** and other COHERENT programs to form complex patterns.

Ranges of Substitution

Perhaps you need to change several lines that have the same misspelling or need the same editorial change. **s** can do that for you also. Simply prefix the command **s** with the line-number range as you would do with **p**. Borrowing the "rose" example again, if the saying were typed:

TUTORIALS

```
a rose is
a rose is
a rose
```

then you could do the same change as before, but across the entire file by typing

```
1,$s/a/the/
```

Note that the **g** after the **s** command has been omitted here, because you know that the string that you want to change appears only once on each line.

If some of the lines do not have the string you want to change, **ed** will not complain that the string is missing. However, if none of the lines in the range has the requested string, **ed** will print a **?**.

Intermediate Editing

This section introduces the more advanced command features of **ed**. Although you have already learned enough about **ed** to become productive, this section covers additional features that will increase your editing power considerably.

This section discusses the following topics: relative line numbering, moving blocks of text, finding strings, using special characters in substitution and search commands, processing global commands, and marking lines.

Relative Line Numbering

As discussed in the previous section, most commands allow you to use line numbers to control their range of operation. Before the command you can enter a single line number; for example:

```
1p
```

This, of course, prints the first line of the file. You may also specify a *range* of line numbers, by entering two numbers separated by a comma. For example, if the file contains at least ten lines, the command

```
1,10p
```

prints the first ten lines of the file.

The period (dot) always represents the number of the current line. For example, to print the file from the first line through the current line, just type:

```
1,.p
```

A command used without a line number always acts on the current line only. For example, typing

```
p
```

is equivalent to typing:

```
.p
```

There is yet another level of shorthand to line numbering — the plus and minus characters. These characters indicate *offsets* from the current line. For example, the command

```
.+3p
```

prints the third line after the current line. Likewise, the command

```
.-1p
```

prints the line that precedes the current line. Note that using a line offset changes the current line to the one addressed. Thus, after the above command is executed, the current line will be the one that preceded the original current line.

You can abbreviate this notation still further by leaving out the dot. The commands

```
+p
-p
```

do the following: First, **ed** advances to the next line and prints it; then it backs up to the previous line (which was the original current line) and printing it.

You can place several of these commands on one line to move the current line multiple lines. To back up three lines and then print, type:

```
---p
```

Note that in the absence of any other command, **ed** defaults to the **p** command. Thus

```
---
```

is equivalent to

```
---p
```

and

```
5
```

is identical to:

```
5p
```

The print command has one more abbreviation. If **ed** is expecting a command from you and you type nothing except **<return>**, **ed** interprets this as a command to advance the current line to the next line and print it. This action is equivalent to

```
+
```

or

```
.+1
```

<return> is the shortest command in **ed**.

All of the abbreviations for line numbers can be used by other commands that expect a range of line numbers. For example, if you want to delete five lines centered about the current line, you could type:

```
.-2,.+2d
```

and you would get your wish.

Note that **ed** does not allow you to specify a line number that is beyond the range of the file; this is regardless of whether you are typing a line number or any form of abbreviated line numbering. For example, suppose the current line is the last line in the file and you type:

```
+
```

This tells **ed** to "advance one line then print"; however, this is impossible because you are at the last line of the file, so there is no next line to print. When you request an impossible line number, **ed** replies by printing a question mark. Note, however, that the current line is always be valid so long as the file has at least one line in it. Thus, unless the file is empty, the command

```
.
```

will never give an error message.

Changing Lines

Earlier, an example of spelling correction was solved two ways. The first way was the clumsy way of deleting a line and retyping the entire line. This strategy means much work to change a single letter, so the substitute command was introduced instead.

On occasion, however, it is handy to be able to change lines en masse — as was done by deleting then inserting. **ed** provides this power with the change command **c**. In general terms,

```
m,nc
new lines
to be inserted
.
```

removes lines *m* through *n*, and insert new lines up to the period in place of them.

Moving Blocks of Text

When handling text, you will often need to shift a block of text from one position to another. In a manuscript, for example, you may need to rearrange the order of paragraphs to increase clarity. In a program, you may need to rearrange the order in which procedures appear.

To allow you to do this easily, **ed** provides a move command **m** that moves a block of text from one point in the file to another.

m is different from the other commands that we have discussed so far, in that line numbers follow as well as precede the **m** command itself. The line number that follows the command gives the line *after* which the text is to be moved. So, the general form of the move command is

b,*e*m*d*

which means "move lines *b* through *e* to after line *d*".

To see how this works, first build the following file:

```
ed
a
    This is a paragraph of natural language
text. Due to stylistic considerations, it
really should be the second paragraph.
    If you can read this paragraph first,
the text has been properly arranged, and
our move example has been successfully done.
.
w example5
q
```

The file **example5** contains two paragraphs, each three lines long. We will now move the first paragraph to after the second paragraph.

You can do this in either of two ways: you can move the first paragraph to after the second paragraph, or you can move the second paragraph to before the first paragraph. Either gives the same result, but the commands are somewhat different. To shift the first paragraph to after the second paragraph, type:

```
ed example5
1,3m$
*p
Q
```

Remember that **$** always represents the last line in the file. The result is:

```
    If you can read this paragraph first,
the text has been properly arranged, and
our move example has been successfully done.
    This is a paragraph of natural language
text. Due to stylistic considerations, it
really should be the second paragraph.
```

To move the second paragraph to before the first, type:

```
4,6m0
```

Note that the destination is 0, which means that the text is to be moved to immediately after line 0. Because there is no line number 0, the move command interprets this to mean the beginning of the file.

Of course, in our small example, line number abbreviations and knowledge of the current line may be used in a number of different ways to perform exactly the same action. For example,

```
1,3m.
```

says to move lines 1 through 3 of the file to the line after the current line. When you invoke **ed**, it always sets the line number to the last line in the file. Thus, this form of the command has the same effect as the previous forms.

If the destination of a move command is not specified, **ed** assumes the current line. Therefore, the command

```
1,3m
```

also repositions the first paragraph correctly.

The move command changes the line numbers in the file, although the number of lines in the file remains the same. The different forms of the move command will, however, yield different settings for the current line.

After a move command, the current line becomes the number of the last line moved. Thus, if you moved the first paragraph to after the second paragraph, the current line will be reset to the last line in the file — the original line 3. However, if you moved the second paragraph to before the first paragraph, the current line would be reset to line 3 — which was originally the last line in the file.

Copying Blocks of Text

The transfer command **t** resembles the move command, except that it copies text rather than moving it. When you move text, it is erased from its original position. When you copy text, however, the text then appears both in its original position and in the position to which you copied it. **ed** uses the term *transfer* rather than *copy* because the command **c** is already used as the change command.

The form of the transfer command is as follows:

b,etd

This means to transfer (copy) the group of lines that begins with *b* and that ends with *e* (inclusive) to after line *d*.

After copying the text, **ed** sets the current line to the last line copied.

String Searches

The methods of line location that have been discussed to this point all involve line numbers. They specified an absolute line number, a relative line number, or a shorthand symbol such as **.** or **$**.

Often, however, line numbers are not useful, because there is no easy way to tell what number a line has, how many lines ago a block of text began, and so on.

ed's solution to this problem is to locate a line by asking **ed** to search for a pattern of text. **ed** begins searching on the line that follows the current line, and looks for a line that matches the specified pattern. If it finds a line that contains the requested pattern, **ed** resets the current line to that line.

If **ed** encounters the end of the file before it finds a match, **ed** jumps to the first line in the file, and continues its search from there. If it finds no match by the time it returns to the line where the search began, **ed** gives up and issues an error message — the question mark **?**. Remember, if you type a question mark in response to an error message, **ed** will tell you in more detail what the error is.

What does it mean to "match" a pattern? The simplest meaning is that two patterns are the same — the strings have exactly the same characters in exactly the same order. To see how this works, type the following to create file **example6**:

```
ed
a
    This is an example that we will
use for string searching.  There
is much natural language here as well
as some genuine arbitrary strings.
890,;+  foxtrot
qwertyuiop ##
.
w example6
q
```

Now, to locate and print any line contains the pattern **fox**, type:

```
ed example6
/fox/p
```

In response, **ed** prints the line:

```
890,;+  foxtrot
```

Also, you can use string expressions to print a range of lines. For example:

```
/This/,/much/p
```

This prints:

TUTORIALS

```
  This is an example that we will
use for string searching. There
is much natural language here as well
```

That is, it printed all lines from the first line that contains the pattern **This** through the first line that contains the pattern **much**.

Pattern searches can also be combined with relative line numbers. If you have a Pascal program file with several procedures in it, but you find that you need to rearrange the procedures, you can combine the power of the move command with the string searches.

```
PROCEDURE A;
...
...
PROCEDURE B;
...
...
PROCEDURE C;
```

Assume that the section of text that begins with **PROCEDURE A** should follow the line that contains **PROCEDURE B**. The following command moves the text properly:

```
/PROCEDURE A/,/PROCEDURE B/-1m/PROCEDURE C/-1
```

This commands **ed** (1) to locate the chunk of text that begins with a line containing the pattern **PROCEDURE A** and ends with the line just before the first line that contains the pattern **PROCEDURE B**, and then (2) move that text to just before the first line that contains the pattern **PROCEDURE C**. As you can see, you can pack a lot of information into one **ed** command.

Let's look at this command in more detail, to see exactly how it works. First, remember that the move command **m** is defined as

b,*e*m*d*

where *b* indicates the first line of the text to be moved, *e* indicates the last line of the text to be moved, and *d* indicates the line that the moved text is to follow. Thus, *b* corresponds to the number of the line that contains **PROCEDURE A** and is the first line of the procedure in question. *e*, however, corresponds to the line before the **PROCEDURE B** begins, by virtue of the -1. Here is an example of mixing pattern searches with relative line numbers, as mentioned above. Thus, you have found the beginning and ending lines of procedure A.

The final string search locates the first line of subroutine C. The move command normally moves text to **after** the given line; and because we wish to move the text to *before* the line that contains **PROCEDURE C**, we must include the -1 to move the text up one line.

Remembered Search Arguments

As discussed earlier, line numbers may be abbreviated in many ways. They may be entered as **.**, or **+**, or **-**, and certain combinations of these. With some commands, pressing **<return>** tells **ed** to use the current line number.

ed encourages you to abbreviate the search string. If you enter no string between the slashes in a search or substitution, then **ed** uses the last-used search string. A common use is in the global substitution command (which will be discussed in detail later in this section):

```
g/please remove this string/s// /p
```

This does not quite remove it, but replaces it with a blank. The last-used string can be specified by a string search, a substitute command, or a reverse string search (also discussed later in this section). Also, the remembered search argument may also be used in any one of these. You can use the remembered search feature to "walk" through the file, finding the next occurrence of a remembered search pattern.

Uses of Special Characters

As powerful as the line locator seems, some features are even even more powerful. These will be discussed in the Expert Editing section, below. However, these more powerful capabilities depend upon certain punctuation marks used in a special way. As you use the line locator (as well as the substitute command), be aware of these following characters:

```
[ ^ $ * . \ &
```

They have special significance to **ed** when they appear in a string search or a substitution pattern.

If you need to use one of these characters without invoking its special meaning, precede it with a backslash '\'. This tells **ed** not to interpret the character in a special way.

For example, to find a backslash character, type the search command:

```
/\\/
```

If any of these characters is to be used in another context, for example, within lines that you are adding with the **a** command, it should *not* be preceded with the backslash. Only use the backslash to hide the meaning when it appears within the string search command, or within the first part of the substitution command.

Global Commands

The global commands **g** and **v** let you repeat commands on all lines within a specified range. For example, to print all lines that contain the word **example**, type:

```
g/example/p
```

The global command can prefix almost any command. For example, the following command deletes all lines that contain three consecutive plus signs:

```
g/+++/d
```

Likewise, the command

```
g/foxtrot/.-2,.+2p
```

prints the five lines that surrounds any line that contains the word **foxtrot**.

A common use of the global command is to perform global substitution. The command

```
g/PROCEDURE/s/PROCEDURE/PROC/gp
```

performs the substitution on each line that contains the string **PROCEDURE** and prints the resulting line.

This may appear similar to the command

```
1,$s/PROCEDURE/PROC/gp
```

but is different in that the global command prints each of the changed lines, whereas the substitute command prints only the last line changed. Also, the method of operation of these two commands is different.

A related command **v** performs much the same task, but executes the commands only for lines that do *not* contain the specified string. Thus, to print all the lines that do not have the letter **w**, use:

```
v/w/p
```

For more sophisticated uses of the **g** and **v** commands and how they work, see the section on Expert Editing.

Joining Lines

What do you do if you inadvertently hit **<return>** as you are adding lines and need to combine the two lines?

```
ed
a
Look out, I seem to have hit ret
urn in the
middle of a word and don't know
what to do!
.
w rid
q
```

Rather that retyping the entire line, you can use the join command **j**:

```
ed rid
1,2j
1,$p
```

This gives:

```
Look out, I seem to have hit return in the
middle of a word and don't know
what to do!
```

If no line number is specified, **j** joins the current line and the following line. If a single line number is specified, join operates on that and the following line.

Several lines can be joined by using the form of the command:

a,*b*j

This joins lines *a* through *b* into one line. Likewise, the command

```
1,$j
```

joins all the lines in the file into one line. Then, the command **.p** or **p** prints the entire file.

Note that the command

```
3j
```

does the same job as the command

```
3,4j
```

The join command generates its own second line number if none is specified, so that the command

*n*j

is equivalent to

n,*n*+1j

where **n** is a line number. This command is the only one that interprets a missing line number this way.

Splitting Lines

You can split one line into two with the substitute command **s**. To illustrate, suppose you typed in the following commands:

```
ed
a
This line wants to be two, with this second.
.
w split
q
```

To perform the split, type:

```
ed split
s/two, /two,\
/p
*p
w
q
```

The line split is caused by the backslash that precedes the **<return>**. This tells **ed** that the **<return>** does not terminate the command, but that it is part of the substitution. The contents of file **split** are now:

```
This line wants to be two,
with this second.
```

Marking Lines

As you edit a manuscript or program, it is sometimes handy to be able to leave a "bookmark" in the text for later reference. **ed** provides this feature with the mark command **k**. To mark the next line that has the word **find**, use

```
/find/ka
```

where the letter **a** is the mark. To print the line that has been so marked, use:

```
'ap
```

You can place these references anywhere that a line number is expected.

The mark must be one lower-case letter. Also, each mark is associated with one line. Marking a line with the **k** command does not change the current line.

Marks can be especially handy when you move paragraphs with the **m** command. They give you a chance to review the sections that you will be moving before you do the move.

For example, suppose that you have a manuscript with a paragraph that must be moved to a different part of the document. Create the following example:

```
ed
a
    This is a paragraph, first line, that
needs to be moved.
text
text
And this is the last sentence of the paragraph.
    Next paragraph begins here.
text
text
text
    This is the spot that we want the paragraph
to precede.
.
w example7
q
```

Now, place three marks to help with the move:

```
ed example7
/first line,/ka
/Next paragraph/kb
/is the spot/kc
```

This marks the first line to be moved with **a**, the line after the last to be moved with **b**, and the paragraph's destination with **c**. But you can see that the move command moves lines to the line *after* the third number specified, so let's change the third mark:

```
'c-1kc
```

Now we can use **c** in the move command without arithmetic. Now, print the paragraph to be moved to be sure that the marks are correct.

```
'a,'bp
```

ed replies with

```
    This is a paragraph, first line, that
needs to be moved.
text
text
And this is the last sentence of the paragraph.
    Next paragraph begins here.
```

You can see that we would move one line too many if we used the marks as they are. So, change **b** also.

```
'b-1kb
```

Now, do the move:

```
'a,'bm'c
1,$p
```

The file now contains:

```
    Next paragraph begins here.
text
text
text
    This is a paragraph, first line, that
needs to be moved.
text
text
And this is the last sentence of the paragraph.
    This is the spot that we want the paragraph
to precede.
```

Marking sections of text can increase the ease with which you solve your complex **ed** problems.

Searching in Reverse Direction

All scanning, processing, and searching has been shown going from the beginning of the file toward the end. Sometimes it is useful to find some word that occurs **before** the current line.

You can get **ed** to do string searching in the reverse direction by specifying the search with question marks **?** rather than slashes **/**. To find the previous occurrence of the word **last**, use:

```
?last?
```

This form of searching can be useful in finding the beginning and end of a **repeat/until** statement. For example, if the current line is in the middle of a Pascal **repeat/until** group, you can print the group with the command:

```
?repeat?,/until/p
```

The reverse search is like the forward search in every way except the direction of search. The search begins one line before the current or specified line, and proceeds toward the beginning of the file. If the string is not found by the time that the search reaches the beginning of the file, the search resumes at the end of the file, and progresses towards the starting point of the search. If the string is not found when the search reaches the original starting point, the question-mark error message is issued signifying no match.

Also, the command

```
??
```

uses the remembered search argument.

Expert Editing

This section describes the most advanced **ed** commands.

File Processing Commands

Earlier, we discussed the commands

```
ed
```

and:

```
ed filename
```

ed also has file-handling commands that go beyond those already discussed.

Suppose that you entered the command

```
ed file1
```

only to discover when you examined the contents of **file1** that you really wish to edit file **file2**. You could correct this error by exiting from **ed** and then re-invoking it for **file2**. However, **ed** command **e** lets you close out the current file and begin to edit a new file without exiting from the editor. For example, to stop editing **file1** and begin to edit **file2**, simply issue the command:

```
e file2
```

If you had made any changes to **file1**, **ed** will prompt you with a **?**, which is its way of asking if you wish to throw away the changes you have made to this file. If you immediately repeat the command, **ed** proceeds even if there are unsaved changes. The command

```
E new
```

commands **ed** to edit the new file, whether or not there are unsaved changes.

ed's "read" command **r** also reads a new file, but adds it to the file being edited instead of replacing the current file with it. This can be handy for copying one file into another one. For example, if you have a manuscript prefix stored in the file **prefix** to include the prefix at the beginning of the file you are editing, type:

```
0r prefix
```

r inserts the file being read after the line number specified; in this case, line 0 means at the beginning of the file. If used without a line number, **r** appends the newly read lines to the end of the file.

ed's command **w** writes the entire file if no line number is specified; however, you can specify line numbers. For example

```
1,3w new
```

writes the first three lines to file **new**. If the file name is omitted, the lines are written to the remembered file name.

The **w** command is unique in that it never changes the current line. This is true regardless of what line numbers are specified in the range for the command, or how those line numbers were developed.

The **W** command resembles the **w** command, except that it appends lines to the end of the file, whereas **w** creates a new file and erases any previous contents.

The **f** command prints the remembered file name that was set in

```
ed filename
```

or

```
e filename
```

or

```
w filename
```

commands. You can also use **f** to reset the remembered name, by typing:

```
f newname
```

This form of the command tells you what the new remembered file name is, even though you just typed it in.

Note that the command

```
w filename
```

changes the remembered name only if there is currently no remembered name, as does the **r** command.

Patterns

Earlier, you were cautioned that certain punctuation characters have special effect in search and substitute commands. These characters are:

```
[ ^ $ * . \ &
```

They are used to form powerful substitute and locator commands. An orderly combination of these special characters is called a *pattern*, sometimes called a *regular expression*. You can use a pattern to find or *match* a variety of strings with one search argument.

The simplest patterns use alphabetic characters and numeric digits, which match themselves. For example,

```
/ab/
```

finds and prints the next line containing the string **ab**.

The next simplest character to use in a pattern is the period or dot. It matches any character except the **newline** character that separates lines. Two periods in succession match any two consecutive characters, and so on. For example, if you have a file that contains algebraic statements of the form

```
a+b
c+e
a-b
a/b
d*e
```

and wanted to find and print any line involving **a** and **b** (in that order), then use the search statement:

```
/a.b/
```

The **.** in this example matches **+**, **-**, and **/**.

Then, you ask, how do I find a string that contains a period? For example, if you want to find all the sentences that ended with "lost." (that is, the word **lost** followed by a period), you might first try:

```
/lost./p
```

This, however, also matches the string "lost " (the word **lost** followed by a space), which is not what you want.

This is where the special character backslash comes in handy. A backslash tells **ed** to treat the next character as a regular character, even if it usually is a special character. Thus, to find "lost.", you need only type:

```
/lost\./p
```

This will not incorrectly find "lost ". If you want to find backslashes in your file, simply say:

```
/\\/p
```

Matching Many With One Character

The asterisk * matches an indefinite number of characters. For example, to remove extra spaces between words in a document, type

```
g/##*/s//#/p
```

(The character **#** has been substituted here for the space character to make the example more readable.) This replaces each series of spaces by one space.

Note that there are two spaces before the * in the search string. This is necessary because the * matches any length of string, including zero. Therefore, searching for a space followed by any number of spaces finds strings that are at least one space long.

The * matches the longest possible string of the previous character. This requires careful attention on your part, because the string matched by * might be longer than your required string, or even zero in length. Either way could give you unexpected results.

If you have a line

```
a+b-c
```

in your file and want to change it to

```
a+c
```

type the command:

```
s/a.*c/a+c/p
```

However, if the line read instead

```
a+b-c*d+c
```

and you applied the command, the result would be

```
a+c
```

since the **.*** matches the longest string between any **a** and any **c**.

Beginning and Ending of Lines

The characters **^** and **$** match, respectively, the beginning and ending of a line. Thus, you can find and print all lines that end with a bang:

```
g/bang$/p
```

or those that begin with a whimper:

```
g/^whimper/p
```

These two characters can also help you find lines of specific length. If you need to see all lines exactly five characters long, the command

```
g/^.....$/p
```

does the trick. To find and delete all blank lines, type:

```
g/^ *$/d
```

Note that this time the * matches a string of zero spaces. However, this is correct, because a blank line includes lines that have nothing in them, as well as lines that contain only spaces.

Replacing Matched Part

In many cases of substituting, you find yourself extending a word, or adding information to the end of a phrase. This can lead to extensive retyping of characters. The special **&** character can help out.

This character is special only when used in the right part, or *pattern2* of the substitute command. It means "the string that matched the left part". For example, to add **ing** to the word **help** in the current line, use:

```
s/help/&ing/
```

The ampersand may appear more than once in the right side.

This can be more interesting if the left part has a non-trivial *pattern*. For every word in a line that is preceded by two or more spaces, double the number of spaces before it:

```
s/###*/&&/gp
```

(Again, spaces have been replaced with # for clarity.)

Replacing Parts of Matched String

A more sophisticated feature, which is similar to the ampersand, helps you to rearrange parts of a line. To see how this works, create a file by typing:

```
ed
a
first part=second part
.
w eql
q
```

Two special bracket symbols, **\(** and **\)** can be used to delineate patterns in the left part of a substitution expression. Then, you can use the special symbols **\1**, **\2**, etc., to insert the delimited parts. The symbol **\(** marks the beginnning of the pattern, and **\)** marks the end. For example, to delete everything in the line except the characters to the left of the **=**, type

```
ed eql
s/^\(.*\)=.*/\1/p
Q
```

In the substitute command, the ^ matches the beginning of the line, **.*** matches "first part", and **=.*** matches the rest of the line. The symbol **\1** signifies the matched characters between the first **\(** (the only one in this example) and **\)**. The **p** then prints the result, which will be:

```
first part
```

To interchange the two parts, type

```
ed eql
s/\(.*\)=\(.*\)/\2=\1/
p
wq
```

The result is

```
second part=first part
```

The first portion of the substitution expression,

```
\(.*\)=\(.*\)
```

can be thought of as being in three parts. The first part

```
\(.*\)
```

matches all characters up to but not including the **=**, which are

```
first part
```

The second part

```
=
```

matches the **=** in the line, and finally the third part

```
\(.*\)
```

matches all characters following the "=", or

```
second part
```

The remainder of the substitution expression

```
\2=\1
```

which is the replacement part, rebuilds the line in interchanged order. The symbol **\2** replaces the matched string enclosed in the second pair of **\(\)** delimiters, and the symbol **\1** inserts the matched string enclosed in the first pair of **\(\)**.

The right side of the substitution inserts the second matched expression (from **\2**), then inserts the **=** sign again, followed finally with the first part of the line from **\1**.

This may appear involved, but can be immensely valuable in situations that require rearrangement of a large number of lines.

The next special characters for patterns that we will consider are the bracket characters **[** and **]**. These are used to define the character class. Inside the brackets, put a group of characters; **ed** will match any of them if it appears. For example, to print a line that contains any odd digit, say:

```
g/[13579]/p
```

For even more power and flexibility, you can combine character classes with the asterisk. For example, the following command finds and prints all lines that contain a negative number followed by a period:

```
g/-[0123456789]*\./p
```

This matches lines containing the following example strings:

```
-1.
-666.
-3.7.77
```

You can also match all lower-case letters by listing them in brackets, but the following abbreviation simplifies this:

```
g/[a-z]/p
```

This can also be used for the negative number example above:

```
g/-[0-9]*\./p
```

Most special characters lose their original meaning within the brackets, but one of the special characters, **caret ^**, gets a new meaning. In this context, it matches all characters *except* those listed in the brackets. For example, the following pattern matches a string that begins with **K** and continues with any character except a number:

```
/K[^0-9]/
```

This matches:

```
KQ
KK
KK9
```

but not:

```
K7
kK0
```

Other special characters may be part of a character class, but lose their special meaning when used in that context. Remember, however, that if you want to match the right bracket, it must appear first in the list. So, to find all occurrences of special characters in the file, type:

```
g/[]^\.*[&]/p
```

Listing Funny Lines

The **p** command prints lines with graphic characters in them. It also prints lines with non-graphic (or *control*) characters, but these do not appear on the screen. For example, printing a line that contains the BEL character **<ctrl-G>** will ring your terminal's bell, but you will not see where the BEL character occurs within the line.

The **l** command behaves like the **p** command, except that it also decodes and prints control characters. For example, if you use the **l** command to print a line that containing the word **bell** followed by a BEL character, you would see:

```
bell\007\n
```

Note that "007" is the ASCII value for **<ctrl-G>**. (ASCII is the system of encoding characters within your computer; see **ASCII** in the Lexicon for the full ASCII table.) The **l** command displays the backspace character **<ctrl-H>** as a hyphen first overstruck with a **<** and then a newline, which appears as **\n** on your screen. It displays a tab character as a **-**, first overstruck with a **>** and followed by a newline character, which appears as **>\n**. If the line being listed with **l** is too long to be displayed on one line on your screen, **l** separates it into two lines, with the backslash character placed at the end of the first line to indicate the split.

All other features of the **p** command apply to the **l** command.

Keeping Track of Current Line

The most commonly used abbreviation in **ed** is the dot, or period, which stands for the current line. Many commands can change the value of the dot, and it is useful to you to be able to anticipate this change when using the abbreviation.

Different classes of commands affect the value of the dot in different ways; in general, however, the simple explanation is usually correct: the current line is the last line processed by the last command to be executed.

Consider, for example, how the substitution command **s** changes the current line:

```
1,$s/flow/change/
p
```

In this example, the current line will be the last line modified by the substitutions; and that will be the line that the **p** command prints.

The **w** command is an exception to this rule. It does not change the current line, regardless of any line range selection or how these ranges are developed.

The **r** command changes the current line to the last of the lines read.

The **d** command sets the current line to the line after the last line deleted unless the last line in the file was deleted, in which case the new last line becomes the current line.

The line insertion commands **i**, **c**, and **a** all leave the current line as the last line added. If no lines are added, however, their behaviors differ: **i** and **c** effectively back up the last line by one, whereas **a** leaves it the same.

When Current Line Is Changed

When the current line changes is also important. Normally, the current line does not change until the command is completed.

To illustrate, create a file **semi** by typing:

```
ed
a
begin
second
first
in between
second
last
.
w semi
q
```

Now, edit the file and type all lines from **first** to **second**:

```
ed semi
/first/,/second/p
Q
```

This will cause an error! The reason is that the search command begins with current line set to **$**, so "first" is found on line 3. But the search for "second" also begins with the current line set at **$**, and finds "second" on line 2. Thus, the command translates to

```
3,2p
```

which is clearly invalid.

To do what was intended, use the **semicolon ;** instead of the comma to separate the two searches. This forces **ed** to change the current line to be changed after the search for **first** rather than after the entire command. Thus, the commands

```
ed semi
/first/;/second/p
Q
```

are correct and will do what is intended. The result will be:

```
first
in between
second
```

The search for **first** still begins with the current line set at **$**. However, after it finds **first**, **ed** resets the current line to 3, and begins the search for **second** there, and succeeds on line 5.

Finally, to be sure of where the current line is, you can use the **p** command to show you the line; or you can have **ed** tell you the number of the current line by typing:

```
.=
```

To give you a perspective on where you are with respect to the end of the file, type

```
&=
```

and **ed** will tell you the number of the last line in the file.

You can put any line number expression before **=** and it will type the result. For example

```
/next/=
```

types the number of the next line to contain "next" (if there is one). The command **=** never changes the line number.

More About Global Commands

All the global commands discussed thus far have been followed by only one command — substitute, print, and delete. You can, however, put several commands after a global command, and execute each of those commands for each line that matches.

To change all occurrences of the word **cacophonous** to the word **noisy** and print the three lines that follow, issue the command:

```
g/cacophonous/s//noisy/\
.+1,.+3p
```

Here, the additional commands are separated by the backslash before the **<return>**. Several commands can be added, and all but the last need the backslash at the end.

This will work for the line-adding commands, as well. To insert a spelling warning before each line that contains the word **occurrance**, issue the command:

```
g/occurrance/i\
((the following line needs spelling check))\
.
```

Note that the last line of the **i** group can be entered without a backslash, in which case the line containing only the period must be omitted. This has the same effect as:

```
g/occurrance/i\
((the following line needs spelling check))
```

You should not depend upon the setting of the current line in any multiline global command. There are two reasons for this. First, if one of the commands is a substitute and the string is not found in the matched line, the current line will not be changed.

Second, the global command operates in two phases. The first part scans the file for lines that match the string argument. **ed** marks these lines internally in a manner similar to the **k** command. The second phase then executes the commands on each of the marked lines. Therefore, you cannot count on a string search following the **g** to set the current line number.

Again, the **v** command behaves in the same way, except that it selects lines that do *not* match the pattern.

Caution is advised when using remembered search arguments, for a similar reason. A search argument is remembered only if the search has been executed. Thus, in a command of the form

```
g/backup/s//reverse/\
s/backin /backing/
```

the first remembered search may use **backup** on some occasion, and "**backin**" on others. The reason for this is that the second phase of the **g** command begins with a remembered search argument of **backup.** After the second line of the multiline command executes, the remembered search argument is "**backin** ". This remains throughout the remainder of the second **g** phase.

Thus, it is recommended that you avoid remembered search arguments when using multiline global commands.

Issuing COHERENT Commands Within ed

While you are using **ed**, you can issue COHERENT commands by prefixing them with the **!** command.

This can be useful if, for example, you need to discover a file name while in the middle of an edit, and you want to find it without leaving **ed**. Thus, to list your directory while in **ed**, type:

```
!lc
```

ed sends the command to COHERENT and echoes a **!** character when the command is finished.

There is no limitation on the type of command that you may issue with this feature. It is even plausible that you want to start another **ed**.

For More Information

The Lexicon article on **ed** summarizes its commands and options. The COHERENT system also includes three other useful editors: **sed**, the stream editor; MicroEMACS, the screen editor; and **vi**, a clone of the standard UNIX screen editor. MicroEMACS and **sed** are introduced with their own tutorials, and each is summarized in the Lexicon.

TUTORIALS

Introduction to the sed Stream Editor

This is a tutorial for the COHERENT editor **sed**. It describes in elementary terms what **sed** does.

This guide is meant for two types of reader: the one who wants a tutorial introduction to **sed**, and the one who wants to use specific sections as references.

Related tutorials include *Using the COHERENT System*, which presents the basics of using COHERENT and introduces many useful programs, and the tutorials for the interactive line editor **ed** and for the screen editor MicroEMACS.

In a nutshell, **sed** edits files non-interactively; that is, **sed** applies your set of commands to every line of the file being edited. It is not meant to create a text, as you can do with **ed**, **me**, or **vi**. Rather, it lets you perform large, intricate transformations on a file of text, using commands that resemble those used by **ed** or **vi**'s colon-command mode.

Although **sed** is not as easy to control as **ed** or MicroEMACS, both of which are interactive, it can edit a large file very quickly. You can use **sed** to change computer programs, natural language manuscripts, command files, electronic mail messages, or any other type of text file.

One last point: **sed** normally writes its output to the standard output, which by default is your screen. To save its output into a file, use the shell's '>' operator to redirect the standard output into a file.

Getting to Know sed

sed is a text editor. It reads a text file one line at a time, and applies your set of editing commands to each line as it is read. Because it does not ask you for instructions after it executes each command, **sed** is a *non-interactive* text editor.

The advantages of **sed** are that it can readily apply the same editing commands to many files; it can edit a large file quickly; and it can readily be used with *pipes*. A pipe takes the product of one program and feeds it into another program for further processing. If you are unsure of how a pipe works, refer to *sh Shell Command Language Tutorial*.

sed resembles closely **ed**. **sed** and **ed** use almost all of the same commands, and locate lines in much the same way. However, there are important differences between **ed** and **sed**. **ed** is interactive: when you give **ed** a command from the keyboard, it executes that command immediately and then waits for you to enter the next command. **sed**, on the other hand, accepts your editing commands all at once, either from the keyboard or, more often, from a file you prepare; then, as it reads your text file one line at a time, it applies every command to every line of text. Therefore, *addressing* (that is, telling the program what commands should be applied to which lines) is much more important with **sed** than with **ed**.

Keep in mind, too, that **sed** does not change your original text file; rather, **sed** copies it, changes it, and sends the edited file either to the standard output or to another file that you name in the command line.

Getting Started

Here are a few exercises to introduce you to **sed**. Type them into your COHERENT system to get a feel for how **sed** works.

As explained above, **sed** applies a set of editing commands to your text file. To edit a file with **sed**, you must prepare three elements: (1) the text file that you wish to edit; (2) a command file (or *script*) that contains the **sed** commands you want to apply to the text file; and (3) a command line that tells the COHERENT system what you want done and with which files.

To begin, then, type the following text into your computer using the **cat** command. (Remember that **<ctrl-D>** is typed by holding down the *ctrl* key and simultaneously typing *D*.)

```
cat >exercise1
No man will be a sailor who has contrivance enough
to get himself into a gaol; for being in a ship is
being in a gaol, with the chance of being drowned.
<ctrl-D>
```

Now, type in the following **sed** script. This script will substitute *jail* for *gaol*:

```
cat >script1
s/gaol/jail/g
<ctrl-D>
```

The last step is to prepare the command line. The command line consists of the **sed** command, the options that tell **sed** where its instructions will be coming from (either from a file or directly from the command line), the name of the file to be edited, and where the edited file should be send. The general form of the command line is as follows:

```
sed [-n] [-e commands] [-f scriptname] textfile [>file]
```

The **-n** option will be explained below, in the section on *Output*. The **-e** option tells **sed** that *commands* follow immediately. The **-f** option tells **sed** that the commands are contained in the file *scriptname*. *textfile* is the name of the text file to be edited. The greater-than symbol '>' followed by a file name redirects the edited version of the text file into *file*; if this option is not used, the edited copy of the text file will be sent to the standard output.

In this example, a command script has been prepared, so the **-f** option will be used. Also, the edited text should appear on the terminal screen, so the '>' will not be used. Type the command line as follows:

```
sed -f script1 exercise1
```

The following text will appear on your screen:

```
No man will be a sailor who has contrivance enough
to get himself into a jail; for being in a ship is
being in a jail, with the chance of being drowned.
```

You can use **sed** not only to substitute one word for another, but to add lines to files, delete lines, and perform more involved editing. No matter how complex your **sed** editing becomes, though, **sed** will always use the basic format just described.

The next few sections describe **sed**'s basic commands.

Simple Commands

Type in the exercises exactly as shown and examine the results. Use the **cat** command to enter the command file as well as the input file. The edited text will appear on your terminal. Usually when you edit, you will want to redirect the edited text to a new file; however, for the exercises presented here, let the edited text appear on your terminal so you can examine the results immediately.

Substituting

The substitution command is used very often when editing. **sed**'s substitution command **s** resembles the same command in **ed**. Its form is as follows:

```
s/term1/term2/
```

This tells **sed** to substitute *term2* for *term1*. To correct a misspelled word, for example, use this command form:

```
s/mispel/misspell/
```

As written, this command changes only the first occurrence of **mispel** in each line of your text file. To change *every* occurrence of *mispel* in each line, add **g** (the **g**lobal option) at the end of the command:

```
s/mispel/misspell/g
```

If you want to change only the *third* occurrence of **mispel** on every line, put a **3** after the **s**:

```
s3/mispel/misspell/
```

When no digit follows the **s** and no **g** follows the command, only the first occurrence of the term in each line (should there be one) will be changed.

To practice the substitution, type the following file into your system (please include the misspellings):

```
cat >exercise2
From the Devils Dictionary:
Hemp, n. A plant from whose fiberous bark is made
an article of neckware which is frequently put on
after public speaking in the open air and prevents
the wearer from tking cold.
<ctrl-D>
```

Now, prepare the following **sed** script to correct the misspellings:

```
cat >script2
s/Devils/Devil's/
s/fiberous/fibrous/
s/tking/taking/
<ctrl-D>
```

Invoke **sed** with the following command:

```
sed -f script2 exercise2
```

The following will appear on your screen:

```
From the Devil's Dictionary:
Hemp, n. A plant from whose fiberous bark is made
an article of neckwear which is frequently put on
after public speaking in the open air and prevents
the wearer from taking cold.
```

To see how the **g** command and the number option work, prepare the following text file:

```
cat >exercise3
sd      sd      sd      sd
sd      sd      sd      sd
sd      sd      sd      sd
<ctrl-D>
```

The following **sed** script changes the *third* **sd** in each line to **sed**:

```
cat >script3
s3/sd/sed/
<ctrl-D>
```

Invoke **sed** with the following command line:

```
sed -f script3 exercise3
```

The following will appear on your screen:

```
sd      sd      sed     sd
sd      sd      sed     sd
sd      sd      sed     sd
```

To change *every* **sd** to **sed**, use the **g** option. Prepare the following **sed** script:

```
cat >script3a
s/sd/sed/g
<ctrl-D>
```

The following will appear on your screen:

```
sed     sed     sed     sed
sed     sed     sed     sed
sed     sed     sed     sed
```

The **g** command will be most useful for editing prose, when you have no way to tell how many times a given error will appear on a line. The numeric option will be most useful for editing tables and lists.

Selecting Lines

Each of the substitution commands given above will be applied to every input line. Unlike **ed**, there is no error message if no line of text contains *term1*.

TUTORIALS

In certain instances, however, you may wish to apply a particular command only to specific lines. Lines can be specified (or *addressed*) by *preceding* the command with the identifying line number. The following exercise demonstrates line selection. First, prepare the following text file:

```
cat >exercise4
When a man is tired of London,
he is tired of life; for there
is in London all that life can afford.
<ctrl-D>
```

To change the word **tired** to **fatigued** on line 2 only, prepare the following **sed** script:

```
cat >script4
2s/tired/fatigued/
<ctrl-D>
```

Begin the editing of your text file by typing the following command line:

```
sed -f script4 exercise4
```

The following will appear on your screen:

```
When a man is tired of London,
he is fatigued of life; for there
is in London all that life can afford.
```

Remember that to specify a line number, you must place the number *before* the command; but to specify the numeric option (that is, position within the line), you must place the number *after* the command.

You can define a *range* of lines to be edited. One way to do this is to list the first and last line numbers, separated by commas, of the block of text in question. For example, the following script will change **is** to **was** only in the first two lines of the text file you just prepared:

```
cat >script4a
1,2s/is/was/
<ctrl-D>
```

Entering the command line

```
sed -f script4a exercise4
```

will bring the following text to your screen:

```
When a man was tired of London,
he was tired of life, for there
is in London all that life can afford.
```

Note that the word **is** in line 3 was unaffected by the substitution command, because it lay outside the range of lines specified by the command.

You can also select lines by *patterns*. Patterns are *strings* (any collection of letters and numbers, such as a word) that can be combined with commands. A fuller description of *patterns* can be found in the tutorial for **ed**. Later on, when we show you other commands, you will see that line selection by pattern rather than by line number is quite useful.

You can use the end-of-file symbol '$' for line selection. When you use this symbol, you do not have to know the exact number of lines in your text file. For example, if you want to apply a substitution command from line 10 through the end of your text file, the command form is:

```
10,$s/term1/term2/
```

p: Print Lines

When **sed** edits a text file, the edited text is by default sent to the *standard output*, which usually is your terminal's screen. (As noted above, the edited text can be optionally redirected to another file by using the shell's '>' operator.) Normally, **sed** prints every line in the text file, whether the line is changed or not.

The next exercise will demonstrate these defaults. First, type in the following text file:

```
cat >exercise5
Bill            g7              r115
Nora            g8              r115
Steve           g7              r120
Ella            g8              r120
Dave            g7              r115
Robert          g8              r120
<ctrl-D>
```

Next, create a script that contains no commands, by typing:

```
cat >script5
<ctrl-D>
```

Now, execute this empty script:

```
sed -f script5 exercise5
```

Note that **sed** simply copied your text file to the screen, without changing it in any way.

This default, however, can be inconvenient if you want to print only a selected portion of a file. Fortunately, with a couple of commands you can control **sed**'s printing.

The command line option **-n** changes **sed**'s printing behavior. When you invoke **-n**, the text file no longer is printed automatically. **sed** prints only the lines specified by the **p** command. The **p** command makes **sed** print whatever line (or lines) to which it is applied. Use **-n** on the command line to stop **sed** from printing every line automatically; then use the **p** command in the script to target the lines you want to print. The following exercise will help you grasp this point. First, type in the following **sed** script:

```
cat >script5a
/g7/p
<ctrl-D>
```

Enter the command line:

```
sed -n -f script5a exercise5
```

and the following text will appear on your terminal:

```
Bill            g7              r115
Steve           g7              r120
Dave            g7              r115
```

sed prints only the records of the students in grade 7 (**g7**).

It is important to note the order, or *syntax*, of the **-n** and **-f** command line options. The correct order is to enter **-n**, then **-f**. (**-nf** or **-fn** are also acceptable.) If you type **-f** and then **-n**, however, all you will get is an error message.

When you use the **p** option with a **sed** command, **sed** will print every line of text in which that command makes a substitution. This can be useful, but if you are not careful it can also create some problems. **sed** normally prints every line in your text file, whether or not it is changed by your script, unless you specify the **-n** option in your command line. Therefore, if you *do not* use the **-n** option in your command line and you **do** use the **p** option with your **s** commands, every line that **sed** edits will be printed more than once.

The following script illustrates this point:

```
cat >script5b
s/g7/g8/gp
s/r115/r120/gp
<ctrl-D>
```

Now, execute it with the following command:

```
sed -f script5b exercise5
```

The result will look like this:

```
Bill            g8              r115
Bill            g8              r120
Bill            g8              r120
Nora            g8              r120
Nora            g8              r120
Steve           g8              r120
Steve           g8              r120
Ella            g8              r120
Dave            g8              r115
Dave            g8              r120
Dave            g8              r120
Robert          g8              r120
```

Bill and **Dave** were printed three times: the first time because the **p** option was specified after editing the grade number, the second time because the **p** option was specified after editing the room number, and the third time because the **-n** option was *not* used on the command line. **Steve** and **Nora** were printed twice: the first time because their lines were edited once each, and the second time because the **-n** option was not used on the command line. **Ella** and **Robert** appeared once because their lines were not edited at all and the **-n** option was not specified in the command line.

To get around this problem, use the **-n** option and use **p** only once, on the last substitution:

```
cat >script5c
s/g7/g8/g
s/r115/r120/gp
<ctrl-D>
```

When you enter the following command line

```
sed -n -f script5c exercise5
```

the new result will be:

```
Bill            g8              r120
Nora            g8              r120
Dave            g8              r120
```

The **w** command acts like the **p** command, except that matched lines are written to the file whose name follows the **w**. The following script shows the correct form:

```
cat >script5d
s/g8/g9/w grade.9
s/g7/g8/w grade.8
<ctrl-D>
```

When you execute script5d with this command:

```
sed -f script5d exercise5
```

the normal product will be seen produced at your terminal, and the edited lines will be written to files **grade.8** and **grade.9**. File **grade.8** will contain:

```
Bill            g8              r115
Steve           g8              r120
Dave            g8              r115
```

Note the order in which the two **s** commands were given. If their order were reversed, every text line with **g7** in it would have **g7** changed to **g8** by the first **s** command, then have this newly created **g8** changed to **g9** by the second **s** command. Thus, *all* the students would be shown to be in **g9**, and every text line would be printed into the file **grade.9**.

Line Location

When you edit a file with **sed**, it is hard to keep track of line numbers. As noted earlier, you can locate specific lines with **sed** by using patterns as *line locators*. To see how this works, type the following text file into your system:

```
cat >exercise6
From the Book of Proverbs:
As a door turneth upon his hinges, so the
slothful man turneth upon his bed.
A soft answer turneth away wrath:  but grievous
words stir up anger.
<ctrl-D>
```

Now, prepare the following **sed** script:

```
cat >script6
/door/,/bed/s/turneth/turns/
<ctrl-D>
```

Execute it by entering the following command line:

```
sed -f script6 exercise6
```

The text will appear on your terminal this way:

```
From the Book of Proverbs:
As a door turns upon his hinges, so the
slothful man turns upon his bed.
A soft answer turneth away wrath:  but grievous
words stir up anger.
```

Note that the word *turns* was substituted for the word *turneth* only in the first proverb, not the second. The reason is that the **s** command in this instance was preceded by the *patterns* **door** and **bed**. These told **sed** to begin making the substitution on the first line in which the word **door** appears, and to stop making the substitution with the first line in which the word **bed** appears. In the text file, the fourth line also contained the word **turneth**, but because it lay outside the range of line specified by the line locators, no substitution was made.

When **sed** locates the last line of a block of text that you have defined, it will immediately look for the next occurrence of the first line locator. If it finds that first line locator, it will then resume making the substitution to your file until it again finds the second line locator or comes to the end of the file, whichever occurs first. In this example, when **sed** found the word **bed**, it began to look again for the word **door**; and if it had found the word **door**, it would have resumed substituting **turns** for **turneth**.

Remember that, as explained earlier, line numbers can also be used as line locators. For example, the **sed** script

```
2,3s/turneth/turns/
```

would have produced the same changes as did the script with the pattern line locators prepared earlier.

Add Lines of Text

sed can add lines to your text file. To see how **sed** does this, first prepare the following text file:

```
cat >exercise7
From the Devil's Dictionary:
Syllogism, n. A logical formula consisting of a major
and a minor assumption and an inconsequent.
<ctrl-D>
```

Now, type in the following script:

```
cat >script7
3a\
Economy, n. Purchasing the barrel of whiskey you do not \
need for the price of the cow you cannot afford.
<ctrl-D>
```

When you implement the script:

```
sed -f script7 exercise7
```

you will see this result:

```
From the Devil's Dictionary:
Syllogism, n. A logical formula consisting of a major
and a minor assumption and an inconsequent.
Economy, n. Purchasing the barrel of whiskey you do not
need for the price of the cow you cannot afford.
```

The append command **a** added text *after* the third line of the file. You defined where the text went. Notice the backslash '\' at the end of the line with the **a** command. This indicates that the next line is part of the command. When you append several lines of text, each line but the last one to be added must end with a '\' as in our example.

Note that no other editing command, such as **s**, can affect any line added with **a**. These lines go directly to your screen, or to a file, should you be sending the edited text there, and are invisible to all other **sed** commands.

The insert command **i** works like the **a** command, except that it adds its lines *before* the addressed line, rather than after. The following script shows how the **i** command works:

```
cat >script7a
2i\
Peace, n. In international affairs, a period of cheating\
between two periods of fighting.
<ctrl-D>
```

Invoking it with this command:

```
sed -f script7a exercise7
```

produces this:

```
From the Devil's Dictionary:
Peace, n. In international affairs, a period of cheating
between two periods of fighting.
Syllogism, n. A logical formula consisting of a major
and a minor assumption and an inconsequent.
```

As with the **a** command, no substitutions or other changes are performed on lines added with **i**.

Note, too, that you can *bracket* a text line by using the **a** and **i** commands at the same time. Adding a line with either **a** or **i** does not change line numbers of the text file you are editing (although it does, of course, change the line numbers of the file **sed** writes).

Delete Lines

The **d** command deletes lines that you do not want in the edited text. The original file stays unchanged, of course.

Lines that match the address (be it a line number, range, or pattern) of a **d** command do not appear in the output. Exercise 8 illustrates the **d** command:

```
cat >exercise8
The sun was shining on the sea,
Shining with all his might.
He did his very best to make
The billows smooth and bright --
And this was odd, because it was
The middle of the night.
<ctrl-D>
```

Now, you have to define the lines to be deleted by matching them with a unique pattern or a line number. To delete lines 3 through 6, prepare this script:

```
cat >script8
/best/,/night/d
<ctrl-D>
```

The command:

```
sed -f script8 exercise8
```

generates this result:

```
The sun was shining on the sea,
Shining with all his might.
```

Note that when a line is deleted, no other commands are applied to it. Usually, if a **sed** script holds a number of commands, every one of those commands is applied to every line read from your text file; however, **sed** is logical enough to read the next text line immediately, should a **d** command delete the current line before the series of commands has finished.

Change Lines

The **c** command combines the **i** and **d** options. Text is inserted before the addressed lines, which are then deleted. To see how this command works, prepare the following text file:

```
cat >exercise9
Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.
<ctrl-D>
```

Now, type in the following script:

```
cat >script9
1,2c\
Twas brilliant, and the shining cove\
Did glare and glimmer in the wave;
<ctrl-D>
```

When you execute your script with the following command line:

```
sed -f script9 exercise9
```

the result is:

```
Twas brilliant, and the shining cove
Did glare and glimmer in the wave;
All mimsy were the borogoves,
And the mome raths outgrabe.
```

Like the **i** and **a** commands, the **c** command requires all added lines but the last to end with '\'.

Include Lines From a File

When you edit a file, you may wish to include, or *read in*, a second file as part of it. This is done with **r** command. To see how this works, type the following file into your computer, and call it **include**:

```
cat >include
       Then there comes the often-used refrain
       Whose repetitious writing dulls the brain.
<ctrl-D>
```

Now, prepare the file to be edited:

```
cat >exercise10
To write a poem doesn't take much time;
Just string some words to rhythm and a rhyme.
What poets do to language is a crime,
Words and syntax twisted for a rhyme.
<ctrl-D>
```

When you write your script, you must tell **sed** where to read in **include**. The form of the command should be familiar by now:

```
cat >script10
/rhyme/r include
<ctrl-D>
```

The result is of

```
sed -f script10 exercise10
```

is:

```
To write a poem doesn't take much time;
Just string some words to rhythm and a rhyme.
       Then there comes the often-used refrain
       Whose repetitious writing dulls the brain.
What poets do to language is a crime,
Words and syntax twisting for a rhyme.
       Then there comes the often-used refrain
       Whose repetitious writing dulls the brain.
```

Note that the **r** command inserted **include** *after* the addressed line. You can address lines by number, of course, as well as by pattern.

Quit Processing

The **q** command makes **sed** stop processing the text file. You will use this command most often to limit the application your **sed** script to a portion of your text file. For example, if you were editing a large file and you knew that your commands would be irrelevant to the last half of the file, you could insert an appropriately addressed **q** and save some computer time. You can also use this command to print portions of a file.

To see how this is done, prepare the following text file:

```
cat >exercise11
An hourglass has a very wide top,
       a very narrow
           middle
        and a bottom
that is also extremely wide.
<ctrl-D>
```

The following script will print the top of the text file. Note how the script uses **middle** to address the line where the file is to be split.

```
cat >script11
/middle/q
<ctrl-D>
```

The command:

```
sed -f script11 exercise11
```

produces:

```
An hourglass has a very wide top,
       a very narrow
           middle
```

To print out only the lines *after* the pattern **middle**, simply delete the first half of the file with the **d** command, as follows:

```
cat >script11a
1,/middle/d
<ctrl-D>
```

The result is the output:

```
        and a bottom
that is also extremely wide.
```

Next Line

The **n** command advances to the next line of the text file. The **n** command is useful for instances when you have two or more interrelated lines, and you want to ensure th a particular set of patterns is matched over the entire set of lines. To see how **n** works, prepare the following text file:

```
cat >exercise12
Alpha
One
Beta
Two
Gamma
Three
Delta
Four
Epsilon
Five
<ctrl-D>
```

To print a list of letters alone, type the following script:

```
cat >script12
n
d
<ctrl-D>
```

and execute it with the following command line:

```
sed -f script12 exercise12
```

The result will be the following:

```
Alpha
Beta
Gamma
Delta
Epsilon
```

Remember that **n** does *not* stop processing, go to the next text line, and begin processing all over again. Rather, it simply reads the next input line and continues processing from where it left off. For example, if your **sed** file consisted of three commands, the second of which was the **n** command, **sed** would apply the first command to the first line it read, then jump to the second line and apply the last commands. Then, it would read the third line and begin the pattern over again. To see how this works, prepare the following text file:

```
cat >exercise13
Alpha
Alpha
Alpha
Alpha
<ctrl-D>
```

Now type in this script:

```
cat >script13
s/Alpha/Apple/
/Apple/n
s/Alpha/Banana/
<ctrl-D>
```

When you execute the script with this command line:

```
sed -f script13 script13
```

the following will appear on your terminal:

```
Apple
Banana
Apple
Banana
```

Note that the first substitution command changed the first **Alpha** to **Apple**; the **n** command moved **sed** to the next line; and the second **s** command changed that **Alpha** to **Banana**.

Advanced sed Commands

The following sections discuss **sed**'s advanced features. They also discuss the method of operation.

Work Area

As described earlier, **sed** reads your text file one line at a time, and applies all of your editing commands to that line. After the editing commands have been applied, the edited line is either sent to the *standard output*, written to a file you have named, or thrown away, depending on what you have told **sed** to do.

When **sed** reads a line from your text file, it copies that line into a *work area*, where it actually executes your editing commands. **sed** notes the number of the line in the work area, then executes each editing command in turn, first checking to see if the patterns or line numbers specified in each command actually apply to that line. After each command is checked in turn and performed if indicated, **sed** prints the edited line (if it is supposed to be), and reads the next text line.

Add to Work Area

The exercises so far have used only one line in the work area. The **N** command, however, tells **sed** to read a second line into the work area. The following exercise illustrates the use of the work area and the **N** command.

```
cat >exercise14
This exercise has a brok
en word.
<ctrl-D>
```

Now, prepare the following **sed** script:

```
cat >script14
/brok$/N
s/brok\nen/broken/
s/has/had/
<ctrl-D>
```

and execute it with the following command line:

```
sed -f script14 exercise14
```

which produces the following text:

```
This exercise had a broken word.
```

You will find it helpful to review this exercise in some detail. The first command in the script

```
/brok$/N
```

tells **sed** to search for the pattern **brok** at the *end* of the line of text. (The dollar sign '$' in this instance indicates the end of the line; remember that when the '$' is used with a line number, it indicates the end of the *file*.) The **N** command tells **sed** to keep this line in the working space, and copy the *next* line into the working space as well.

When **sed** executes this command on the present text file, the work area will look like this:

```
This example has a brok<newline>en word.
```

Note that the two lines now appear to **sed** as though they formed one long line. The word **<newline>** represents the end of line character that tells your terminal or printer to jump to a new line when the text file is printed out. This character is invisible, but it is there, and it can be changed or deleted. You can describe this character to **sed** by using the characters **\n**. The first substitution in this script

```
s/brok\nen/broken/
```

replaces **brok<newline>en** with **broken**. Because the newline character is deleted from the text, what used to be printed out as two lines on your screen will now be printed out as one.

Note the difference, too, between the **n** and **N** commands. The **n** command will *replace* the text line in the work area with the next line from your text file. The **N** command, however, *appends* the next line from your text file to the end of the text already in the working area. The next exercise demonstrates this difference. First, create the following text file:

```
cat >exercise15
Apple
Apple
Apple
Apple
<ctrl-D>
```

Now, prepare the following two scripts:

```
cat >script15
/Apple/n
s/Apple/Banana/g
<ctrl-D>

cat >script15a
/Apple/N
s/Apple/Banana/g
<ctrl-D>
```

When script15 is executed with the following command line:

```
sed -f script15 exercise15
```

this will appear on your screen:

```
Apple
Banana
Apple
Banana
```

The **n** command told **sed** to print out the line already in the work area before reading in the next line from the text file. This meant that **sed** substituted **Banana** for **Apple** only on the *second* line of each pair.

Note what happens, however, when you run script15a, using this command line:

```
sed -f script15a exercise15
```

This text appears:

```
Banana
Banana
Banana
Banana
```

Because *both* lines of each pair were kept in the work area, the substitution command changed both of them.

Print First Line

The **P** command prints material from the work area. Unlike the **p** command, which prints *everything* in the work area, **P** prints only the *first* line in the work area. To see how this works, prepare the following text file:

```
cat >exercise16
Student:  George
Teacher:  Mr. Starzynski
Student:  Marian
Teacher:  Miss Peterson
Student:  Ivan
Teacher:  Mr. Starzynski
<ctrl-D>
```

Now, prepare the following scripts:

```
cat >script16
/Student/N
/Mr. Starzynski/p
<ctrl-D>

cat >script16a
/Student/N
/Mr. Starzynski/P
<ctrl-D>
```

When the first of these scripts is executed with the following command line (note the use of the **-n** option):

TUTORIALS

```
sed -n -f script16 exercise16
```

the result is

```
Student:  George
Teacher:  Mr. Starzynski
Student:  Ivan
Teacher:  Mr. Starzynski
```

whereas script16a, when executed as follows:

```
sed -n -f script16a exercise16
```

produces

```
Student:  George
Student:  Ivan
```

In **script16**, the **N** command lines pull both the name of the student and the name of the teacher into **sed**'s work area; the **p** command prints the student and teacher in each case where the teacher is Mr. Starzynski. In **script16a**, however, the **N** pulled both student and teacher into the work area, the **P** command printed only the *first* line of the work area — that is, the name of the student.

As you can see, **P** is a powerful tool that will allow you to select material from tables, lists, and other repetitive files.

Save Work Area

sed can create a second work area in addition to the primary work area in which **sed** performs its editing. **sed** does not execute any editing commands on the material stored in this secondary work area; rather, this work area can be used to store material that you want to edit or insert later.

The commands **h** and **H** copy material from the primary work area into the secondary work area. **h** and **H** differ in that **h** *displaces* any material in the secondary work area with the line being copied there, whereas **H** *appends* the line being copied onto the material already in the secondary work area. Note, too, that both **h** and **H** merely *copy* the primary work area into the secondary work area — after these commands have been executed, the material in the primary work area remains intact, and can be edited further, printed out, or deleted, whichever you prefer.

The commands **g** and **G** copy material back from the secondary work area into the primary work area. Again, these commands differ in that **g** *displaces* whatever is in the primary work area with the material from the secondary work area, whereas **G** *appends* the material from the secondary work area onto the material already in the primary work area.

The following exercises will demonstrate how these commands are used. First, create the following text file:

```
cat >exercise17
fruit:  apple
berry:  gooseberry
fruit:  orange
berry:  raspberry
fruit:  pear
berry:  blueberry
<ctrl-D>
```

The first script uses the **h** and **g** commands:

```
cat >script17
/fruit/h
/fruit/d
/berry/g
<ctrl-D>
```

When you execute this script with the following command line:

```
sed -f script17 exercise17
```

you receive the following text on your screen:

```
fruit:  apple
fruit:  orange
fruit:  pear
```

Review the last script in detail. The first command, **/fruit/h**, copied the line beginning with "fruit" into the

secondary work area, displacing whatever happened to be there. The command **/fruit/d** then deleted the line from the primary work area; if this were not done, it would then have been printed out. The third command, **/berry/g** then recopied the material from the secondary work area into the primary work area, displacing all lines in the primary work area that begin with "berry". The result of all this shuffling and displacing was that the three lines that begin with **fruit** were printed out.

The next script demonstrates the **H** command:

```
cat >script17a
/fruit/H
/fruit/d
/berry/g
<ctrl-D>
```

When you execute this script with the following command line:

```
sed -f script17a exercise 17
```

you see:

```
fruit:  apple
fruit:  apple
fruit:  orange
fruit:  apple
fruit:  orange
fruit:  pear
```

Because the **H** command *appends* material into the secondary work area, rather than replacing it as **h** does, all three lines that began with **fruit** were cumulatively stored in the secondary work area. Because the **g** command was used for every line that began with **berry**, the contents of the secondary work area (that is, the **fruit** lines) were written over each of the three lines that began with **berry**.

The next script demonstrates the use of the **G** command:

```
cat >script17b
/fruit/H
/fruit/d
/berry/G
s/berry://g
s/fruit://g
<ctrl-D>
```

When you execute this script with the following command line:

```
sed -f script17b exercise17
```

you will see:

```
gooseberry
apple
raspberry
apple
orange
blueberry
apple
orange
pear
```

The **H** command copies the lines that begin with **fruit** into the secondary work area. The **G** command then re-copies them from the secondary work area into the primary work area, and appends them to the material already in the primary work area — that is, to a line that begins with **berry**.

The two substitution commands then strip off the **fruit** and **berry** prefixes; obviously, these substitutions do not affect the operation of the **H** and **G** commands, but they do create a tidier result.

By the way, be sure you distinguish the **g** command from the **g** option used with the **s** command. If you do not, what **sed** finally prints out for you may appear very strange.

The final command that uses the secondary work area is **x**, which exchanges the two work areas. The following script shows how this is used:

```
cat >script17c
/fruit/H
/fruit/d
/blueberry/x
s/berry://g
s/fruit://g
<ctrl-D>
```

When you execute this script with the following command line:

```
sed -f script17c exercise17
```

you see:

```
gooseberry
raspberry
apple
orange
pear
```

The text lines that began with **fruit** were moved into the secondary working area. The **x** command was executed when the line that contained the word **blueberry** was reached, and the two working areas exchanged their contents. The **fruit** lines were then printed out, while the **blueberry** line was simply left in the secondary working at the end of the program, and disappeared when the program concluded.

Note that **x** simply swaps the two working areas — there is no "**X**" command that appends the work areas onto each other.

Transform Characters

The **y** command is a special form of the **s** command. With the **y** command, you can replace a number of characters easily, without having to write a series of **s** commands.

The form of the command is:

```
y/123/abc/
```

In the above example, **1** will be replaced with **a**, **2** with **b**, and **3** with **c** throughout the document (no **g** option is needed). For **y** to work properly there must be a one-to-one relationship between the characters being replaced and the characters replacing them. Also, **y** cannot make exchanges that involve more than one character — it cannot, for example, replace **apple** with **banana**.

One useful task for the **y** command is to change all upper-case letters in a file to lower case. Prepare the following text file to see how this is done:

```
cat >exercise18
NOW IS THE TIME FOR ALL GOOD MEN TO COME
TO THE AID OF THE PARTY.
<ctrl-D>
```

And prepare the following script, which will change these capitals:

```
cat >script18
y/ABCDEFGHI/abcdefghi/
y/JKLMNOPQR/jlkmnopqr/
y/STUVWXYZ/stuvwxyz/
<ctrl-D>
```

The alphabet is entered here in three chunks, to prevent the command from being too long to type easily. Execute this script with the following command line:

```
sed -f script18 exercise18
```

The result is:

```
now is the time for all good men to come
to the aid of the party.
```

Command Control

sed gives you advanced control over the execution of commands. The next subsections describe how these command controls help you write compact, powerful scripts.

{ }: Command Grouping

In several of the exercises presented so far, more than one command specified the same line locator. By using braces '{' and '}', you can bundle commands, which makes writing your scripts easier and lessens the chance of making a typographical error.

To see how this is done, type the following exercise:

```
cat >exercise19
When my love swears that she is made of truth,
I do believe her, though I know she lies,
That she might think me some untutored youth,
Unlearned in the world's false subtleties.
<ctrl-D>
```

Now, prepare the following script:

```
cat >script19
/truth/{N
P
}
/lies/d
<ctrl-D>
```

When you execute this script with the following command line:

```
sed -f script19 exercise19
```

the result on your terminal is:

```
When my love swears that she is made of truth,
That she might think me some untutored youth,
Unlearned in the world's false subtleties.
```

Note the syntax of this command. Each subsequent command must go on a line of its own, as must the right brace '}'. If this syntax is not observed, you will receive an error message.

!: All But

The **!** flag inverts a line selector; that is to say, the command will be performed on every line *but* the one named in the line selector. The following script will show how this works:

```
cat >script19a
2!d
<ctrl-D>
```

which, when run with the following command line:

```
sed -f script19a exercise19
```

produces

```
I do believe her, though I know she lies,
```

This script deleted every line *except* line 2. The **!** flag may also be used with a range of lines, as indicated by line numbers or line patterns; in either case, you must place the **!** flag *after* the line selectors and immediately *before* the command. Obviously, the **!** flag is very powerful, and can be used to sift out a few desired lines from a large file.

= : Print Line Number

You may wish to print only the *line number* of lines that contain a selected pattern. This is done with the **=** command. For example, you may wish to know the number of each line in the exercise that contains the word **she**. The following script:

```
cat >script19b
/she/=
<ctrl-D>
```

when executed with the following command line (note the **-n** option):

```
sed -n -f script19b exercise19
```

produces this result:

```
1
2
3
```

These numbers can be stored in a file and used in further editing, or included with the text of the fully edited file to provide a series of line markers.

Skipping Commands

sed normally processes editing commands in order, beginning with the first command and proceeding sequentially to the last. This behavior can be modified by the branching commands: **b**, **t**, and **:**.

These commands must be used with the colon (**:**) command, which defines a *label* point in the list of commands.

The **branch** command **b** allows you to skip unconditionally some editing commands in your script. The following exercise demonstrates how this can be used:

```
cat >exercise20
They went to sea in a sieve, they did;
In a sieve they went to sea;
In spite of all their friends could say,
On a winter's morn, on a stormy day,
In a sieve they went to sea.
<ctrl-D>
```

The following script uses the **b** command to avoid making certain changes to the first line of the poem:

```
cat >script20
s/sea/drink/g
/They/bend
s/sieve/ship/g
:end
```

When you execute this script with the following command line:

```
sed -f script20 exercise20
```

you will see:

```
They went to drink in a sieve, they did;
In a ship they went to drink;
In spite of all their friends could say,
On a winter's morn, on a stormy day,
In a ship they went to drink.
```

Note that the word **sea** is changed to **drink** throughout the file; however, when **sed** noted that the word **They** appeared in line 1, the **b** command forced it to seek the **:** command that was labeled with the word **end**, and to continue editing only *after* it found the labelled **:** command. In so doing, **sed** skipped the command to substitute **ship** for **sieve**, which is why that substitution was not made in line 1.

Note the syntax of the **b** command: the label follows it without a break. The text of the label is unimportant, just so long as it matches that used in the **b** command; however, the use of a label allows you to place several **b** or (as will be seen) **t** commands in the same script without mixing them up.

t: Test Command

The **test** command, **t**, also allows you to change the order in which editing commands are executed. Unlike the **b** command, which simply examines a line for a given pattern, the **t** command *tests* to see if a particular substitution has been performed.

The following script demonstrates the use of the **t** command:

```
cat >script20a
s/They/they/g
tend
s/sieve/ship/
:end
s/sea/drink/g
<ctrl-D>
```

which, when executed with the following command line:

```
sed -f script20a exercise20
```

produces:

```
they went to drink in a sieve, they did;
In a ship they went to drink;
In spite of all their friends could say,
On a winter's morn, on a stormy day,
In a ship they went to drink.
```

Note that the **t** command checked to see that **they** was substituted for **They** before branching to the ':' command labeled with the word **end**.

Also note the syntax of the **t** command: Like the **b** command, the label immediately follows the command and is not separated by a space; unlike the **b** command, however, the **t** command appears on the line *below* the substitution command for which it is testing.

For More Information

The Lexicon entry for **sed** summarizes its command-line options and commands. The COHERENT line editor **ed** resembles **sed**, except that it works interactively instead of in a stream. For information on **ed**, see its tutorial or its entry in the Lexicon.

The C Language

C is a computer language invented by Dennis Ritchie and Ken Thompson at AT&T Bell Laboratories in the early 1970s. In the approximately 25 years since its creation, C has become one of the most popular computer languages in the world. C is powerful and flexible, and it is highly portable. It has been implemented on practically every computer, and under practically every operating system, in the world.

C is the "native language" of the COHERENT system. COHERENT is written in C, and it includes a powerful C compiler among its suite of language tools for your use. You do not need to know C to use COHERENT to great advantage; however, if you plan to program under COHERENT, you would be well advised to become at least passably acquainted with it.

This tutorial is an introduction to the COHERENT C compiler and to the C language itself. The first part of this section describes how to compile programs under COHERENT. The second part is a brief tutorial in the C language.

Compiling C Programs under COHERENT

A C compiler is a program that transforms files of C source code into machine code. Compilation is a complex process that involves several steps; however, COHERENT simplifies it with the command **cc**, which controls all the actions of the compiler.

Try the Compiler

Before we launch into a lengthy explanation of what **cc** is and what it does, you can get a feel for it by trying it with a simple example. To begin, type the following to create a simple C program:

```
cat >hello.c
main() {
        printf("Hello, world\n");
}
<ctrl-D>
```

This creates a simple C program called **hello.c**. Now, compile your program by typing the following command:

```
cc -V hello.c
```

If you typed the program correctly, **cc** prints something like the following on your screen:

```
cc0 D2B000000201 hello.c  0x418CB8
cc1 D2B000000201 0x418CB8 0x408CB4
cc2a D2B000000201 0x408CB4 0x418CB8
cc2b D2B000000201 0x418CB8 hello.o
/bin/ld -X -o hello /lib/crts0.o hello.o /lib/libc.a -Z hello.o
```

What each of these messages means will be described below. If you receive an error message, try re-typing the program, and then re-compile it. When compilation is successfully completed, you will now have an executable program called **hello**. To invoke it, type:

```
hello
```

It should print the following on your screen:

```
Hello, world
```

As you can see, **cc** makes it easy to transform a file of C code into an executable program.

Phases of Compilation

As you noticed, **cc** printed a number of messages on your screen as it compiled **hello.c**. The reason you saw the messages was that compilation was performed with the **-V** option to **cc**; this tells **cc** to print a verbose output that describes each of its actions. **cc** prints numerous messages because the COHERENT C compiler is not just one program, but a number of different programs that work together. Each program performs a *phase* of compilation. The following summarizes each phase:

cpp The C preprocessor. This processes any of the '#' directives, such as **#include** or **#ifdef**, and expands macros.

cc0 The parser. This phase parses programs. It translates the program into a parse-tree format, which is independent of both the language of the source code and the microprocessor for which code will be generated.

cc1 The code generator. This phase reads the parse tree generated by **cc0** and translates it into machine code. The code generation is table driven, with entries for each operator and addressing mode.

cc2a The optimizer generator. This phase optimizes the generated code.

cc2b The optimizer generator. This phase writes the object module.

cc3 COHERENT also includes a fifth phase, called **cc3**, which can be run after the object generator, **cc2**. **cc3** generates a file of assembly language instead of a relocatable object module. **cc3** allows you to examine the code generated by the compiler. You did not see this phase when you compiled **hello.c** because this phase is optional and you did not request it. If you want COHERENT to generate assembly language, use the **-S** option on the **cc** command line.

Unless you specify the **-S** option, **cc** creates an *object module* that is named after the source file being compiled. This module has the suffix **.o**. An object module is *not* executable; it contains only the code generated by compiling a C source file, plus information needed to link the module with other program modules and with the library functions.

As the final step in its execution, **cc** calls the linker **ld** to produce an executable program.

Renaming Executable Files

When **cc** compiles a source file, by default it names the executable program after the *first* source file named on the **cc** command line. If you wish, you can give the executable file a different name. Use the **-o** (output) option, followed by the desired name.

Floating-Point Numbers

Often, you will need to use floating-point numbers in your programs. If you are unsure what a floating-point number is, see the Lexicon entry for **float**.

The routines that print floating-point numbers are large, and most C programs do not need to print floating-point numbers; therefore, the code to perform floating-point arithmetic is not included in a program by default. You must ask **cc** to include these routines with your program by using the **-f** option to **cc**.

To see how this works, let's modify **hello.c** to use floating-point numbers. Edit **hello.c** by typing the following commands:

```
ed hello.c
2
c
        printf("Hello, world %f\n", 123.4);
.
w
q
```

Now, compile the program with the same command line as before:

```
cc -V hello.c
```

When compilation has finished, type **hello**. You'll see the following output:

```
You must compile with the -f flag
to include printf() floating point.
Hello, world
```

COHERENT is telling you that you are using a floating-point number but that you did not compile the program to include code to process floating-point numbers. Now, recompile the program using the **-f** option to **cc**:

```
cc -V -f hello.c
```

When compilation has finished, type **hello**. If you typed the program correctly, you will see the following:

```
Hello, world 123.400000
```

TUTORIALS

As you can see, **hello** is now displaying the floating-point number **123.4** for you. For detailed information on **printf()**, see its entry in the Lexicon; **printf()** is also discussed in the tutorial section below.

Compiling Multiple Source Files

Many programs are built from more than one file of C source code. For example, the program **factor**, which is provided with COHERENT, is built from the C source files **factor.c** and **atod.c**. To produce the executable program **factor**, both source files must be compiled; the linker **ld** then joins them to form an executable file.

To compile a program that uses more than one source file, type all of the source files onto the **cc** command line. For example, to compile **factor** you would type the following:

```
cc -o factor -f factor.c atod.c -lm
```

This command compiles both C source files to create the program **factor**.

In the above example, **cc** produces the non-executable object modules **factor.o** and **atod.o**, and then links them to produce the executable file **factor**.

The argument **-lm** tells **cc** to include routines from the mathematics library when the object modules are linked. This option must come *after* the names of all of the source files, or the program will not be linked correctly.

Linking Without Compiling

When you are writing a program that consists of several source files, you will need to compile the program, test it, and then change one or more of the source files. Rather than recompile all of the source files, you can save time by recompiling only the modified files and relinking the program.

For example, if you modify the **factor** program by changing the source file **factor.c**, you can recompile **factor.c** *and* relink the entire program with the following command:

```
cc -o factor -f factor.c atod.o -lm
```

This **cc** command refers to the C source file **factor.c** and the *object module* **atod.o**. **cc** recognizes that **atod.o** is an object module and simply passes it to the linker **ld** without re-compiling it. You will find this particularly useful when your programs consist of many source files and you need to compile only a few of them.

To simplify compiling, especially if you are developing systems that use many source modules, you should consider using the **make** utility that is included with COHERENT. For more information on **make**, see its entry in the Lexicon, or see the tutorial for **make** that appears later in this manual.

Compiling Without Linking

At times, you will need to compile a source file but not link the resulting object module to the other object modules. You will do this, for example, to compile a module that you wish to insert into a library. Use **cc**'s option **-c** to tell **cc** not to link the compiled program. This option is often used to create relocatable object modules that can be archived into a library for later use.

For example, if you wanted just to compile **factor.c** without linking it, you would type:

```
cc -c factor.c
```

To link the resulting object module with the object module **atod.o** and with the appropriate libraries, type the following command:

```
cc -o factor -f factor.o atod.o -lm
```

Assembly-Language Files

C makes most assembly language programming unnecessary. However, you may wish to write small parts of your programs in assembly language for greater speed or to access processor features that C cannot use directly. COHERENT includes an assembler, named **as**, which is described in detail in the Lexicon.

To compile a program that consists of the C source file **example.c** and the assembly-language source file **example.s**, simply use the **cc** command as usual:

```
cc -o example example1.c example2.s
```

cc recognizes that the suffix **.s** indicates an assembly-language source file, and assembles it with **as**; then it links both object modules to produce an executable file.

Changing the Size of the Stack

The *stack* is the segment of memory that holds function arguments, local variables, and function return addresses. COHERENT takes advantage of the 80386 microprocessor's ability to allocate stack dynamically.

Where To Go From Here

This discussion of the **cc** command is by no means complete, but it includes enough information for you to begin to compile your programs. The Lexicon's entry for **cc** gives all of the command-line options available with **cc**. The Lexicon also has entries for **cpp**, the compiler phases, and for the linker **ld**, and describes them at greater length. All error messages generated by **cc** and by the assembler **as** appear in the appendix to this manual.

The next section in this tutorial introduces the C programming language.

C for Beginners

This section briefly introduces the C programming language. It is in two parts. The first part describes what a programming language is, and gives the history of the C programming language. It also introduces some concepts basic to C, such as *structured programming*, *pointer*, and *operator*. The second part walks through a C programming session. It emphasizes how a C programmer deals with a real problem, and demonstrates some aspects of the language.

This chapter is not designed to teach you the entire C language. It introduces you to C, so you can read the rest of this manual with some understanding. We urge you to look up individual topics of C programming in the Lexicon, and especially to study the example programs given there.

Programming Languages and C

Before beginning with C, it is worthwhile to review how a microprocessor and a computer language work.

A *microprocessor* is the part of your computer that actually computes. Built into it is a group of *instructions*. Each instruction tells the microprocessor to perform a task; for example, one instruction adds two numbers together, another stores the result of an arithmetic operation in memory, and a third copies data from one point in memory to another.

Together, a microprocessor's instructions form its *instruction set*. The instruction set is, in effect, the microprocessor's "native language".

A microprocessor also contains areas of very fast storage, called *registers*. The registers are essential to arithmetic and data handling within the microprocessor. How many registers a microprocessor has, and how they are designed, help to determine how much memory the microprocessor can read and write, or *address*, and how the microprocessor handles data.

A *computer language*, as the name implies, lets a human being use the microprocessor's instruction set. The lowest level language is called "assembly language". In assembly language, the programmer calls instructions directly from the microcomputer's instruction set, and manipulates the registers within the microprocessor. To write programs in assembly language, a programmer must know both the microprocessor's instruction set and the configuration of its registers.

Assembly and High-Level Languages

With assembly language, the programmer can tailor the program specifically to the microprocessor. However, because each microprocessor has a unique instruction set and configuration of registers, a program written in one microprocessor's assembly language cannot be run on another microprocessor. For example, no program written in the assembly language for the Motorola 68000 microprocessor can be run on the IBM PC or any PC-compatible computer. The program must be entirely rewritten in the assembly language for the Intel microprocessor, which is difficult and time consuming.

A *high-level language* helps programmers to avoid these problems. The programmer does not need to know the microprocessor in detail; instead of specific microprocessor instructions, he writes a set of logical constructions. These constructions are then handed to another program, which translates them into the instructions and register calls used by a specific microprocessor. In theory, a program written in a high-level language can be run on any microprocessor for which someone has written a translation program.

A high-level language allows the programmer to concentrate on the task being executed, rather than on the details of registers and instructions. This means that programs can be written more quickly than in assembly language, and can be maintained more easily.

So, What Is C?

As noted earlier, C was invented at AT&T Bell Laboratories by Dennis Ritchie and Ken Thompson. They created C specifically to re-write the UNIX operating system from PDP-11 assembly language. Ritchie designed C to have the power, speed, and flexibility of assembly language, but the portability of high-level languages.

In 1978, Ritchie and Brian W. Kernighan published *The C Programming Language*, which described and defined the C language. In 1988, the American National Standards Institute (ANSI) published its standard for the C language. This standard has, on the whole, become the basis for current implementations of C.

Because C is modeled after assembly language, it has been called a "medium-level" language. The programmer doesn't have to worry about specific registers or specific instructions, but he can use all of the power of the computer almost as directly as he can with assembly language.

Because C was written by experienced programmers for experienced programmers, it makes little effort to protect a programmer from himself. A programmer can easily write a C program that is legal and compiles correctly but crashes when run. Also, C's punctuation marks, or "operators", closely resemble each other. Thus, a mistake in typing can create a legal program that compiles correctly but behaves very differently from what you expect.

Structured Programming

C is a *structured language*. This means that a C program is assembled from a number of sub-programs, or *functions*, each of which performs a discrete task. If this concept is difficult to grasp, consider the following example.

Suppose you want to turn a file of text into upper-case letters and print it on the screen. This job seems simple, but a program to do it must perform five tasks:

1. Read the name of the file to open.
2. Open the file so it can be read, in much the same way that you must open a book before you can read it.
3. Read the text from the file.
4. Turn what is read into upper-case letters.
5. Print the transformed text onto the screen.

A good program will also perform the following tasks:

1. Check that the file requested actually exists.
2. Check that the file requested is actually a text file rather than a file of binary information; the latter makes very little sense when printed on the screen.
3. Close the program neatly when the work is finished.
4. Stop processing and print an error message if a problem occurs.

A structured language like C allows you to write a separate function for each of these tasks.

A structured programming language offers two major advantages over a non-structured language. First, it is easier to debug a function than an entire program because the function can be unplugged from the program as a whole, made to work correctly, and then plugged back in again. Second, once a function works, it can be used again and again in different programs. This allows you to create a *library* of reliable functions that you can pull off the shelf whenever you need them.

The functions within a program communicate by passing values to each other. The value being passed can be an integer, a character, or — most commonly — an address within memory where a function can find data to manipulate. This passing of addresses, or *pointers*, is the most efficient way to manipulate data because by receiving one number, a function can find its way to a large amount of data. This speeds up a program's execution.

C adds some extra tools to help you construct programs. To begin, C allows you to store functions in compiled form. These precompiled functions are added only when the program is finally loaded into memory; this spares you the trouble of having to recompile the same code again and again. Second, C adds a preprocessor that expands definitions, or *macros*, and pulls in special material stored in *header files*. This allows you to store often-used definitions in one file and use them just by adding one line to your program.

Writing a C Program

As noted above, a C program consists of a bundle of sub-programs, or *functions*, which link together to perform the task you want done. Every C program must have one function that is called **main**. This is the main function; when the computer reads this, it knows that it must begin to execute the program. All other functions are subordinate to **main**. When the **main** function is finished, the program is over.

To see how these elements work, review the program **hello.c**, which you worked with earlier in this tutorial:

```
main()
{
        printf("Hello, world\n");
}
```

As you can see, this program begins with the word **main**. The program begins to work at this point. The parentheses after **main** enclose all of the *arguments* to **main** — or would, if this program's **main** took any. An argument is an item of information that a function uses in its work.

The braces '{' and '}' enclose all the material that is subsidiary to **main**.

The word "printf" *calls* a function called **printf()**. This function performs formatted printing. The line of characters (or "string") *Hello, world* is the argument to **printf()**: this argument is what **printf()** is to print.

The characters '\n' stand for a newline character. This character "tosses the carriage", or moves the cursor to a new line and returns it to the leftmost column on your screen. Using this character ensures that when printing is finished, the cursor is not left fixed in the middle of the screen. Finally, the semicolon ';' at the end of the command indicates that the function call is finished.

One point to remember is that **printf()** is *not* part of the C language. Rather, it is a *function* that was written by Mark Williams Company, then compiled and stored in a library for your use. This means that you do not have to re-invent a formatted printing function to perform this simple task: all you have to do is *call* the one that Mark Williams has written for you.

Although most C programs are more complicated than this example, every C program has the same elements: a function called **main()**, which marks where execution begins and ends; braces that fence off blocks of code; functions that are called from libraries; and data passed to functions in the form of arguments.

A Sample C Programming Session

This section walks you through a C programming session. It shows how you can go about planning and writing a program in C.

C allows you to be precise in your programming, which should make you a stronger programmer. Be careful, however, because C does exactly what you tell it to do, nothing more and nothing less. If you make a mistake, you can produce a legal C program that does very unexpected things.

Designing a Program

Most programmers prefer to work on a program that does something fun or useful. Therefore, we will write something useful: a version of the COHERENT utility **scat**, that we'll call **display**. It will do the following:

1. Open a text file on disk.
2. Display its contents in 23-line chunks (one full screen).
3. After displaying a chunk, wait to see if the user wants to see another chunk. If the user presses the **<return>** key alone, display another chunk; if the user types any other key before pressing the **<return>** key, exit.
4. Exit automatically when the end of file is reached.

As you can see, the first step in writing a program is to write down what the program is to do, in as much detail as you can manage, and preferably in complete sentences.

Now, invoke **ed** or MicroEMACS and get ready to type in the program:

```
ed display.c
```

or:

```
me display.c
```

We suggest that you use the MicroEMACS editor, because this tutorial will make numerous changes to the program as it progresses and it will be easier to see these changes in context if you use a screen editor rather than a line editor. The rest of this tutorial assumes that you are using MicroEMACS. If you are not familiar with MicroEMACS, it is briefly described in *Using the COHERENT System*. A tutorial for MicroEMACS also appears in this manual, or you may wish to see the entry for **me** in the Lexicon.

In the above commands, the suffix **.c** on the file name indicates that this is a file of C code. If you do not use this suffix, the **cc** command will not recognize that this is a file of C code and will refuse to compile it.

Begin by inserting a description of the program into the top of the file in the form of a *comment*. When a C compiler sees the symbol '/*', it throws away everything it reads until it sees the symbol '*/'. This lets you insert text into your program to explain what the program does.

Type the following:

```
/*
 * Truncated version of the 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, print another 23 lines.
 * If user types any other key, exit.
 * Exit when EOF is read.
 */
```

Save what you have typed by pressing **<ctrl-X>** and then **<ctrl-S>**. Now, anyone, including you, who looks at this program will know exactly what it is meant to do.

The main() Function

As described earlier, the C language permits *structured programming*. This means that you can break your program into a group of discrete functions, each of which performs one task. Each function can be perfected by itself, and then used again and again when you need to execute its task. C requires, however, that you signal which function is the *main()* function, the one that controls the operation of the other functions. Thus, each C program must have a function called **main()**.

Now, add **main()** to your program. Type the code that is shaded, below:

```
/*
 * Truncated version of the 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, print another 23 lines.
 * If user types any other key, exit.
 * Exit when EOF is read.
 */

main()
{
}
```

The parentheses "()" show that **main()** is a function. If **main()** were to take any arguments, they would be named between the parentheses. The braces "{}" delimit all code that is subordinate to **main()**; this will be explained in more detail below.

Note that the shortest legal C program is **main(){}**. This program doesn't do anything when you run it, but it will compile correctly and generate an executable file.

Now, try compiling the program. Save your text by typing **<ctrl-X><ctrl-S>**, and then exit from the editor by typing **<ctrl-X><ctrl-C>**. Compile the program by typing:

```
cc display.c
```

When compilation is finished, type **display**. The shell will pause briefly, then return the prompt to your screen. As you can see, you now have a legal, compilable C program, but one that does nothing.

Open a File and Show Text

The next step is to install routines that open a file and print its contents. For the moment, the program will read only a file called **display.c**, and not break it into 23-line portions.

Type the shaded lines into your program, as follows:

```
/*
 * Truncated version of the 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, print another 23 lines.
 * If user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>

main()
{
        char string[128];
        FILE *fileptr;

/* Open file */
        fileptr = fopen("display.c", "r");

/* Read material and display it */
        for (;;) {
                fgets(string, 128, fileptr);
                printf("%s", string);
        }
}
```

Note first how comments are inserted into the text, to guide the reader.

Now, note the lines

```
        char string[128];
        FILE *fileptr;
```

These *declare* two data structures. That is, they tell COHERENT to set aside a specific amount of memory for them.

The first declaration, **char string[128];**, declares an array of 128 **char**s. A **char** is a data entity that is exactly one byte long; this is enough space to store exactly one alphanumeric character in memory, hence its name. An *array* is a set of data elements that are recorded together in memory. In this instance, the declaration sets aside 128 **char**s-worth of memory. This declaration reserves space in memory to hold the data that your program reads.

The second declaration, **FILE *fileptr**, declares a *pointer* to a **FILE** structure. The asterisk shows that the data element points to something, rather than being the thing itself. When a variable is declared to be a pointer, the C compiler sets aside enough space in memory to hold an *address*. When your program reads that address, it then knows where the actual data are residing, and looks for them there. C uses pointers extensively, because it is much more efficient to pass the address of data than to pass the data themselves. You may find the concept of pointers to be a little difficult to grasp; however, as you gain experience with C, you will find that they become easy to use.

The **FILE** structure is the data entity that holds all the information your program needs to read information from or write information to a file on the disk. For a detailed discussion of the **FILE** structure, see its entry in the Lexicon. For now, all you need to remember is that this declaration sets aside a place to hold a pointer to such a structure, and the structure itself holds all of the information your program needs to manipulate a file on disk. In effect, the variable **fileptr** is used within your program as a synonym for the file itself.

Now, the line

```
        fileptr = fopen("display.c", "r");
```

opens the file to be read. The function **fopen()** *opens* the file, fills the **FILE** structure, and fills the variable **fileptr** with the address of where that structure resides in memory.

fopen() takes two arguments. The first is the name of the file to be opened, within quotation marks. The second argument indicates the *mode* in which to open the file; **r** indicates that the file will be read rather than written into.

The lines

```
for(;;)
{
```

begin a *loop*. A loop is a section of code that is executed repeatedly until a condition that you set is fulfilled. For example, you may define a loop that executes until the value of a particular variable becomes greater than zero.

for is built into the C language. Note that it has braces, just like **main()** does; these braces mean that the following lines, up to the next right brace (}) are part of this loop. You can set conditions that control how a **for** loop operates; in its present form, it will loop forever. This will be explained in more detail shortly.

Two library functions are executed within the loop. The first,

```
fgets(string, 128, fileptr);
```

reads a line from the file named in the **fileptr** variable, and writes it into the character array called **string**. The middle argument ensures that no more than 128 characters will be read at a time. The second line within this loop,

```
printf("%s", string);
```

prints the line. **printf()** is a powerful and subtle function; in its present form, it prints on the screen the string contained in the variable *string*.

Finally, the line at the top of the program:

```
#include <stdio.h>
```

tells the C preprocessor **cpp** to read the *header file* called **stdio.h**. The term "STDIO" stands for "standard input and output"; **stdio.h** declares and defines a number of routines that will be used to read data from a file and write them onto the screen.

When you have finished typing in this code, again compile the program as you did earlier. If an error occurs, check what you have typed and make sure that it *exactly* matches the code shown on the previous page. If you find any errors, fix them and then recompile. If errors persist, check it in the table of error messages that appear at the end of this tutorial.

When compilation is finished, execute **display** as you did earlier. You will see the text from **display.c** scroll across the screen. When the text is finished, however, the COHERENT prompt does not return; you have not yet inserted code that tells the program to recognize that the file is finished. Type **<ctrl-C>** to break the program and return to COHERENT.

Accepting File Names

Of course, you will want **display** to be able to display the contents of any file, not just files named **display.c**. The next step is to add code that lets you pass arguments to the program through its command line. This task requires that you give the **main()** function two arguments. By tradition, these are always called **argc** and **argv**. How they work will be described in a moment.

The enhanced program appears as follows. You should change or insert the lines that are shaded:

```
/*
 * Truncated version of the 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, print another 23 lines.
 * If user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>
#define MAXCHAR 128

main(argc, argv)
/* Declare arguments to main() */
```

```
int argc;
char *argv[];
{
        char string[MAXCHAR];
        FILE *fileptr;

/* Open file */
        fileptr = fopen(argv[1], "r");

/* Read material and display it */
        for (;;) {
                fgets(string, MAXCHAR, fileptr);
                printf("%s", string);
        }
}
```

First, a small change has been added: the line

```
        #define MAXCHAR 128
```

defines the *manifest constant* **MAXCHAR** to be equivalent to 128. This is done because the "magic number" 128 is used throughout the program. If you decide to change the number of characters that this program can handle at once, all you would have to do is to change this one line to alter the entire program. This cuts down on mistakes in altering and updating the program. If you look lower in the program, you will see that the declaration

```
        char string[128]
```

has been changed to read

```
        char string[MAXCHAR]
```

The two forms are equivalent; the only difference is that the latter is easier to use. It is a good idea to use manifest constants wherever possible, to streamline changes to your program.

Now, look at the line that declares **main()**. You will see that **main()** now has two arguments: **argc** and **argv**.

The first is an **int**, or integer, as shown by its declaration — **int argc;**. **argc** gives the *number* of entries typed on a command line. For example, when you typed

```
        display filename
```

the value of **argc** was set to two: one for the command name itself, and one for the file-name argument. **argc** and its value are set by the compiler. You do not have to do anything to ensure that this value is set correctly.

argv, on the other hand, is an array of pointers to the command line's arguments. In this instance, **argv[1]** points to name of the file that you want **display** to read. This, too, is set by COHERENT, and works automatically.

If you look below at the line that declares **fopen()**, you will see that **display.c** has been replaced with **argv[1]**; this means that you want **fopen()** to open the file named in the first argument to the **display** command.

Now, try running the program by typing

```
        display display.c
```

Be sure that you give the command only one file name as an argument, no more and no less. Code that checks against errors has not yet been inserted, and handing it the wrong number of arguments could cause problems for you.

display will open **display.c** and print its contents on the screen. You still need to type **<ctrl-C>** when printing is finished; the code to recognize the end of the file will be inserted later.

Error Checking

Obviously, the program runs at this stage, but is still fragile, and could cause problems. The next step is to stabilize the program by writing code to check for errors. To do so, a programmer must first write code to capture error conditions, and then write a routine to react appropriately to an error.

Our edited program now appears as follows:

```
/*
 * Truncated version of the 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, print another 23 lines.
 * If user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>
#define MAXCHAR 128

main(argc, argv)
/* define arguments to main() */
int argc;
char *argv[];
{
        char string[MAXCHAR];
        FILE *fileptr;

/* Check if right number of arguments was passed */
        if (argc != 2)
                error("Usage: display filename");

/* Open file */
        if ((fileptr = fopen(argv[1], "r")) == NULL)
                error("Cannot open file");

/* Read material and display it */
        for (;;) {
                fgets(string, MAXCHAR, fileptr);
                printf("%s", string);
        }
}

/* Process error messages */
error(message)
char *message;
{
        printf("%s", message);
        exit(1);
}
```

The additions to the program are introduced by comments.

The first addition

```
        if (argc != 2)
                error("Usage:  display filename");
```

checks to see if the correct number of arguments was passed on the command line; that is to say, it checks to make sure that you named a file when you typed the **display** command.

As noted above, **argc** is the number of arguments on the command line, or rather, the number of arguments plus one, because the command name itself is always considered to be an argument. The statement **if (argc != 2)** checks this. The **if** statement is built into C. If the condition defined between its parentheses is true, then do something, but if it is not true, do nothing at all. The operator **!=** means "does not equal". Therefore, our statement means that if **argc** is not equal to two (in other words, if there are not two and only two arguments to the **display** command — the command name itself plus a file name), execute the function **error**. **error** is defined below.

Our **fopen()** function also has some error checking added (which will be described in a moment):

```
if ((fileptr = fopen(argv[1], "r")) == NULL)
        error("Cannot open file");
```

fopen() returns a value called "NULL" if, for any reason, it cannot open the file you requested. Thus, our new **if** statement says that if **fopen()** cannot open the file named on the command line (that is, **argv[1]**), it should invoke the **error()** function.

C always executes nested functions from the "inside out". That means that the innermost function (that is, the function that is enclosed most deeply within the pairs of parentheses) is executed first. Its result, or what it *returns*, is then passed to next outermost function as an argument; that function is then executed and what it returns is, in turn, passed to the function that encloses it, and so on. In this instance, the innermost function is

```
fileptr = fopen(argv[1], "r")
```

fopen() is executed and what it returns is written into **fileptr**. What **fopen** returned is then passed to the next outer operation; in this case, it is compared with NULL, as follows:

```
(fileptr = fopen(argv[1], "r")) == NULL)
```

What that operation returns is then passed to the outermost function, in this case the **if** statement, which evaluates what it is passed, and acts accordingly. If **fileptr** is NULL (that is, if **fopen** couldn't open the file), the **if** statement will be true and the **error** function called. If, however, the file was opened, **fileptr** will not equal NULL and the program will proceed.

As this example shows, C allows a programmer to nest functions quite deeply. Although nested functions are sometimes difficult to untangle when you read them, they make programming much more convenient.

Finally, at the bottom of the file is a new function, called **error()**:

```
error(message)
char *message;
{
        printf("%s", message);
        exit(1);
}
```

This function stands outside of **main()**, as you can tell because it appears outside of **main()**'s closing brace. This function is called only when your program needs it. If there are no errors, the program progresses only until the closing brace in **main** and the **error** function is never called.

error() takes one argument, the message that is to be printed on the screen. This message is defined by the routine that calls **error()**. **error()** uses the function **printf()** to print the message, then calls the **exit()** function; this, as its name implies, causes the program to stop. The argument **1** is a special signal that tells COHERENT that something went wrong with your program.

When the error checking code is inserted, recompile the program and execute it without an argument. Previously, this would cause the program to crash; now, all it does is print the message

```
Usage: display filename
```

and terminate the program.

Print a Portion of a File

So far, our utility just opens a file and streams its contents over the screen. Now, you must insert code to print a 23-line portion of the file. At present, it will only print the first 23 lines, and then exit.

To do so, you must insert another **for** loop. Unlike our first loop, which ran forever, this one will cycle only 23 times, and then stop. Our updated program appears as follows:

```
/*
 * Truncated version of the 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, print another 23 lines.
 * If user types any other key, exit.
 * Exit when EOF is read.
 */
```

```
#include <stdio.h>
#define MAXCHAR 128

main(argc, argv)
int argc;
char *argv[];
{
        char string[MAXCHAR];
        FILE *fileptr;
        int ctr;

/* Check if right number of arguments was passed */
        if (argc != 2)
                error("Usage: display filename");

/* Open file */
        if ((fileptr = fopen(argv[1], "r")) == NULL)
                error("Cannot open file");

/* Output 23 lines */
        for (;;) {
                for (ctr = 0; ctr < 23; ctr++) {
                        fgets(string, MAXCHAR, fileptr);
                        printf("%s", string);
                }
                exit(0);
        }
}

/* Process error messages */
error(message)
char *message;
{
        printf("%s", message);
        exit(1);
}
```

The new **for** loop is nested inside the loop governed by **for(;;)**. The program also declares a new variable, **ctr**, at the beginning of the program. **ctr** keeps track of how many times the loop has executed. Now, look at the line:

```
        for (ctr = 0; ctr < 23; ctr++)
```

It has three sub-statements, which are separated by semicolons. The first sub-statement sets **ctr** to zero; the second says that execution is to continue as long as **ctr** is less than 23; and the third says that **ctr** is to be increased by one every time the loop executes (this is indicated by the **++** appended to **ctr**). With each iteration of this loop, **fgets()** reads a line from the file named on the **display** command line, and **printf()** prints it on the screen.

Also, an **exit()** call has been set after this new loop. This ensures that the program will exit automatically after the loop has finished executing. This is a temporary measure, to make sure that you no longer have to type **<ctrl-C>** to return to the shell.

When you have updated the program, recompile it in the usual way. When you run it with an appropriate file of an appropriate length, e.g., **display.c** itself, **display** will show the first 23 lines of the file, and then the shell's prompt will return.

The program is now approaching its final form.

Checking for the End of File

The next-to-last step in preparing the program is teaching it to recognize the end of a file when it sees it. This does not appear to be needed now because the program exits automatically after 23 lines or fewer, but it will be quite necessary when the program begins to display more than one 23-line portion of text.

The function **fgets()** checks to see if it has arrived at the end of a file, and returns a special value if it has. **fgets()** normally returns the address of the string into which it writes its output; however, if it runs into the end of a file (or if any other error occurs), it returns the special value NULL. By reading the value of what **fgets()** returns, **display** can detect if the end of the file has been encountered, and stop reading. To do so, the **fgets()** statement must be set within an **if** statement. The **if** statement will capture what **fgets()** returns, and continue execution as long as the value of the number returned is not NULL.

The updated program now appears as follows:

```
/*
 * Truncated version of the 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, print another 23 lines.
 * If user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>
#define MAXCHAR 128

main(argc, argv)
int argc;
char *argv[];
{
        char string[MAXCHAR];
        FILE *fileptr;
        int ctr;

/* Check if right number of arguments was passed */
        if (argc != 2)
                error("Usage: display filename");

/* Open file */
        if ((fileptr = fopen(argv[1], "r")) == NULL)
                error("Cannot open file");

/* Output 23 lines, while checking for EOF */
        for (;;) {
                for (ctr = 0; ctr < 23; ctr++) {
                        if (fgets(string, MAXCHAR, fileptr) != NULL)
                                printf("%s", string);
                        else
                                exit(0);
                }
                exit(0);
        }
}

/* Process error messages */
error(message)
char *message;
{
        printf("%s", message);
        exit(1);
}
```

First, note that the comment that describes the program's output has been changed to reflect our changes to the program. It is important for a programmer to ensure that the comments and the code are in step with each other.

Our new **if** statement

```
if (fgets(string, MAXCHAR, fileptr) != NULL)
```

checks what **fgets()** returns: if it does not return NULL, the end of the file has not been reached, the **if** statement is true and the program prints out the next line. (The operator **!=** indicates "not equal".) If it returns NULL, however, the end of file has been reached, the **if** statement is false so the **else** statement is executed, which causes **display** to exit.

Note, too, that a new control statement is introduced: **else**. This, like **if**, is built into the C language. An **else** statement is always paired with an **if** statement; together, they mean that if the condition for which **if** is testing is true, the program should do one thing; otherwise, it should do something else. In this case, the program says that if the end of file has not been reached, another line has been read from the file and should be printed on the screen; however, if it has been reached, then the program should exit. As you can imagine, **if/else** pairs are common in C programming; they are logical and useful.

One more task must be done on our program; then it is finished.

Polling the Keyboard

For the program to be complete, it has to ask you if you want to see another 23-line portion of text whenever the argument contains more than 23 lines. The program should write another portion if you press the **<return>** key alone; if you type any other key before you press **<return>**, then it should exit.

To do so, we will print a query on the screen, then read what the user has typed and interpret it. When these changes are inserted, the program is complete:

```
/*
 * Truncated version of the 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, print another 23 lines.
 * If user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>
#define MAXCHAR 128

main(argc, argv)
int argc;
char *argv[];
{
        char string[MAXCHAR];
        FILE *fileptr;
        int ctr;

/* Check if right number of arguments was passed */
        if (argc != 2)
                error("Usage: display filename");

/* Open file */
        if ((fileptr = fopen(argv[1], "r")) == NULL)
                error("Cannot open file");

/* Output 23 lines, while checking for EOF */
        for (;;) {
                for (ctr = 0; ctr < 23; ctr++) {
                        if (fgets(string, MAXCHAR, fileptr) != NULL)
                                printf("%s", string);
                        else
                                exit(0);
```

```
                }
/* Query if user wishes to continue */
                printf("Continue? ");
                fflush(stdout);
                fgets(string, MAXCHAR, stdin);

                if (string[0] != '\n')
                        exit(0);
        }
}

/* Process error messages */
error(message)
char *message;
{
        printf("%s", message);
        exit(1);
}
```

These new lines introduce a few new twists. The lines

```
        printf("Continue? ");
        fflush(stdout);
```

print the prompt **Continue?** on the screen. Note that no '\n' appears after the prompt; this ensures that the cursor does *not* jump to the next line, but stays next to the prompt. Because no '\n' appears after the line, however, you have to force it to appear on the screen; this is accomplished with the statement:

```
        fflush(stdout);
```

fflush() flushes matter to an output device. **stdout** points to a file stream, just like the stream that you opened with the call to **fopen()**, earlier in the program. **stdout** is opened in the header file **stdio.h**, which was read at the beginning of the program; it always points to the user's screen.

The next line reads the user's keyboard:

```
        fgets(string, MAXCHAR, stdin);
```

This version of **fgets** reads matter into our array **string**; however, instead of reading the file pointed to by **fileptr**, it reads what is pointed to by **stdin**. **stdin** is a stream that is also defined in **stdio.h**; it always points to the user's keyboard.

Finally, the statement

```
        if (string[0] != '\n')
```

checks what the user typed by reading the first (that is, the zero-th) character written in the array **string** by the preceding call to **fgets()**. (Note that with C, counting always begins with zero rather than one.) If the user just types **<return>**, then **string[0]** will hold '\n'; and the **if** statement will *not* be true, the program jumps to the preceding **for** statement, and more text is written to the screen. However, if the user types anything before typing **<return>**, the **if** statement will succeed and the program will exit. This may seem a little convoluted, but it actually is a straightforward and efficient way to receive information from the user.

After you have inserted these changes, again compile the program.

When compilation is finished, try typing

```
        display display.c
```

The first 23 lines of the source code to the program now appear on your screen. Hit **<return>**; the next 23 lines appear. Now, type any other key, and then press **<return>**: the program exits.

You now have a simple but helpful **display** utility.

For More Information

This section has given you a brief, concentrated introduction to writing a C program. If you are new to programming, much of what happened must seem strange, but we hope it helped you to appreciate the logic of how C works.

Numerous books are on the market to teach beginners how to program in C; the following section gives a small bibliography of books on C. Also, look at the sample C programs in the Lexicon. These demonstrate how to use many of the functions available to you with COHERENT.

Bibliography

The following books may be helpful in developing your skills with C. This list also contains all books that are referenced in this manual. It is by no means exhaustive; however, it should prove helpful to both beginners and experienced programmers.

American National Standards Institute: *Draft Programming Language C (October 1986 Draft)*. Washington, D.C.: X3 Secretariat, Computer and Business Equipment Manufacturers Association, 1986.

AT&T Bell Laboratories: *The C Programmer's Handbook*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1985.

Bentley, J.: *Programming Pearls*. Reading, Mass.: Addison-Wesley Publishing Company, 1986. *Not, strictly speaking, about C — but belongs on every programmer's bookshelf.*

Brooks, F.P., Jr.: *The Mythical Man-Month: Essays on Software Engineering.* Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1975. *Not about programming, but should be read by every programmer.*

Chirlin, P.M.: *Introduction to C*. Beaverton, Or.: Matrix Publishers, Inc., 1984.

Derman, B. (ed.): *Applied C*. New York: Van Nostrand Reinhold Co., Inc., 1986.

Feuer, A.R.: *The C Puzzle Book*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1982.

Gehani, G.: *Advanced C: Food for the Educated Palate*. Rockville, Md.: Computer Science Press, 1985.

Hancock, L.: Krieger, M.: *The C Primer*. New York: McGraw-Hill Book Publishers, Inc., 1982.

Harbison, S.; Steele, G.: *C: A Reference Manual*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

Haviland, K.F., Salama, B.: *UNIX System Programming.* Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1987.

Hogan, T.: *The C Programmer's Handbook*. Bowie, Md.: Brady Publishing, 1984.

Kelley, A.; Pohl, I.: *C by Dissection: The Essentials of C Programming*. Menlo Park, Ca.: The Benjamin/Cummings Publishing Company, Inc., 1987.

Kernighan, B.W.; Ritchie, D.M.: *The C Programming Language*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1978.

Kernighan, B.W.; Plauger, P.J.: *The Elements of Programming Style*, ed. 2. New York: McGraw-Hill Book Co., 1978.

Kochan, S.G.: *Programming in C*. Hasbrouck Heights, N.J.: Hayden Book Co., Inc., 1983.

Knuth, D.E.: *The Art of Computer Programming*, vol. 1: *Basic Algorithms.* Reading, Ma.: Addison-Wesley Publishing Co., 1969.

Knuth, D.E.: *The Art of Computer Programming*, vol. 2: *Seminumerical Algorithms.* Reading, Ma.: Addison-Wesley Publishing Co., 1969.

Knuth, D.E.: *The Art of Computer Programming*, vol. 3: *Sorting and Searching.* Reading, Ma.: Addison-Wesley Publishing Co., 1969.

Lapin, J.E.: *Portable C and UNIX System Programming.* Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1987.

Mark Williams Company: *ANSI C: A Lexical Guide.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

Plum, T.: *C Programming Guidelines*. Cardiff, N.J.: Plum Hall, Inc., 1984.

Plum, T.; Brodie, J.: *Efficient C*. Cardiff, NJ: Plum Hall, Inc., 1985.

Purdum, J.: *C Programming Guide*. Indianapolis: Que Corp., 1983.

Purdum, J.; Leslie, T.C.; Stegemoller, A.L.: *C Programmer's Library*. Indianapolis: Que Corp., 1984.

Rochkind, M.J.: *Advanced UNIX Programming.* Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1985.

Traister, R.J.: *Going from BASIC to C*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1984.

Traister, R.J.: *Mastering C Pointers*. New York: Academic Press, Inc., 1990.

Traister, R.J.: *Programming in C for the Microprocessor User*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1984.

Vile, R.C., Jr.: *Programming in C with Let's C*. Glenview, IL: Scott, Foresman and Company, 1988.

Waite, M.; Prata, S.; Martin, D.: *C Primer Plus*. Indianapolis: Howard W. Sams, Inc., 1984.

Weber Systems, Inc.: *C Language User's Handbook*. New York: Ballantine Books, 1984.

Zahn, C.T.: *C Notes*. New York: Yourdan Press, 1979.

Introduction to the awk Language

awk is a general-purpose language for processing text. With **awk**, you can manipulate strings, process records, and generate reports.

awk is named after its creators: A. V. Aho, P. J. Weinberger, and Brian W. Kernighan. Unfortunately, its name suggests that **awk** is awkward — whereas in truth the **awk** language is simple, elegant, and powerful. With it, you can perform many tasks that would otherwise require hours of drudgery.

awk uses a simple syntax. Each statement in an **awk** program contains either or both of two elements: a *pattern* and an *action*. The pattern tells **awk** what lines to select from the input stream; and the action tells **awk** what to do with the selected data.

This tutorial explains how to write **awk** programs. It explains how to describe a pattern to **awk**. It also describes the range of actions that **awk** can perform; these include formatted printing of text, assigning variables, defining arrays, and controlling the flow of data.

Example Files

Before you begin to study **awk**, please take the time to type the following text files that are used by the examples in this tutorial.

The first is some text from Shakespeare. Use the command **cat** to type it into the file **text1**, as follows. Note that **<ctrl-D>** means that you should hold down the **Ctrl** (or **control**) key and simultaneously press 'D'. Do not type it literally.

```
cat > text1
When, in disgrace with fortune and men's eyes,
I all alone beweep my outcast state,
And trouble deaf heaven with my bootless cries,
And look upon myself, and curse my fate,
Wishing me like to one more rich in hope,
Featured like him, like him with friends possest,
Desiring this man's art and that man's scope,
With what I most enjoy contented least.
Yet in these thoughts myself almost despising,
Haply I think on thee - and then my state,
Like to the lark at break of day arising
From sullen earth, sings hymns at heaven's gate;
For thy sweet love remember'd such wealth brings
That then I scorn to change my state with kings.
<ctrl-D>
```

The second example consists of some of Babe Ruth's batting statistics, which we will use to demonstrate how **awk** processes tabular input. Type it into file **table1**, as follows:

```
cat > table1
1920    .376    54      158     137
1921    .378    59      177     171
1922    .315    35      94      99
1923    .393    41      151     131
1924    .378    46      143     121
1925    .290    25      61      66
1926    .372    47      139     145
1927    .356    60      158     164
1928    .323    54      163     142
1929    .345    46      121     154
<ctrl-D>
```

The columns give, respectively, the season, the batting average, and the numbers of home runs, runs scored, and runs batted in (RBIs).

The rest of this tutorial presents many examples that use these files. Type them in and run them! In that way, you can get a feel for **awk** into your fingers. Experiment; try some variations on the examples. Don't be afraid of making mistakes; this is one good way to learn the limits (and the strengths) of a language.

Using awk

awk reads input from the standard input (entered from your terminal or from a file you specify), processes each input line according to a specified **awk** program, and writes output to the standard output. This section explains the structure of an **awk** program and the syntax of **awk** command lines.

Command-line Options

The complete form for the **awk** command line is as follows:

awk [**-y**] [**-F***c*] [**-f** *progfile*] [*prog*] [*file1*] [*file2*] ...

The following describes each element of the command line.

-y Map *patterns* from lower case to both lower-case and upper-case letters. For example, with this option the string **the** would match **the** or **The**.

-F*c* Set the field-separator character to the character *c*. The field-separator character and its uses are described below.

-f *progfile*
Read the **awk** program from *progfile*

prog An **awk** program to execute. If you do not use the **-f** option, you must enter **awk**'s statements on its command line.

Note that if you include **awk**'s program on its command line (instead of in a separate file), you must enclose the program between apostrophes. Otherwise, some of the **awk** statements will be modified by the shell before **awk** ever sees them, which will make a mess of your program. For example:

```
awk 'BEGIN {print "sample output file"}
            {print NR, $0}'
```

(The following sections explain what the stuff between the apostrophes means.) However, if you include the statement within a file that you pass to **awk** via its **-f** option, you must *not* enclose the statements within parentheses; otherwise, **awk** will become very confused. If you were to put the statements in the above program into an **awk** program file, they would appear as follows:

```
BEGIN {print "sample output file"}
 {print NR, $0}
```

file1 file2 ...
The files whose text you wish to process. For example, the command

```
awk '{print NR, $0}' text1
```

prints the contents of **text1**, but precedes each line with a line number.

If you do not name an input file, **awk** processes what it reads from the standard input. For example, the command

```
awk '{print NR, $0}'
```

reads what you type from the keyboard and echoes it preceded with a line number. To exit from this program, type **<ctrl-D>**.

Structure of an awk Program

An **awk** program consists of one or more statements of the form:

pattern { *action* }

Note that **awk** insists that the action be enclosed between braces, so that it can distinguish the action from the pattern.

A program can contain as many statements as you need to accomplish your purposes. When **awk** reads a line of input, it compares that line with the *pattern* in each statement. Each time a line matches *pattern*, **awk** performs the corresponding *action*. **awk** then reads the next line of input.

A statement can specify an *action* without a *pattern*. In this case, **awk** performs the action on every line of input. For example, the program

```
awk '{ print }' text1
```

prints every line of **text1** onto the standard output.

An **awk** program may also specify a pattern without an action. In this case, when an input line matches the pattern, **awk** prints it on the standard output. For example, the command

```
awk 'NR > 0' table1
```

prints all of **table1** onto the standard output. Note that you can use the same pattern in more than one statement. Examples of this will be given below.

awk's method of forming patterns uses *regular expressions* (also called *patterns*), like those used by the COHERENT commands **sed**, **ed**, and **egrep**. Likewise, **awk**'s method of constructing actions is modelled after the C programming language. If you are familiar with regular expressions and with C, you should have no problem learning how to use **awk**. However, if you are not familiar with them, they will be explained in the following sections.

Records and Fields

awk divides its input into *records*. It divides each record, in turn, into *fields*. Records are separated by a character called the *input-record separator*; likewise, fields are separated by the *input-field separator*. **awk** in effect conceives of its input as a table with an indefinite number of columns.

The newline character is the default input-field separator, so **awk** normally regards each input line as a separate record. The space and the tab characters are the default input-field separator, so white space normally separates fields.

To address a field within a record, use the syntax **$***N*, where *N* is the number of the field within the current record. The pattern **$0** addresses the entire record. Examples of this will be given below. In addition to input record and field separators, **awk** provides output record and field separators, which it prints between output records and fields. The default output-field separator is the newline character; **awk** normally prints each output record as a separate line. The space character is the default output-field separator.

Patterns

This section describes how **awk** interprets the pattern section of a statement.

Special Patterns

To begin, **awk** defines and sets a number of special *pattern*s. You can use these patterns in your program for special purposes. You can also redefine some of these patterns to suit your preferences. The following describes the commonest such special *pattern*s, and how they're used:

BEGIN This pattern matches the beginning of the input file. **awk** executes all *actions* associated with this pattern before it begins to read input.

END This pattern matches the end of the input file. **awk** executes all *actions* associated with this pattern after it had read all of its input.

FILENAME
awk sets this pattern to the name of the file that it is currently reading. Should you name more than one input file on the command line, **awk** resets this pattern as it reads each file in turn.

FS Input-field separator. This pattern names the character that **awk** recognizes as the field separator for the records it reads.

NF This pattern gives the number of fields within the current record.

NR This pattern gives the number of the current record within the input stream.

OFS Output-field separator. **awk** sets this pattern to the character that it writes in its output to separate one field from another.

ORS Output-record separator. **awk** sets this pattern to the character that it writes in its output to separate one field from another.

RS Input-record separator. **awk** sets this pattern to the character by which it separates records that it reads.

Arithmetic Relational Expressions

An *operator* marks a task to be within an expression, much as the '+' or '/' within an arithmetic expression indicates that the numbers are to be, respectively, added or divided. You can use **awk**'s operators to:

- Compare a special pattern with a variable, a field, or a constant.
- Assign a value to a variable or to a special pattern.
- Dictate the relationship among two or more expressions.

The first type of operator to be discussed are *arithmetic relational operators*. These compare the input text with an arithmetic value. **awk** recognizes the following arithmetic operators:

<	Less than
<=	Less than or equal to
==	Equivalent
!=	Not equal
>=	Greater than or equal to
>	Greater than

With these operators, you can compare a field with a constant; compare one field with another; or compare a special pattern with either a field or a constant.

For example, the following **awk** program prints all of the years in which Babe Ruth hit more than 50 home runs:

```
awk '$3 >= 50' table1
```

(As you recall, column 3 in the file **table1** gives the number of home runs.) The program prints the following on your screen:

```
1920    .376    54      158     137
1921    .378    59      177     171
1927    .356    60      158     164
1928    .323    54      163     142
```

The following program, however, shows the years in which Babe Ruth scored more runs than he drove in:

```
awk '$4 > $5 { print $1 }' table1
```

Remember, field 4 in file **table1** gives the number of runs scored, and field 5 gives the number of runs batted in. You should see the following on your screen:

```
1920
1921
1923
1924
1928
```

In the above program, expression

```
{print $1}
```

defines the action to perform, as noted by the fact that expression is enclosed between braces. In this case, the program tells **awk** that if the input record matches the pattern, to print only the first field. However, to print both the season and the number of runs scored, use the following program:

```
awk '$4 > $5 { print $1, $4 }' table1
```

This prints the following:

```
1920 158
1921 177
1923 151
1924 143
1928 163
```

Note that **$1** and **$4** are separated by a comma. The comma tells **awk** to print its default output-field separator between columns. If we had left out the comma, the output would have appeared as follows:

```
1920158
1921177
1923151
1924143
1928163
```

As we noted above, the special pattern **OFS** gives the output-field separator. **awk** by default defines this special pattern to the space character. If we wish to redefine the output-field separator, we can use an operator, plus the special pattern **BEGIN**, as follows:

```
awk 'BEGIN { OFS = ":" }
       $4 > $5 { print $1, $4 }' table1
```

This prints:

```
1920:158
1921:177
1923:151
1924:143
1928:163
```

The first statement

```
BEGIN { OFS = ":"}
```

tells **awk** to set the output-field separator (the special pattern **OFS**) to ':' before it processes any input (as indicated by the special pattern **BEGIN**).

Although we're getting a little ahead of ourselves, note that there's no reason to print the fields in the order in which they appear in the input record. For example, if you wish to print the number of runs scored before the season, use the command:

```
awk 'BEGIN { OFS = ":"}
       $4 > $5 { print $4, $1 }' table1
```

This prints:

```
158:1920
177:1921
151:1923
143:1924
163:1928
```

As you recall, the special pattern **NR** gives the number of the current input record. You can execute an action by comparing this pattern with a constant. For example, the command

```
awk 'NR > 12' text1
```

prints:

```
For thy sweet love remember'd such wealth brings
That then I scorn to change my state with kings.
```

That is, the program prints every line after line 12 in the input file. As you recall, a statement that has a pattern but no action prints the entire record that matches the pattern.

As we saw with the special patterns, some patterns can be defined to be numbers and others to be text. If you compare a number with a string, **awk** by default makes a string comparison. The following example shows how **awk** compares one field to part of the alphabet:

```
awk '$1 <= "C"' text1
```

This program prints:

```
And trouble deaf heaven with my bootless cries,
And look upon myself, and curse my fate,
```

The statement **$1 <= "C"** selected all records that begin with an ASCII value less than or equal to that of the letter 'C' (0x43) — in this case, both lines that begin with 'A' (0x41). If we ran this example against **table1**, it would print every record in the file. This is because each record begins with the character '1' (0x31), which matches the pattern **$1 <= "C"**.

Finally, you can use a numeric field plus a constant in a comparison statement. For example, the following program prints all of the seasons in which Babe Ruth had at least 100 more runs batted in than home runs:

```
awk '$3 + 100 < $5 {print $1}' table1
```

This prints the following:

```
1921
1927
1929
```

Boolean Combinations of Expressions

awk has a number of operators, called *Boolean* operators, that let you hook together several small expressions into one large, complex expression. **awk** recognizes the following Boolean operators:

||	Boolean OR (one expression or the other is true)
&&	Boolean AND (both expressions are true)
!	Boolean NOT (invert the value of an expression)

(The eponym "Boolean" comes from the English mathematician George Boole.) In a Boolean expression, **awk** evaluates each sub-expression to see if it is true or false; the relationship of sub-expressions (as set by the Boolean operator) then determines whether the entire expression is true or false.

For example, the following program prints all seasons in which Babe Ruth hit between 40 and 50 home runs:

```
awk '$3 >= 40 && $3 <= 50 { print $1, $3 }' table1
```

This prints the following:

```
1923 41
1924 46
1926 47
1929 46
```

In the above program, **awk** printed its output only if the subexpression **$3 >= 40** was true *and* (**&&**) the subexpression **$3 <= 50** was true.

The next example demonstrates the Boolean OR operator. It prints all seasons for which Babe Ruth hit fewer than 40 home runs or more than 50 home runs:

```
awk '$3 < 40 || $3 > 50 { print $1, $3}' table1
```

This example prints the following:

```
1920 54
1921 59
1922 35
1925 25
1927 60
1928 54
```

In this example, **awk** printed its output if the subexpression **$3 < 40** was true *or* (**||**) the subexpression **$3 > 50** was true. Note that the output would also printed if both subexpressions were true (although in this case, this is impossible).

Finally, the Boolean operator '!' negates the truth-value of any expression. For example, the expression **$1 = "And"** is true if the first field in the current record equals "And"; however, the expression **$1 != "And"** is true if the first field does *not* equal "And". For example, the program

```
awk '$1 != "And"' text1
```

prints:

```
When, in disgrace with fortune and men's eyes,
I all alone beweep my outcast state,
Wishing me like to one more rich in hope,
Featured like him, like him with friends possest,
Desiring this man's art and that man's scope,
With what I most enjoy contented least.
Yet in these thoughts myself almost despising,
Haply I think on thee - and then my state,
Like to the lark at break of day arising
From sullen earth, sings hymns at heaven's gate;
For thy sweet love remember'd such wealth brings
That then I scorn to change my state with kings.
```

These are the 12 lines from **text1** that do not begin with "And".

Note that **awk** evaluates all operators from left to right unless sub-expressions are grouped together with parentheses, as is described in the following section.

Patterns

The previous examples have all matched strings or numbers against predefined fields in each input record. This is fine for manipulating tabular information, like our table of Babe Ruth's batting statistics, but it is not terribly useful when you are processing free text. Free text is not organized into predefined columns, nor are you likely to know which field (that is, which word) will contain the pattern you're seeking.

To help you manage free text, **awk** has a pattern-matching facility that resembles those of the editors **ed** and **sed**.

The most common way to search for a pattern is to enclose it between slashes. For example, the program

```
awk '/and/' text1
```

prints every line in **text1** that contains the string "and".

```
When, in disgrace with fortune and men's eyes,
And look upon myself, and curse my fate,
Desiring this man's art and that man's scope,
Haply I think on thee - and then my state,
```

Note that "and" does not have to be a word by itself — it can be a fragment within a word as well. Note, too, that this pattern matches "and" but does not match "And" — but it would if we were to use the **-y** option on the **awk** command line (described above).

You can use Boolean operators to search for more than one string at once. For example, the program

```
awk '/and/ && /or/' text1
```

finds every line in **text1** that contains both "and" and "or". There is only one:

```
When, in disgrace with fortune and men's eyes,
```

Note that the "or" in this line is embedded in the word "fortune".

awk can also scan for classes and types of characters. To do so, enclose the characters within brackets and place the bracketed characters between the slashes. For example, the following program looks for every line in **text1** that contains a capital 'A' through a capital 'E':

```
awk '/[A-E]/' text1
```

This prints the following:

```
And trouble deaf heaven with my bootless cries,
And look upon myself, and curse my fate,
Desiring this man's art and that man's scope,
```

In addition, you can use the following special characters for further flexibility:

[ ]	Class of characters
( )	Grouping subexpressions
|	Alternatives among expressions
+	One or more occurrences of the expression
?	Zero or more occurrences of the expression
*	Zero, one, or more occurrences of the expression
.	Any non-newline character

When adding a special character to a pattern, enclose the special character as well as the rest of the pattern within slashes.

To search for a string that contains one of the special characters, you must precede the character with a backslash. For example, if you are looking for the string "today?", use the following pattern:

```
/today\?/
```

When you need to find an expression in a particular field, not just anywhere in the record, you can use one of these operators:

~	Contains the data in question
!~	Does not contain the data in question

For example, if you need to find the digit '9' in the fourth field of file **table1**, use the following program:

```
awk '$4~/9/ {print $1, $4}' table1
```

This prints the following:

```
1922 94
1926 139
```

As you can see, the above program found every record with a '9' in its fourth field, regardless of whether the '9' came at the beginning of the field or its end. **awk** also recognizes two operators that let you set where a pattern is within a field:

^	Beginning of the record or field
$	End of the record or field

For example, to find every record in **table1** whose fourth field *begins* with a '9', run the following program:

```
awk '$4~/^9/ {print $1, $4}' table1
```

This prints:

```
1922 94
```

Finally, to negate a pattern use the operator **!~**. For example, to print every record in **table1** whose fourth column does *not* begin with a '9', use the following program:

```
awk '$4!~/^9/ {print $1, $4}' table1
```

This prints:

```
1920 158
1921 177
1923 151
1924 143
1925 61
1926 139
1927 158
1928 163
1929 121
```

Ranges of Patterns

You can tell **awk** to perform an action on all records between two patterns. For example, to print all records between the *patterns* **1925** and **1929**, inclusive, enclose the strings in slashes and separate them with a comma, then indicate the **print** action, as follows:

```
awk '/1925/,/1929/ { print }' table1
```

TUTORIALS

You can also use the special pattern **NR** (or *record number*) to name a range of record numbers. For example, to print records 5 through 10 of file **text1**, use the following program:

```
awk 'NR == 5, NR == 10 { print }' text1
```

Resetting Separators

As noted above, **awk** recognizes certain characters by default to parse its input into records and fields, and to separate its output into records and fields:

FS Input-field separator. By default, this is one or more white-space characters (tabs or spaces).

OFS Output-field separator. By default, this is exactly one space character.

ORS Output-record separator. By default, this is the newline character.

RS Input-record separator. By default, this is the newline character.

By resetting any of these special patterns, you can change how **awk** parses its input or organizes its output. Consider, for example, the command:

```
awk 'BEGIN {ORS = "|"}
       /1920/,/1925/ {print $1, $5}' table1
```

This prints the following:

```
1920 137|1921 171|1922 99|1923 131|1924 121|1925 66|
```

As you can see, this prints the season and the number of runs batted in for the 1920 through 1925 season. However, **awk** uses the pipe character '|' instead of the newline character to separate records. If you wish to change the output-field separator as well as the output-record separator, use the program:

```
awk 'BEGIN {ORS = "|" ;       OFS = ":"}
       /1920/,/1925/ {print $1, $5}' table1
```

This produces:

```
1920:137|1921:171|1922:99|1923:131|1924:121|1925:66|
```

As you can see, **awk** has used the colon ':' instead of a white-space character to separate one field from another.

Note, too, that the semicolon ';' character separates expressions in the action portion of the statement associated with the **BEGIN** pattern. This lets you associate more than one action with a given pattern, so you do not have to repeat that pattern. This is discussed at greater length below.

You can also change the input-record separator from the newline character to something else that you prefer. For example, the following program changes the input-record separator from the newline to the comma:

```
awk 'BEGIN {RS = ","}
       {print $0}' text1
```

This yields the following:

```
When
 in disgrace with fortune and men's eyes

I all alone beweep my outcast state

And trouble deaf heaven with my bootless cries

And look upon myself
 and curse my fate

Wishing me like to one more rich in hope

Featured like him
 like him with friends possest

Desiring this man's art and that man's scope

With what I most enjoy contented least.
Yet in these thoughts myself almost despising

Haply I think on thee - and then my state

Like to the lark at break of day arising
From sullen earth
 sings hymns at heaven's gate;
For thy sweet love remember'd such wealth brings
That then I scorn to change my state with kings.
```

The blank lines resulted from a comma's occurring at the end of a line.

Note that by specifying the null string **(RS="")**, you can make two consecutive newlines the record separator. Note, too, that only one character can be the input-record separator. If you try to reset this separator to a string, **awk** uses the first character in the string as the separator, and ignores the rest.

You can change the input-field separator by redefining **FS**. The default **FS** is **<space>\t** exactly and in that order (where **<space>** is the space character). In this case, **awk** uses its "white-space rule," in which **awk** treats any sequence of spaces and tabs as a single separator. This is the default rule for **FS**. If you set **FS** to anything else, including **\t<space>**, then each separator is separate. For example, the following program changes the input-field separator to the comma and prints the first such field it finds in each line from file **text1**:

```
awk 'BEGIN {FS = ","}
       {print $1}' text1
```

This produces:

```
When
I all alone beweep my outcast state
And trouble deaf heaven with my bootless cries
And look upon myself
Wishing me like to one more rich in hope
Featured like him
Desiring this man's art and that man's scope
With what I most enjoy contented least.
Yet in these thoughts myself almost despising
Haply I think on thee - and then my state
Like to the lark at break of day arising
From sullen earth
For thy sweet love remember'd such wealth brings
That then I scorn to change my state with kings.
```

As you can see, this program prints text up to the first comma in each line. **awk** throws away the comma itself, because the input-field separator is not explicitly printed.

You can define several characters to be input-field separators simultaneously. When you specify several characters within quotation marks, each character becomes a field separator, and all separators have equal precedence. For example, you can specify the letters 'i', 'j', and 'k' to be input-field separators. The following program does this, and prints the first field so defined from each record in file **text1**:

```
awk 'BEGIN  {FS = "ijk"}
       {print $1}' text1
```

This prints:

```
When,
I all alone beweep my outcast state,
And trouble deaf heaven w
And loo
W
Featured l
Des
W
Yet
Haply I th
L
From sullen earth, s
For thy sweet love remember'd such wealth br
That then I scorn to change my state w
```

Note that if you set the input-record separator to a null string, you can use the newline character as the input-field separator. This is a handy way to concatenate clusters of lines into records that you can then manipulate further.

One last point about the **FS** separator. If the white-space rule is not invoked and an assignment is made to a nonexistent field, **awk** can add the proper number of field separators. For example if **FS=":"** and the input line is **a:b**, then the command **$5 = "e"** produces **a:b:::e**. If the white-space rule were in effect, **awk** would add spaces as if each space were a separator, and print a warning message. In short, it would try to produce the sanest result from the error.

Finally, the variable **NR** gives the number of the current record. The next example prints the total number of records in file **text1**:

```
awk 'END {print NR}' text1
```

The output is

```
14
```

which is to be expected, since **text1** is a sonnet.

Actions

The previous section described how to construct a *pattern* for **awk**. For each pattern, there must be a corresponding *action*. So far, the only action shown has been to print output. However, **awk** can perform many varieties of actions. In addition to printing, **awk** can:

- Execute built-in functions
- Redirect output
- Assign variables
- Use fields as variables
- Define arrays
- Use control statements

These actions are discussed in detail in the following sections.

As noted above, each **awk** statement must have an action. If a statement does not include an action, **awk** assumes that the action is **{print}**.

Within each statement, **awk** distinguishes an action from its corresponding pattern by the fact that the action is enclosed within braces. Note that the action section of a statement may include several individual actions; however, each action must be separated from the others by semicolons ';' or newlines.

Some forms of **awk**, such as that provided by the Free Software Foundation (FSF), allow user-defined functions. The FSF version of **awk** is available from the MWC BBS as well as via COHware. Note that your system must have at least two megabytes of RAM to run the FSF version of **awk**.

awk Functions

awk includes the following functions with which you can manipulate input. You can assign a function to any variable or use it in a pattern. The following lists **awk**'s functions. Note that an *argument* can be a variable, a field, a constant, or an expression:

abs(*argument***)**
Return the absolute value of *argument*.

exp(*argument***)**
Return Euler's number *e* (2.178...) to the power of *argument*.

index(*string1***,***string2***)**
Return the position of *string2* within *string1*. If *s2* does not occur in *s1*, **awk** returns zero. This **awk** function resembles the COHERENT C function **index()**.

int(*argument***)**
Return the integer portion of *argument*.

length Return the length, in bytes, of the current record.

length(*argument***)**
Return the length, in bytes, of *argument*.

log(*argument***)**
Return the natural logarithm of *argument*.

print(*argument1 argument2 ... argumentN***)**
Concatenate and print *argument1* through *argumentN*.

print(*argument1***,***argument2***, ... argumentN***)**
Print *argument1* through *argumentN*. Separate each *argument* with the **OFS** character.

printf(*f***, *argument1*, ... *argumentN*)**
Format and print strings *argument1* through *argument* in the manner set by the formatting string *f*, which can use **printf()**-style formatting codes.

split(*str***, *array*, *fs*)**
Divide the string *str* into fields associated with *array*. The fields are separated by character *fs* or the default field separator.

sprintf(*f***, *e1*, *e2*)**
Format strings *e1* and *e2* in the manner set by the formatting string *f*, and return the formatted string. *f* can use **printf()**-style formatting codes.

sqrt(*argument***)**
Return the square root of *argument*.

substr(*str***, *beg*, *len*)**
Scan string *str* for position *beg*; if found, print the next *len* characters. If *len* is not included, print from from *beg* to the end of the record.

Printing with awk

Printing is the commonest task you will perform in your **awk** programs. **awk**'s printing functions **printf** and **sprintf** resemble the C functions **printf()** and **sprintf()**; however, there are enough differences to make a close reading of this section worthwhile.

print is the commonest, and simplest, **awk** function. When used without any arguments, **print** prints all of the current record. The following example prints every record in file **text1**:

```
awk '{print}' text1
```

You can print fields in any order you desire. For example, the following program reverses the order of the season and batting-average columns from file **table1**:

```
awk '/1920/,/1925/ { print $2,$1 }' table1
```

The output is as follows:

```
.376 1920
.378 1921
.315 1922
.393 1923
.378 1924
.290 1925
```

TUTORIALS

Because the field names are separated by a comma, **awk** inserts the **OFS** between the fields when it prints them. If you do not separate field names with commas, **awk** concatenates the fields when it printing them. For example, the program

```
awk '/1920/,/1925/ { print $2 $1 }' table1
```

produces:

```
.3761920
.3781921
.3151922
.3931923
.3781924
.2901925
```

When you use **awk** to process a column of text or numbers, you may wish to specify a consistent format for the output. The statement for formatting a column of numbers follows this *pattern*:

```
{printf "format", expression}
```

where *format* prescribes how to format the output, and *expression* specifies the fields for **awk** to print.

The following table names and defines the most commonly used of **awk**'s format control characters. Each character must be preceded by a percent sign '%' and a number in the form of *n* or *n.m*.

%*n***d**	Decimal number
%*n***.***m***f**	Floating-point number
%*n***.***m***s**	String
%%	Literal '%' character

When you use the **printf()** function, you must define the output-record separator within the format string. The following codes are available:

\n	Newline
\t	Tab
\f	Form feed
\r	Carriage return
\"	Quotation mark

For example, the following program prints Babe Ruth's RBIs unformatted:

```
awk '/1920/,/1925/ { print $1, $5 }' table1
```

The output appears as follows:

```
1920 137
1921 171
1922 99
1923 131
1924 121
1925 66
```

As you can see, **awk** right-justifies its output by default. To left-justify the second column, use the following program:

```
awk '/1920/,/1925/ { printf("%d %3d\n", $1, $5) }' table1
```

The output is as follows:

```
1920 137
1921 171
1922  99
1923 131
1924 121
1925  66
```

Note that the '3' in the string **%3d** specifies the minimum number of characters to be displayed. If the size of the number exceeds the space allotted to it, **awk** prints the entire number. A different rule applies when printing strings, as will be shown below.

To print a floating-point number, you must specify the minimum number of digits you wish to appear on either side of the decimal point. For example, the following program gives the average number of RBIs Babe Ruth hit in each game between 1920 and 1925:

```
awk '/1920/,/1925/ { printf("%d %1.2f\n", $1, $5/154.0) }' table1
```

This prints the following:

```
1920 0.89
1921 1.11
1922 0.64
1923 0.85
1924 0.79
1925 0.43
```

Note the following points about the above program:

- To get the average number of runs batted in, we had to divide the total number of RBIs in a season by the number of games in a season (which in the 1920s was 154). **awk** permits you to use a constant to perform arithmetic on a field; this will be discussed in more detail below.
- To force **awk** to produce a floating-point number, the constant had to be in the format of a floating-point number, i.e., "154.0" instead of "154". Dividing an integer by another integer would not have produced what we wanted.

awk rounds its output to match sensitivity you've requested — that is, the number of digits to the right of the decimal point. To see how sensitivity affects output, run the following program:

```
awk '/1920/,/1925/{printf("%1.2f %1.3f %1.4f\n",$5/154.0,$5/154.0,$5/154.0)}'\
table1
```

This prints the following:

```
0.89 0.890 0.8896
1.11 1.110 1.1104
0.64 0.643 0.6429
0.85 0.851 0.8506
0.79 0.786 0.7857
0.43 0.429 0.4286
```

As an aside, the above example also shows that you can break **awk**'s command line across more than one line using a backslash '\' at the end of every line but the last. Note, however, that you *cannot* break an **awk** statement across more than one line, or **awk** will complain about a syntax error.

One last example of floating-point numbers prints Babe Ruth's ratio of runs scored to runs batted in between 1920 and 1925:

```
awk '/1920/,/1925/{x = ($5*1.0) ; printf("%1.3f\n", $4/x)}' table1
```

This produces the following:

```
1.153
1.035
0.949
1.153
1.182
0.924
```

The expression **x = ($5*1.0)** was needed to turn field 5 (the divisor) into a floating-point number, so we could obtain the decimal fraction that we wanted. This is discussed further below, when we discuss how to manipulate constants.

The function **sprintf()** also formats expressions; however, instead of printing its output, it returns it for assignment to a variable. For example, you could rewrite the previous example program to replace the multiplication operation with a call to **sprintf()**:

```
awk '/1920/,/1925/{x = sprintf("%3.1f", $5)
       printf("%1.3f\n", $4/x)}' table1
```

The output is the same as that shown above.

TUTORIALS

The **%s** formatting string can be used to align text in fields. The digit to the left of the period gives the width of the field; that to the right of the period gives the number of characters to write into the field. Note that if input is larger than the number of characters allotted to it, **awk** truncates the input. For example, the following program aligns on seven-character fields some words from file **text1**:

```
awk '{x=sprintf("%7.5s %7.5s %7.5s %7.5s", $1, $2, $3, $4)
       print x}' text1
```

The output is as follows:

```
  When,      in   disgr    with
     I     all   alone   bewee
   And   troub    deaf   heave
   And    look    upon   mysel
 Wishi      me    like      to
 Featu    like    him,    like
 Desir    this   man's     art
  With    what       I    most
   Yet      in   these   thoug
 Haply       I   think      on
  Like      to     the    lark
  From   sulle   earth   sings
   For     thy   sweet    love
  That    then       I   scorn
```

Note that fields (words) longer than five characters are truncated; and every word is right-justified on a seven-character field.

Redirecting Output

In addition to printing to the standard output, **awk** can redirect the output of an action into a file, or append it onto an existing file. With this feature, you can extract information from a given file and construct new documents. The following example shows an easy way to sift Babe Ruth's statistics into four separate files, for further processing:

```
awk '{  print $1, $2 > "average"
        print $1, $3 > "home.runs"
        print $1, $4 > "runs.scored"
        print $1, $5 > "rbi"}' table1
```

Note like as under the shell, the operator '>' creates the named file if it does not exist, or replaces its contents if it does. To append **awk**'s onto the end of an existing file, use the operator '>>'.

awk can also pipe the output of an action to another program. As under the shell, the operator '|' pipes the output of one process into another process. For example, if it is vital for user **fred** to know Babe Ruth's batting average for 1925, you can mail it to him with the following command:

```
awk '/1925/ {print $1, $2 | "mail fred"}' table1
```

Assignment of Variables

A number of the previous examples assign values to variables. **awk** lets you create variables, perform arithmetic upon them, and otherwise work with them.

An **awk** variable can be a string or a number, depending upon the context. Unlike C, **awk** does not require that you declare a variable. By default, variables are set to the null string (numeric value zero) on start-up of the **awk** program. To set the variable **x** to the numeric value one, you can use the assignment operator '=':

```
x = 1
```

To set x to the string **ted** also use the assignment operator:

```
x = "ted"
```

When the context demands it, **awk** converts strings to numbers or numbers to strings. For example, the statement

```
x = "3"
```

initializes to **x** to the string "3". When an expression contains an arithmetic operator such as the '-', **awk** interprets the expression as numeric. (Alphabetic strings evaluate to zero.) Therefore, the expression

```
x = "3" - "1"
```

assigns the numeric value two to variable **x** not the string "2".

When the operator is included within the quotation marks, **awk** treats the operator as a character in the string. In the following example

```
x = "3 - 1"
```

initializes **x** to the string "3 - 1".

A number of examples in the previous section showed you how to perform arithmetic on fields. The following table gives **awk**'s arithmetic operators:

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement
+=	Add and assign value
-=	Subtract and assign value
***=**	Multiply and assign value
/=	Divide and assign value
%=	Divide modulo and assign value

Variables are often used with increment operators. For example, the following program computes the average number of home runs Babe Ruth hit each season during the 1920s:

```
awk '   { x += $3 }
        END { y = (NR * 1.0)
                printf("Average for %d years: %2.3f.\n", NR, x/y) }' table1
```

The output is:

```
Average for 10 years: 46.700.
```

Field Variables

awk lets fields receive assignments, be used in arithmetic, and be manipulated in string operations. One task that has not yet been demonstrated is using a variable to address a field. For example, the following program prints the **NR**th field (word) from the first seven lines in file **text1**:

```
awk 'NR < 8 {print NR, $(NR)}' text1
```

The output is:

```
1 When,
2 all
3 deaf
4 myself,
5 one
6 with
7 man's
```

Control Statements

awk has seven defined control statements. This section explains them and gives examples of their use.

if (*condition***)** *action1* **[else** *action2* **]**

If *condition* is true, then execute *action1*. If the optional **else** clause is present and *condition* is false, then execute *action2*.

The following program keeps running totals of Babe Ruth's RBIs, for both the years where his runs scored exceeded his RBIs and the years where they did not:

```
awk '{ if ( $4 > $5 )
            gyear++
       else
            lyear++
     }
     END { printf("Scored exceed RBIs: %d years.0, gyear)
           printf("Scored not exceed RBIs: %d years.0, lyear)
     }' table1
```

This produces:

```
Scored exceed RBIs: 5 years.
Scored not exceed RBIs: 5 years.
```

Note that if more than one action is associated with an **if** or **else** statement, you must enclose the statements between braces. If you use braces with both the **if** and **else** statements, note that the beginning and closing braces *must* appear on the same line as the **else** statement. For example:

```
if (expr) {
        stuff
        stuff
} else {
        stuff
        stuff
}
```

while (*condition***)** *action*

The **while** statement executes *action* as long as *condition* is true. For example, the following program counts the number of times the word **the** appears in file **text1**. The **while** loop uses a variable to examine every word in every line:

```
awk ' { i = 1
        while (i <= NF ) {
                if ($i == "the") j++
                i++
        }
      }
END   { printf ("The word \"the\" occurs %d times.\n", j) }' text1
```

The result, as follows, shows Shakespeare's economy of language:

```
The word "the" occurs 1 times.
```

By the way, note that if a control statement has more than one statement in its action section, enclose the action section between braces. If you do not, **awk** will behave erratically or exit with a syntax error.

for(*initial* **;** *end* **;** *iteration* **)** *action*
for(*variable* **in** *array* **)** *action*

awk's **for** statement closely resembles the **for** statement in the C language. The statement *initial* defines actions to be performed before the loop begins; this is usually used to initialize variables, especially counters. The statement *end* defines when the loop is to end. The statement *iteration* defines one or more actions that are performed on every iteration of the loop; usually this is used to increment counters. Finally, *action* can be one or more statements that are executed on every iteration of the loop. *action* need not be present, in which case only the action defined in the *iteration* portion of the **for** statement is executed. **for** is in fact just an elaboration of the **while** statement, but adjusted to make it a little easier to use. The following example writes the previous example, but replaces the **while** loop with a **for** mechanism:

```
awk ' { for (i = 1 ; i <= NF ; i++)
             if ($i == "the") j++
      }
END   { printf ("The word \"the\" occurs %d times.\n", j) }' text1
```

The output is the same as the previous example, but the syntax is neater and easier to read.

The second form of the **for** loop examines the contents of an array. It is described in the following section, which introduces arrays.

break The statement **break** immediately interrupts a **while** or **for** loop. For example, the following program is the same as the previous example, but counts only the first occurrence of the word **the** in each line of **text1**. Thus, it counts the number of lines in **text1** that contain **the**:

```
awk '{ for (i = 1 ; i <= NF ; i++) {
           if ($i == "the") {
                  j++
               break
           }
       }
     }
END   {printf ("The word \"the\" occurs in %d lines.\n", j)}' text1
```

continue

The statement **continue** immediately begins the next iteration of the nearest **while** or **for** loop. For example, the following program prints all of Babe Ruth's statistics — runs scored, runs batted, and home runs — in which he had more than 59 in one year:

```
awk ' { for (i = 3 ; i <= NF ; i++)
           if ($i <= 59)
               continue
           else
               printf("%d, column %d: %d\n", $1, i, $i)
     } ' table1
```

This produces the following:

```
1920, column 4: 158
1920, column 5: 137
1921, column 4: 177
1921, column 5: 171
1922, column 4: 94
1922, column 5: 99
...
```

next The statement **next** forces **awk** abort the processing of the current record and skip to the next input record. Processing of the new input record begins with the first pattern, just as if the processing of the previous record had concluded normally. To demonstrate this, the following program skips all records in file **text1** that have an odd number of fields (words):

```
awk ' { if (NF % 2 == 0) next }
      { print $0 } ' text1
```

This produces:

```
I all alone beweep my outcast state,
Wishing me like to one more rich in hope,
With what I most enjoy contented least.
Yet in these thoughts myself almost despising,
Like to the lark at break of day arising
```

exit Finally, the control statement **exit** forces the **awk** program to skip all remaining input and execute the *actions* at the **END** pattern, if any. For example, the following program prints the year in which Babe Ruth hit his 300th home run:

```
awk ' { i =  $1 }
      (j += $3) >= 300 { exit }
      END {print "Babe Ruth hit his 300th homer in", i "."}' table1
```

This produces:

```
Babe Ruth hit his 300th homer in 1926.
```

Arrays

awk has a powerful feature for managing arrays. Unlike C, **awk** automatically manages the size of an array, so you do not have to declare the array's size ahead of time. Also, unlike C, **awk** lets you address each element within an array by a label, not just by its offset within the array. This lets you generate arrays "on the fly," which can be very useful in transforming many varieties of data.

To declare an array, simply name it within a statement. **awk** recognizes as an array every variable that is followed by brackets '[ ]'. To initialize a row within an array, you must define its value and name its label. A label can be either a number or a string. A value, too, can be a number or a string; if the value is a number, then you can perform arithmetic upon it, as will be shown in a following example.

Initializing an Array

To demonstrate how an array works, use the line editor **ed** to add a line of text to the beginning of file **table1**. Type the following; please note that the token **<tab>** means that you should type a tab character:

```
ed table1
1i
Year<tab>BA<tab>HRs<tab>Scored<tab>RBIs
.
wq
```

This change writes a header into **table1** that names each column. Now, we can read these labels into an array and use them to describe Babe Ruth's statistics. For example, the following prints a summary of Babe Ruth's statistics for the year 1926:

```
awk '   NR == 1 { for (i=1; i <= NF; i++) header [i] = $i }
        $1 == 1926 {
                for (i=1; i <= NF; i++)
                        print header[i] ":\t", $i
        } ' table1
```

This produces:

```
Year:           1926
BA:             .372
HRs:            47
Scored:         139
RBIs:           145
```

The statement

```
NR == 1 { for (i=1; i <= NF; i++) header [i] = $i }
```

reads the first line in **table1**, which contains the column headers, and uses the headers to initialize the array **header**. Each row is labeled with the contents of the variable **i**.

The loop

```
for (i=1; i <= NF; i++)
        print header[i] ":\t", $i
```

prints the contents of **header**. Because we labeled each row within **header** with a number, we can use a numeric loop to read its contents.

The for() Statement With Arrays

In the previous example, each element in the array was labeled with a number. This permitted us to read the array with an ordinary **for** statement, which sets and increments a numeric variable. However, the rows within an array can be labeled with strings, instead of numbers. To read the contents of such an array, you must use a special form of the **for** statement, as follows:

for (*offset* **in** *array* **)**

array names the array in question. *offset* is a variable that you name at the time of contructing the **for** statement. You can use the value of *offset* in any subordinate printing actions.

The following program demonstrates this new form of **for**, and (incidentally) to demonstrate the power of **awk**'s array-handling feature. It builds an array of each unique word in the file **text1**, and notes the number of times that word occurs within the file:

```
awk '   { for (i = 1 ; i <= NF ; i++)
                words [$i]++ }
        END { for (entry in words)
                print entry ":", words[entry] }' text1 | sort
```

This prints:

```
-: 1
And: 2
Desiring: 1
Featured: 1
For: 1
From: 1
Haply: 1
I: 4
Like: 1
That: 1
When,: 1
Wishing: 1
With: 1
Yet: 1
all: 1
almost: 1
...
```

As you can imagine, a similar program in C would require many more lines of code. However, a few features of this program are worth noting.

First, the expression

```
{ for (i = 1 ; i <= NF ; i++)
       words [$i]++ }
```

declares the array **words** Every time **awk** encounters a new field (word), it automatically adds another entry to the array, and labels that entry with the word. No work on your part is needed for this to happen. The '++' operator increments the value of the appropriate entry within **words**. Because we did not initialize the entry, it implicitly contains a number.

The expression

```
{ for (entry in words)
       print entry ":", words[entry] }
```

walks through the array **words**. **awk** initializes the variable **entry** to the label for each row in **words**; the **print** statement then prints **entry** and the contents of that row in the array — in this case, the number of times the row appears in our input file.

Finally, we piped the output of this program to the command **sort** to print the words in alphabetical order.

For More Information

This tutorial just gives a brief introduction to the power of **awk**. To explore the language in depth, see *sed & awk* by Dale Dougherty (Sebastopol, Calif, O'Reilly & Associates, Inc., 1985). This book, however, describes a more complex version of **awk** than that provided with COHERENT.

The Lexicon's article on **awk** gives a quick summary of its features and options.

Introduction to lex, the Lexical Analyzer

Many computer applications involve reading text strings. This is especially true for man-machine communication.

For some forms of textual input, a programmer can design a program by hand to process it. However, it is much easier to implement such programs when you use a software tool that will automatically construct a program to process the data. The COHERENT command **lex** is such a tool.

lex accepts expressions that describe the text input, and generates a program to process it. In computer-ese, **lex** is a "lexical scanner program generator".

This document tells you how to use **lex**. It presents many simple examples to illustrate how to use its features and how to use the generated program with other tools provided with COHERENT, notably the parser generator **yacc**.

Readers of this document are presumed to be familiar with the C programming language and the use of the COHERENT system. Related documents include *Using the COHERENT System* and the tutorial to yacc, the COHERENT parser generator.

How To Use lex

lex generates lexical scanners for compilers, to do statistical analysis of text, and to generate filters for many diverse tasks. This section gives examples of how to use **lex**. Later sections discuss the concepts used in these examples in detail.

Translating Strings

The first example tells **lex** to match an input string and replace it with a different string — in this case, replace the misspelled word "removeable" with the correctly spelled "removable". The program outputs unchanged all strings that it does not recognize. Enter the following program into the file **rmv.lex**.

```
%%
removeable     printf ("removable");
```

This creates the **lex** specification. Use the following command line to pass this specification through **lex**:

```
lex rmv.lex
```

This produces a C program named **lex.yy.c**, which you can compile by typing:

```
cc lex.yy.c -ll -o rmv
```

The executable program **rmv** is now ready to use. To illustrate its use, type:

```
rmv
Is this file removeable?
<ctrl-D>
```

rmv replies:

```
Is this file removable?
```

Note that the generated program reads from standard input and writes to standard output.

Remove Blanks From Input

The next example deletes all blanks and tabs from the input. Type the following **lex** program into file **nosp.lex**:

```
%%
[ \t]+  ;
```

Generate and compile the program with the following commands:

```
lex nosp.lex
cc lex.yy.c -ll -o nosp
```

To invoke the program, type **nosp**. Now, test it by typing the following:

```
This may be hard to read after processing.
<ctrl-D>
```

nosp outputs:

```
Thismaybehardtoreadafterprocessing.
```

Trimming Blanks

The previous example can be rewritten to remove strings of blanks or tabs and replace them with one space. Type the following into file **onesp.lex**:

```
%%
[ \t]+  printf (" ");
```

Generate and compile this with the following commands:

```
lex onesp.lex
cc lex.yy.c -ll -o onesp
```

Invoke your program with the command **onesp**. Now, type the following text to test the program; be sure to separate each word by two spaces:

```
This  should  be  easier  to  read.
<ctrl-D>
```

onesp prints the following:

```
This should be easier to read.
```

lex Specification Form

This section discusses the form of the **lex** specification.

Simple Form

The examples shown above use the simplest form of a **lex** program. Consider the text of the example **rmv.lex**:

```
%%
removeable     printf ("removable");
```

The symbol

```
%%
```

divides sections of the **lex** specification. Not all specifications need to be present, but at least one **%%** must appear in a **lex** program.

This symbol separates **lex** *definitions* from *rules*. With nothing before the **%%**, there are no definitions. Rules follow the **%%**. No definitions are needed in the simplest of **lex** specifications.

Rules in lex

The format of a **lex** rule is simple. Every rule has two parts. Refer to the program **rmv**:

```
removeable     printf ("removable");
```

The first part begins at the beginning of the line and ends with a space or tab. In the example rule, the first part is

```
removeable
```

This part is called the *pattern*.

The second part follows the space or tab, and is called the *action*. The action in this example is:

```
printf ("removable");
```

When the pattern specified by the rule is found in the input, the corresponding action is performed. Thus, this rule detects every appearance of *removeable* and outputs the correct spelling.

A **lex** program tries each rule's pattern in turn, and performs the associated action if and only if the pattern matches. Actions often modify the input that matched the pattern; they may also do nothing for certain patterns. To illustrate this, type the following specification into file **erase.lex**:

```
%%
erase   ;
```

Then compile the generated program with the following commands:

```
lex erase.lex
cc lex.yy.c -ll -o erase
```

This program copies all its input to its output, except for any appearance of the string **erase**. Invoke the program by typing **erase**, and then test it by typing:

```
Have you erased the blackboard?
<ctrl-D>
```

erase then prints:

```
Have you d the blackboard?
```

If the input contains patterns that do not match any of the patterns in the suite of rules you typed into **lex**, they are simply output unchanged. Usually, you will want to write a rule to cover every case.

Statements in lex

As noted earlier, **lex** is a program generator. It reads the specifications that you prepare for it, and writes a C program that is used with the **lex** library. Many of the actions in the rules you specify, such as

```
printf ("removable");
```

are themselves C statements. These statements are included in the resulting program, along with other statements that **lex** provides so the program can run.

You can include other statements, should the program need them, by placing them in appropriate places. The following program, called **count.lex**, shows how this is done. It counts the number of *tokens*, or strings of non-blank characters. Type the following into the file **count.lex** exactly as printed:

```
        int count;
%%
[^ \t\n]+      count++;
[ \t\n]+       ;
%%
yywrap ()
{
        printf ("Number of tokens:%d\n", count);
        return (1);
}
```

Statements other than rule actions appear in two places in the program. The first such statement is in the definition section, which precedes the rule section delimiter **%%**:

```
        int count;
```

This C statement declares the variable **count** to be an integer variable. Notice that it is preceded by a tab; a tab or a space indicates to **lex** that an input line is not a rule.

The second kind of non-rule statement follows the second **%%**, which marks the end of the rules section. **lex** regards anything that follows the second delimiter as being source statements.

The above example includes a function named **yywrap**. **lex** programs always call this function at the end of processing. The above program fills this function with code that prints the number of tokens in the text. If you do not include a routine named **yywrap**, **lex** will use a standard one from its library.

Compile the program by typing the following commands:

```
lex count.lex
cc lex.yy.c -ll -o count
```

Run the program by typing:

```
count <count.lex
```

This counts the tokens in the **count.lex** file itself. **count** will print the following:

```
Number of tokens:21
```

Groups of Statements

In previous examples, the C statement in the action part of the rule is a single statement. In many **lex** applications, however, you will need to use more than one statement per rule.

To do so, enclose the statements in the braces { }. The following example illustrates grouping. This **lex** specification generates a program to add numbers found in the input and print the total whenever it reads an asterisk '*'. Type the following program into **nsum.lex**:

```
         int number, sum;
%%
[0-9]+ {
       sscanf (yytext, "%d", &number);
       sum += number;
       printf ("%s", yytext);
       }
"*"    {
       printf ("%s", yytext);
       printf ("%d", sum);
       sum = 0;
       }
```

Compile the program by typing:

```
lex nsum.lex
cc lex.yy.c -ll -o nsum
```

To run the generated **nsum** program, enter a sample data file by typing:

```
cat >numbers
one two three
1 2 3 4 * 1 2 3 5 *
*
done
<ctrl-D>
```

Run the program by typing:

```
nsum <numbers
```

nsum will print:

```
one two three
1 2 3 4 *10 1 2 3 5 *11
*0
done
```

The statements that follow the definitions

```
[0-9]+
```

and

```
*
```

are enclosed in braces, because each action triggers several statements. Consider the first of these:

```
[0-9]+ {
       sscanf (yytext, "%d", &number);
       sum += number;
       printf ("%s", yytext);
       }
```

The pattern looks for strings of digits. **sscanf** converts each such string into a number and saves it in the variable **number**. Now, consider the second rule:

```
"*"     {
        printf ("%s", yytext);
        printf ("%d", sum);
        sum = 0;
        }
```

This specifies that upon detection of * in the input, the program is to print the sum of the numbers and then reset the counter to zero. In both of these rules, the statement

```
printf ("%s", yytext);
```

prints the number or * so that the output shows the input as well as the total. **lex** defines the variable **yytext**. It always contains the string that matches the pattern.

If the input is neither a number nor an asterisk, no pattern specifically matches it. Therefore, the program echoes it unchanged to the standard output.

Using the Same Action

To make it easier for you to write actions, **lex** allows you to *abbreviate* rules; that is, you have to write only once any action that is performed upon detection of several patterns. To abbreviate rules represented symbolically by

```
p1      action1
p2      action1
```

use the vertical bar operator:

```
p1      |
p2      action1
```

The vertical bar means "use the action from the first rule that declares an action." An example is given in the section on macro abbreviations, below.

Patterns

The first part of each rule in the **lex** rules section is a pattern that may match parts of the input. This section describes how to construct these patterns, sometimes called *regular expressions*. If you are familiar with **ed** and how its patterns work, this will be familiar to you.

Simple Patterns

The simplest kind of pattern is a string of characters that matches itself. A previous section presented an illustration of this:

```
%%
removeable     printf ("removable");
```

The regular expression "removeable" matches all occurrences of *removeable* that appear in the input text.

Certain characters have special meaning to **lex** patterns. To match a special character literally, you must *quote* it. For example, * has special meaning. To match the asterisk literally (that is to match any '*'s that appear in the input), surround it with quotation marks:

```
"*"
```

Another way to quote characters is to precede it with the backslash character '****'.

```
\*
```

The following characters each have special meaning and must be quoted to be matched as text characters:

```
" \ ( ) < > { } % * + ? [ ] - ^ / $ . |
```

However, within **"**, the **** still has its meaning, so to match the string ***** use the regular expression:

```
"\\*"
```

Also, to match a quote character, use:

```
\"
```

Classes of Characters

The power of patterns comes from special characters that match more than one character. The following examines each special character in detail.

The *period* or *dot* matches any character except newline. The following regular expression matches any pair of characters that begins with **J**:

```
J.
```

The following example prints in square brackets any sequence of five characters that precedes a blank. Type the following into the file **five.lex**:

```
%%
....." "       printf ("[%s]", yytext);
```

Compile the program with the following commands:

```
lex five.lex
cc lex.yy.c -ll -o five
```

Now, type the following to create a test file for **five**:

```
cat > work
how well does this work?
no match
<ctrl-D>
```

Now, test **five** by typing:

```
five < work
```

The result is:

```
how[ well ]does[ this ]work?
no match
```

The second line of the input does not have any matches. Because the **dot** pattern character does not match the end-of-line character, all five characters that precede the blank must be on the same line.

Another way to match many characters, but selectively, is with the *character class* operation. Enclose in square brackets the set of characters to be matched. Any of the characters listed there will match one character of the input. For example,

```
[0123456789]
```

matches any decimal digit in the input. Characters may be in any order within the brackets. Thus

```
[0246813579]
```

is equivalent to the example above.

To simplify specifying for character classes, you can specify ranges of characters. The beginning and end of the range is separated by a hyphen. To match all decimal digits as above, use:

```
[0-9]
```

To match all alphabetic characters, type:

```
[a-zA-Z]
```

The special character **^**, when used after the opening bracket '**[**', tells **lex** to match any character *except* those enclosed. The following example finds all strings that consist of two digits followed by a third character that is neither an alphabetic character nor a period, and prints them enclosed by **{** and **}**. Type the following into file **twodig.lex**:

```
%%
[0-9][0-9][^\.a-zA-Z]  printf ("{%s}", yytext);
```

Process and compile the program by typing the following commands:

```
lex twodig.lex
cc lex.yy.c -ll -o twodig
```

TUTORIALS

Invoke the program by typing **twodig**, and test it by entering the following text:

```
12. 12 12a 1 12 b
<ctrl-D>
```

twodig prints the following in reply:

```
12. {12 }12a 1 {12 }b
```

Repetition

In the patterns shown so far, each character matches only one character at a time. However, many interesting input patterns involve repetition of characters. The above program **twodig.lex** used such repetition, albeit in a primitive way.

To match one or more instances of a character, follow it with the pattern operator **+**. Consider the summation example in **nsum.lex**, shown earlier, which recognized strings of input numbers and added them to a total:

```
[0-9]+  {
        sscanf (yytext, "%d", &number);
        sum += number;
        printf ("%s", yytext);
        }
```

The pattern

```
[0-9]+
```

matches a string of one or more digits.

The operator * will match *zero* or more characters of a specified type. The following example deletes all characters between square brackets. Type it into file **star.lex**:

```
%%
\[.*\]  printf ("[]");
```

Type the following commands to generate and compile the program:

```
lex star.lex
cc lex.yy.c -ll -o star
```

Type the following to create a test file:

```
cat > disappear
[This should disappear]
[what happens with two] of them [on a line?]
<ctrl-D>
```

Now, use the test file with **star**:

```
star < disappear
```

The output is:

```
[]
[]
```

In looking at the example's input, you might have expected the output to be:

```
[]
[] of them []
```

lex does not produce the latter output because it generates recognizers that find the longest match if several matches are possible. Therefore, **star** matched the first **[**, then all characters up to and including the second **]**. When you write a pattern that matches many characters, you should bear this possibility in mind.

To change the program to match the first **]**, rewrite it as follows:

```
%%
\[[^\]]*\]     printf ("[]");
```

The regular expression now matches a string of all characters except a **]**, when that string is enclosed in square brackets.

The '**?**' character signals that the previous character or regular expression is optional. In other words, '**?**' signals zero or one instance of a character or regular expression.

To see how this would be used in a program, consider a text processor that regards a word as being a strings of alphabetic characters that may or may not be followed by a period. The following example does this, and encloses the recognized words in parentheses. Enter it into file **word.lex**:

```
%%
[a-zA-Z]+\.?   printf ("(%s)", yytext);
```

Generate and compile the program with the following commands:

```
lex word.lex
cc lex.yy.c -ll -o word
```

Create a test file:

```
cat > words
These are words.
Question mark not included?
<ctrl-D>
```

And test **word** with the following command:

```
word < words
```

The result is

```
(These) (are) (words.)
(Question) (mark) (not) (included)?
```

The question mark, like the * and **+** operators, can also follow another specification of a pattern. If you wanted to end a sentence with a character other than a period, the following code will do the job for you:

```
[a-zA-Z]+[.?!,]?
```

The characters

```
.?!,
```

are optional.

The '+' and '*' operators may match many characters. If you wish to match a specific number of characters or patterns, follow the patterns with the repetition within braces **{** and **}**. For example

```
[0-9]{3}
```

matches a string of exactly three characters. With this information, you should be able to rewrite the pattern part of **twodig.lex**, described above.

You can also specify a range of counts. To match from seven to nine occurrences of lower-case alphabetic characters, use:

```
[a-z]{7,9}
```

Choices and Grouping

To indicate alternate choices of characters or regular expressions, separate them in the regular expression with a vertical bar operator **|**. For example, if you wish to match either three decimal digits or the character **a**, use:

```
[0-9]{3}|a
```

Parentheses help to group the parts of the pattern that are separated by the vertical bar:

```
(abc)|(def)
```

This pattern will match either the string **abc** or the string **def**.

Matching Non-Graphic Characters

Non-special, graphic characters in patterns match themselves. Most non-graphic characters, such as space, tab, and control characters, cannot be matched directly. **lex** provides special sequences to match control characters. The following example removes tabs and blanks from the beginning and end of input lines. Type it into file **deblank.lex**:

```
%%
[ \t]+\n       printf ("\n");
\n[ \t]+       printf ("\n");
```

Generate and compile the program with the following commands:

```
lex deblank.lex
cc lex.yy.c -ll -o deblank
```

Type the following to create a test file:

```
cat > sportab
begins with no space or tab
        begins with tab
   begins with three spaces
<ctrl-D>
```

Type the following to test **deblank**:

```
deblank < sportab
```

The result is:

```
begins with no space or tab
begins with tab
begins with three spaces
```

The special regular expression **\t** represents *tab*, and **\n** represents *newline*.

To match the backspace character, use **\b**. Form feed is matched by **\f**. To match an arbitrary character with a known octal value, use three octal digits after the backslash; for example,

```
\007
```

More Patterns

This section discusses more advanced capabilities of patterns.

Line Context

Like **ed**, **lex** patterns can include characters that represent the beginning and end of line. To match a line that consists of exactly five alphabetic characters, type:

```
^[a-zA-Z]{5}$
```

The character **^** matches the beginning of the line, and **$** matches the end of the line.

Context Matching

A slash (virgule) '**/**' shows that a following context is necessary to match a string. For example, the following program matches the string **match** only if it is immediately followed by the string **ing**. Type it into file **match.lex**:

```
%%
match/ing      printf ("{%s}", yytext);
```

To compile the program, type the following commands:

```
lex match.lex
cc lex.yy.c -ll -o match
```

Type the following to create a test file:

```
cat > matchtit
Will this match?
This is a matching test.
<ctrl-D>
```

And run it against **match** by typing:

```
match < matchtit
```

The result is:

```
Will this match?
This is a {match}ing test.
```

Notice that the string before the slash is matched. The program does not match the part that follows the slash, even though the string must be there for the first part to be matched. Thus, the regular expression that follows the slash may also be matched on its own. To see how this works, type the following into the file **match2.lex**:

```
%%
match/ing      printf ("{%s}", yytext);
ing            printf ("ed");
```

To compile the program, type the following commands:

```
lex match2.lex
cc lex.yy.c -ll -o match2
```

Once again, create a test file:

```
cat > matching
Will this match?
This is a matching test.
You must now sing for your supper.
<ctrl-D>
```

And run it:

```
match2 < matching
```

The result is:

```
Will this match?
This is a {match}ed test.
You must now sed for your supper.
```

The context-string that follows the **/** may be a regular expression. The following example matches the whole-number portion of a decimal fraction. Type it into the file **wholept.lex**:

```
%%
"-"?[0-9]+/"."[0-9]+  printf ("(%s)", yytext);
```

To compile the program, type the following commands:

```
lex wholept.lex
cc lex.yy.c -ll -o wholept
```

Invoke the program by typing **wholept**; then type the following to test it:

```
123 12345 1234.567
1       <ctrl-D>
```

The result will be:

```
123 12345 (1234).567
```

As you can see, the part of the regular expression

```
"-"?
```

matches an optional leading minus sign. Then

```
[0-9]+
```

matches a string of at least one decimal digit. Then, the following context must match the regular expression

```
"."[0-9]+
```

which matches the fractional part of the number. When it finds a number that matches, it prints the number's whole part enclosed in parentheses.

Macro Abbreviations

lex also provides a macro facility that can substantially simplify the writing of complex regular expressions.

A *macro* is a named body of text. A macro processor simply replaces the name of the macro with the text of the macro.

To illustrate, type following example into file **float.lex**. It recognizes integer and floating point constants according to the C format:

```
d [0-9]+
e [Ee][+-]?[0-9]+
%%
{d}\.          |
{d}\.{d}       |
\.{d}          |
{d}\.{e}       |
\.{d}{e}       |
{d}\.{d}{e}    |
{d}{e}         printf ("F:[%s]", yytext);
```

lex replaces the macro name **e** with the code that matches a string of digits at least one digit long. It replaces the macro name **d** with code that matches the number's exponent. These two are invoked in the manner of

```
{d}
```

within a pattern. To compile the program, type the following commands:

```
lex float.lex
cc lex.yy.c -ll -o float
```

Type the following to create a test file:

```
cat > flonumb
1 1. 1.2 1.e4 1e4
.1e4 e4 .1 . 0 1.2e3
<ctrl-D>
```

And test it by typing:

```
float < flonumb
```

The result is:

```
1 F:[1.] F:[1.2] F:[1.e4] F:[1e4]
F:[.1e4] e4 F:[.1] . 0 F:[1.2e3]
```

Context: Start Rules

Many tasks in lexical processing require the program to be aware of a token's context. **lex** lets you make processing conditional upon previously processed input. This is done by using **start conditions**.

Start conditions are named in the definitions section as follows:

```
%S name1 name2
```

where **name1** and **name2** are names of start conditions. These start conditions are then used by prefixing a pattern with the start condition's name enclosed in angle brackets. For example:

```
<name1>
```

For example, you can use one start condition to control the scanning of comments in a Pascal-like language. The start condition is set by the **lex** statement **BEGIN** when the beginning bracket of the comment is found. The comment is scanned for strings that begin with **$** to signal compiler operation. To see how this works, type the following into the file **comment.lex**:

```
%S CMNT
%%
<CMNT>\$[ler]     printf ("Option is %s.\n", yytext);
<CMNT>[^\}]       ;
<CMNT>\}          BEGIN 0;
\{                BEGIN CMNT;
```

To compile, use the following commands:

```
lex comment.lex
cc lex.yy.c -ll -o comment
```

Once again, create a test file:

```
cat > option
{This is a comment}
{This comment has options $l $e $r}
program
information
<ctrl-D>
```

And run it by typing:

```
comment < option
```

The result is:

```
Option is $l.
Option is $e.
Option is $r.

program
information
```

The context start condition is named following **BEGIN** in the action part of the rule. To return to the normal condition, use **0** as the context name.

Separate Contexts

If you wish to perform context-dependent processing that is more complex than that shown in the example above, you will find it convenient to use separate contexts.

The names of the contexts are defined in the definitions sections, after the definitions of any start conditions: For example:

```
%C name name ...
```

The **lex** function **yyswitch** switches to a new context.

The body of the context's rules is preceded in the rules section by:

```
%C name
```

To see how this works, type the following into file **pre.lex**. It is part of a program that recognizes the preprocessor statements in a C program:

```
%C PRE
%%
^#      yyswitch (PRE);
[^#\n]+        printf ("[%s]", yytext);
%C PRE
include.+      |
define.+       {
               printf("{%s}", yytext);
               yyswitch(0);
               }
.+             {
               printf ("{??%s}", yytext);
               yyswitch (0);
               }
```

A **#** in column 1 signals the beginning of a preprocessor statement. Upon recognizing this condition, this program uses **yyswitch** to activate the context **PRE**.

TUTORIALS

Within this separate context, individual rules recognize different preprocessor statements; this example includes only two. Each of the rules prints the preprocessor line enclosed in braces { }. In addition, the rules switch back to the original (and unnamed) context by the statement

```
yyswitch (0);
```

To compile and test this program, use the following commands:

```
lex pre.lex
cc lex.yy.c -ll -o pre
pre <lex.yy.c | more
```

This example uses the function **yyswitch** to return to the original context at the end of each rule in the secondary context. Some applications require a return to the context that was previously in force. To assist in this, **yyswitch** returns the value of the previous context.

To modify the example to switch to the previous context, add a statement to the definitions section to declare a variable to hold the previous context:

```
int prev;
```

Then, when switching, save the current context:

```
prev = yyswitch (NEW);
```

To switch back, use:

```
yyswitch (prev);
```

To summarize, you can specify a match at the beginning and end of input lines. You may need a following context for a match. Macros provide a means of abbreviating elements of patterns. **lex** can qualify some patterns based on a start context, or process entirely separate contexts.

More About Writing Actions

This section discusses predefined **lex** actions and how to use them. It also presents other **lex** routines that are useful in writing actions.

ECHO

Many **lex** actions simply output the matched pattern:

```
[0-9]+  printf ("%s", yytext);
```

This form has been used in the examples because many examples also output additional material, such as enclosing braces, to illustrate the matched token.

lex provides a simpler way to echo the exact token matched:

```
[0-9]+          ECHO;
```

The following example echoes all strings of digits twice, and everything else once. Type it into file **double.lex**:

```
%%
[0-9]+          {ECHO; ECHO;}
[^0-9]+         ECHO;
```

To compile the program, use the commands:

```
lex double.lex
cc lex.yy.c -ll -o double
```

To invoke the program, type **double**; and to test it, type the following text:

```
abcdef 123 678 as45 67gh
<ctrl-D>
```

double will reply:

```
abcdef 123123 678678 as4545 6767gh
```

Processing Overlapping Strings

The **lex** processing illustrated to this point has been restricted to programs whose rules recognize distinct strings. That is, once any character of a string is matched by a regular expression, it cannot be matched by another.

Some applications require that strings be matched by more than one rule; such multiply-matched strings are called *overlapping strings*. The **lex** action word **REJECT** provides this capability. When **REJECT** appears in a rule, other rules can also match the string. Remember, however, that **lex** programs give precedence to the longest string that matches a regular expression.

The following example determines the number of letter pairs, or *digrams*, in its input. The input is presumed to be lower-case letters. Enter the following into **digram.lex**:

```
        int digram [128] [128];
%%
[a-z][a-z]     {
               digram [yytext [0]] [yytext [1]]++;
               REJECT;
               }
.              ;
\n             ;
%%
yywrap ()
{
    int i1, i2;
    for (i1 = 'a'; i1 <= 'z'; i1++)
        for (i2 = 'a'; i2 <= 'z'; i2++)
            if (digram [i1] [i2] != 0)
               printf ("%d\t%c%c\n",
                   digram [i1] [i2], i1, i2);
}
```

To compile the program, type the commands:

```
lex digram.lex
cc lex.yy.c -ll -o digram
```

To invoke the program, type **digram**; and test it with the following text:

```
this is a test of digrams.
<ctrl-D>
```

The result will be:

```
1      am
1      di
1      es
1      gr
1      hi
1      ig
2      is
1      ms
1      of
1      ra
1      st
1      te
1      th
```

yylex

lex places the actions you provide for the rules in your **lex** program into a C routine named **yylex**.

If you add variable declarations in the definitions section before the first **%%**, **yylex** can access them, as in the example **digram.lex**, shown above. You can also declare variables that are local to **yylex**, if you place the declarations after the rules section delimiter and before the first rule. A tab or space must precede the declaration, where the **%** symbols are at the beginning of the line. See the example **yacclex.lex**, below.

The following program is a different version of **digram.lex**, called **digram2.lex**; it uses such a declaration.

```
        int digram [128] [128];
%%
        int t0, t1;
[a-z][a-z]      {
                t0 = yytext [0];
                t1 = yytext [1];
                digram [t0] [t1]++;
                REJECT;
                }
%%
yywrap ()
{
    int i1, i2;
    for (i1 = 'a'; i1 <= 'z'; i1++)
        for (i2 = 'a'; i2 <= 'z'; i2++)
            if (digram [i1] [i2] != 0)
                printf ("%d\t%c%c\n",
                    digram [i1] [i2], i1, i2);
}
```

Header Section

You can insert additional code at the beginning of the generated program by including such code in the definitions section. An earlier example, **count.lex**, demonstrated how to do this:

```
        int count;
%%
[^ \t\n]+       count++;
[ \t\n]+        ;
%%
yywrap ()
{
    printf ("Number of tokens:%d \n ", count);
    return (1);
}
```

A tab or space character must precede the code you include.

If you wish to insert **include** or any other C preprocessor statement at the beginning of the program, however, a different technique must be used. This stems from the fact that the preprocessor statements must begin at the beginning of the line, and the blank or tab precludes this.

The alternative method to add code to the beginning is as follows:

```
%{
... code ...
%}
```

where the **%** symbols are at the beginning of the line.

Additional Routines

If your version of **yywrap** or any of the rules that you write need other routines, you can include code for them after a second **%%**. (This was where **yywrap** was shown in **digram.lex**.) If you wish to provide your own version of **input** or **output**, you must define it there.

Using lex With yacc

Although **lex** can handle many applications by itself, it is often used with the parser-generator **yacc**. For example, programming-language compilers often have parts generated by both **lex** and **yacc**.

Like **lex**, **yacc** is a program generator. Its programs can recognize input that is structured according to a grammar fed to the **yacc** program generator. In most instances, **yacc**-generated programs require *tokens* as input, instead of individual characters. In the context of a programming language, a token is a variable name or a special character (such as an operator). **lex** is often used with **yacc** because **lex** is especially well suited for partitioning text input into tokens.

A **yacc**-generated program expects a token number as input from the routine **yylex**. **yacc** assigns a unique number, or constant definition, to each unique type of token, and expects **yylex** to return these numbers as input.

For your **lex** program to access these predefined constant definitions for token types, you must include the generated **lex** source in the **yacc** specification.

The following examples process very simple input, to illustrate how to assemble **lex**- and **yacc**-generated programs. To begin, type the following into the file **yacclex.yy**:

```
%token beginning midtok ending
%start simplistic
%%
simplistic :   beginning middle ending
                        {printf ("recognized"); };
middle  :      midtok;
middle  :      middle midtok;
%%
```

When **yacc** processes this program, it produces the file **y.tab.h** that contains the token-name definitions. The following **lex** source reads **y.tab.h** to learn of the constant definitions that **yacc** generated; type it into file **yacclex.lex**:

```
%{
#include "y.tab.h"
%}
%%
"("    return (beginning);
")"    return (ending);
[a-zA-Z]       return (midtok);
```

The symbolic definition of the token names are **beginning**, **ending** and **midtok**.

To compile the programs, type the following commands:

```
yacc yacclex.yy
lex yacclex.lex
cc y.tab.c lex.yy.c -ly -ll -o yacclex
```

Type **yacclex** to invoke the new program; and test by typing the following:

```
(abcdef)
<ctrl-D>
```

The result will be:

```
recognized
```

Summary

lex is a utility that generates lexical analyzers according to a set of specifications that you write. *Lexical analysis* means to read a mass of text, recognize strings within that mass, and react appropriately when each type of string is discovered. With **lex**, you can write programs to perform complex analysis of text simply by describing what analysis you want to perform, without worrying about the messy details of how that analysis is actually performed; thus, **lex** is a fine example of what is nowadays called a "fourth-generation language".

lex is especially well suited to work with the parser-generator **yacc**. By using them together, you can efficiently build command processors and even entire computer languages.

Introduction to yacc

The first high-level programming language compiler took a very long time to write. Since then, much has been learned about how to design languages and how to translate programs written in high-level languages into machine instructions. With what is known today, the writing of a compiler takes a fraction of the time it used to require.

Much of this improvement is due to the use of more powerful software development methods. In addition, we know about the mathematical properties of computer programming languages. Software tools that apply this mathematical knowledge have played a large part in this improvement.

The COHERENT system provides two tools to simplify the generation of compilers. These tools are the lexical analyzer generator **lex** and the parser generator **yacc**. The following introduces **yacc**, and gives a basic course in its use.

Although initially intended for the development of compilers, **lex** and **yacc** have proven their utility in other, simpler, tasks. Examples of very simple languages are included in this tutorial.

yacc accepts a free-form description of a programming language and its associated parsing, and generates a C program that, when compiled, will parse a program written in the described language. It uses a left-to-right, bottom-up technique, to detect errors in the input as soon as theoretically possible. **yacc** generates parsers that handle certain grammatical ambiguities properly.

This manual presumes that you are familiar with computer-language parsing and formal methods of description of languages. Because **yacc** generates its programs in C and uses many of C's syntactic conventions, you should have a working knowledge of C. Related documents include *Using the COHERENT System* and *Introduction to lex*.

Examples

The following presents a few small examples that you can experiment with to get a feel of how to use **yacc**. Feel free to experiment with the examples to investigate new ideas.

Phrases and Parentheses

The first example describes a language we call **slang**, or *simple language*. **slang** consists of sentences. A sentence, in turn, consists of strings of letters or groups of letters enclosed in parentheses, terminated by a period. A group of letters can also include other groups of letters.

The simplest "sentence" in **slang** is:

```
a.
```

The following demonstrates a sentence that consists only of a group:

```
(ab).
```

As described above, a group can have another group inside it:

```
ab(cd(ef)).
```

The following gives the **yacc** grammar for **slang**. Type it into the file **slang.y.** Note that the lexical-analyzer routine **yylex** is included in the **yacc** input file. Note also that, as in C, comments are strings placed between the characters **/*** and ***/**.

```
/* Tokens (terminals) are all caps */
%token LPAREN RPAREN OTHER PERIOD
%%
run     :       sent     /* Input can be a single */
        |       run sent /* sentence or several */
        ;

sent    :       phrase PERIOD
                        {printf ("sentence\n");}
        ;
```

```
group   :       LPAREN phrase RPAREN
                        {printf ("group\n");}
        ;

phrase  :               /* empty */
        |       others
        |       group
        |       others group
        ;

others  :       OTHER   /* letters and other chars */
        |       others OTHER
        ;
%%

#include <stdio.h>
#include <ctype.h>
/* Called by the parser to get a token  */
yylex ()
{
    int c;
    c = 0;

    while (c == 0) {
        c = getchar();
        if (c == '.') return (PERIOD);
        else if (c == '(') return (LPAREN);
        else if (c == ')') return (RPAREN);
        else if (c == EOF) return (EOF);
        else if (! isalpha(c)) c = 0;
    }
    return (OTHER);
}
```

To generate and compile the parser described by this input, issue the commands

```
yacc slang.y
cc y.tab.c -ly -o slang
```

Now, invoke your new parser by typing

```
slang
```

and test it by typing the following input:

```
a
```

does not reply, since this is not a sentence. When you type:

```
a.
```

slang replies:

```
sentence
```

And if you type:

```
abc(def).
```

slang replies:

```
group
```

As you can see, **slang** recognizes groups and sentences within the input you typed, and reactes by printing an appropriate message. Try typing

```
aaa(bbb(ccc)).
(a).
```

and see what you get. To exit from **slang**, type **<ctrl-C>**.

Simple Expression Processing

The next example creates a small language that includes two types of statements. The first type of statement resembles a procedure call, and the second is an expression. Procedure names are in upper-case letters, whereas the variables in expressions are in lower-case letters. Both procedures and expressions are terminated by a semicolon.

The following code generates a parser that identifies either the procedure being called or the arithmetic expression being calculated. The lexical input routine is independently generated by **lex**.

Enter the following program into the file **calc.y**:

```
%token VARIABLE PROCEDURE
%%
prog    :       stmnt
        |       prog stmnt
        ;

stmnt   :       stat
        |       stat '\n'
        |       error '\n'
        ;

stat    :       PROCEDURE ';'
                    {printf ("PROCEDURE is %c\n", $1);}
        |       expr ';'
                    {printf ("Expression\n");}
        ;

expr    :       expr '-' expr
                    {printf
                    ("Subtract %c from %c giving E\n",
                    $3, $1);
                    $$ = 'E';
                    }
        |       VARIABLE
                    {$$ = $1;}
        ;
```

Enter the lexical-analyzer part of the program into the file **calc.lex**:

```
%{
#include "y.tab.h"
%}
%%
[A-Z]           {
                yylval = yytext [0];
                return PROCEDURE;
                }

[a-z]           {
                yylval = yytext [0];
                return VARIABLE;
                }
\n              return ('\n');
.               return (yytext [0]);
```

Now, generate the programs and compile them by typing:

```
yacc calc.y
lex calc.lex
cc y.tab.c lex.yy.c -ly -ll -o calc
```

The following messages will appear on your console:

```
1 S/R conflict
y.tab.c:
lex.yy.c:
```

For now, you can freely ignore the S/R conflict (Shift/Reduce) message from **yacc**. We shall deal with the shift and reduce notions later on. To invoke the newly generated program, type:

```
calc
```

To test it, type the following:

```
A;B;
C;
a-b-c;
a-b-c-d-e;
```

calc will reply appropriately to each line of input. To exit, type **<ctrl-C>**.

Background

Now that you have tried **yacc,** the following gives some background to it, and how the parsers that it generates operate.

LR Parsing

yacc generates a "bottom up" parser. More specifically, **yacc** generates parsers that read LALR(1) languages.

LR parsers scan the input in a left-to-right fashion. Unfortunately, LR parsers for interesting languages are unpractically large. LALR(k) parsers, which are derived from LR parsers, use a "look ahead" technique, in which the next **k** elements of the input stream are used to help determine reductions. LALR(1) parsers are small enough to be practical, are easy to generate, and are fast.

Input Specification

To generate a language with **yacc**, you must specify its grammar in Backus-Naur Form (BNF). (For a good introduction to BNF, see the section on parsing in *Applied C*.) The languages recognized by **yacc**-generated parsers are rich and compare favorably with modern programming languages. The time required to generate the parser from the input grammar is very small — less than the time required to compile the generated parsers.

In addition to generating the parser to recognize the input language, **yacc** lets you include compiler actions within the grammar rules that are executed as the constructs are recognized. This greatly simplifies the entire task of writing your compiler. When used in combination with **lex**, **yacc** can make the process of writing a recognizer for a simple language the task of an afternoon.

Parser Operation

yacc generates a compilable C program that consists of a routine named **yyparse**, and the information about the grammar encoded into tables. Routines in the **yacc** library are also used.

The basic data structure used by the parser is a *stack*, or *push down list*. At any time during the parse, the stack contains information describing the state of the parse. The state of the parse is related to parts of grammar rules already recognized in the input to the parser.

At each step of the parse, the parser can take one of four actions.

The first action is to *shift*. Information about the input symbol or nonterminal is pushed onto the stack, along with the state of the parser.

The second type of action is to *reduce*. This occurs when a grammar rule is completely recognized. Items describing the component parts of the rule are removed from the stack, and the new state is pushed onto the stack. Thus, the stack is *reduced*, and the symbols corresponding to the grammar rule are *reduced* to the left part of the rule.

Third, the parser can execute an *error* action. If the current input symbol is incorrect for the state of the stack, it is not proper for the parser either to shift or reduce. As a minimum, this state will result in an error message being issued, usually

```
Syntax error
```

yacc provides capabilities for using this error state to recover gracefully from errors in the input.

Finally, the parser can *accept* the input. This means that the *start* symbol, such as *program*, has been properly recognized and that the entire input has been accepted.

Later sections discuss how you can have the parser describe its parsing actions step-by-step.

Form of yacc Programs

A **yacc** program can have up to three sections. Each section is marked by the symbol **%%**. The first section contains declarations. The second section contains the rules of the grammar. User-written routines that are to be part of the generated program can be included in the third section. The outline of **yacc** specifications is as follows:

```
definitions
%%
rules
%%
user code
```

If there are no definitions or user code, the input can be abbreviated to

```
%%
rules
```

Definitions

The first section in a **yacc** specification is the definitions section. This section includes information about the elements used in the **yacc** specification. Additional items are user-defined C statements, such as **include** statements, that are referenced by other statements in the generated program.

Each token, such as **VARIABLE** in example program **calc**, must be predefined in a **%token** statement in the definitions section:

```
%token VARIABLE
```

Tokens are also called **terminals**. Only nonterminals appear as the left part of a rule, and terminals can appear only on the right side of a rule. This helps **yacc** distinguish terminals from nonterminals. Other types of statements that assist in ambiguity resolution appear here, and will be discussed in later sections.

Each grammar that **yacc** generates a parser for must have a *start* symbol. Once the start symbol has been recognized by the parser, its input is recognized and accepted. For a programming-language grammar, this nonterminal represents the entire program.

The start symbol should be declared in the definitions section as:

```
%start program
```

If no **%start** symbol is declared, it is taken to be the left side of the first rule in the rules section.

Rules

Your language's grammar rules must be entered in a variant of BNF. The two following rules illustrate how to define an expression:

```
exp     :       VARIABLE;
exp     :       exp '-' exp;
```

Action statements that are enclosed in braces **{ }** specify the semantics of the language, and are embedded within the rules. More information about how rules are built is given below.

User Code

Action statements may require other routines, such as common code-generating routines, or symbol table building routines. Such user code can be included in the generated parser after the rules section and a **%%** delimiter.

The following sections discuss definitions and rules in detail.

Rules

Rules describe how programming-language constructs are put together. Any given language can be described by many configurations of rules. This frees you to write the rules for clarity and readability.

A rule consists of a left part and a right part. The left part is said to *produce* the right part; or, the right part is said to *reduce to* the left part. A rule can also include the action the parser is to perform once it (the rule) is reduced.

General Form of Rules

Blanks and tabs are ignored within rules (except in the action parts). Comments can be enclosed between **/*** and ***/**. The left part of the rule is followed by a colon. Then come the elements of the right part, followed by a semicolon.

Rules that have the same left part can be grouped together with the left part omitted and a vertical bar signifying "or". For example, the grammar

```
exp     :       VARIABLE;
exp     :       exp '-' exp;
```

can be written as:

```
exp     :       VARIABLE
        |       exp '-' exp;
```

Note that these are equivalent to the BNF:

```
<exp>   ::=     VARIABLE
<exp>   ::=     <exp> - <exp>
```

A rule can also contain C statements that are the compiler actions themselves. These actions are enclosed in braces **{** and **}** and are executed by the generated parser when the grammar rule has been recognized. More will be said about actions in the following section.

Suggested Style

yacc permits you to write rules in completely free form. For example, the grammar for the above rule can be written:

```
exp:VARIABLE|exp'-'exp;
```

However, this form is much less readable.

Two styles of **yacc** grammar are in common use. The first of these is used throughout this manual.

First, start the left part at the beginning of the line; follow it with a tab; then a colon. The right part should be on the same line, also preceded by a tab.

Second, group all rules with the same left part together, and use the vertical bar aligned under the colon for all but the first rule in the group.

Third, place action items on a separate line following the associated rule, preceded by three tabs.

Finally, precede the terminating semicolon with a single tab, to align it with the colon and vertical bar.

The outline of this style is:

```
left    :       right1 right2
                        {action1}
        |       right3 right4
                        {action2}
        ;
```

This style is compact and works well for languages whose rules and actions together are simple.

For somewhat more extensive languages, or for additional flexibility in adding statements to the action part, use the following modification of the style.

```
left    :       right1 right2 {
                        action1
                        }
        |       right3 right4 {
                        action2
                        }
        ;
```

For specifications that have larger rules or more complex actions, another style is recommended.

As in the first style, group rules with the same left part, and use the vertical bar. Place the left part, with its terminating colon, on a line by itself. Then indent the right parts of the rule one or more tabs as necessary to make the rule and actions readable. Finally, the vertical bar and the semicolon should be at the beginning of the line.

The outline for this style is as follows:

```
left:
        right1 right2 {
                action1
                }
|       right3 right4 {
                action2
                }
;
```

Since the input to **yacc** can be entirely free form, there is no restriction on how to write your rules. However, if you use a consistent style throughout, it will make your job easier.

Actions

In addition to generating a parser to recognize a specific language, **yacc** also lets you include parsing action statements. With this feature, you can include C-language action statements that will be performed when specified constructs are recognized.

Basic Action Statements

The example language **slang,** described above, the action statements simply print information on the terminal as productions are recognized:

```
sent    :       phrase PERIOD
                        {printf ("sentence\n");}
        ;
group   :       LPAREN phrase RPAREN
                        {printf ("group\n");}
        ;
```

Even if your actions will be more complex, using **printf** statements in this way can help verify your grammar early in the development process.

Action Values

If the specification is for the grammar of a programming language, the actions will normally interface to routines that access symbol tables or generate code.

yacc lets rules assume a *value* to help keep track of intermediate results within rules. These values can contain symbol-table information, code-generation information, or other semantic information.

To set a value for a rule, simply use a statement of the form

```
$$ = <value>;
```

within an action statement. The symbol **$$** is the value of the production. This value can be used by other rules that use this rule as a non-terminal part.

The example program **calc,** given above, illustrates the use of the value of productions:

```
expr    :       expr '-' expr {
                    printf
                        ("Subtract %c from %c giving E\n",
                        $3, $1);
                    $$ = 'E';
                    }
        |       VARIABLE
                        {$$ = $1;}
        ;
```

The first rule's action statement sets the value of the production **expr** to **'E'**:

```
$$ = 'E';
```

The *value* of a rule is significant in that it can be used in productions including that rule as a nonterminal part.

An example is given in the first rule above. The **printf** statement refers to the items **$1** and **$3**. **yacc** interprets these symbols to mean the value of elements one and three of the right side, respectively; that is to say, **$1** refers to the value of the first **expr** in the right side of the first rule, and **$3** refers to the second **expr**, as illustrated below:

```
expr    :       expr '-' expr
                $1   $2   $3
```

calc does not reference **$2**.

The value for the tokens is provided by the lexical analyzer. The second rule for **expr** uses this to get the value of the token **VARIABLE**. The value represented by **$1** is provided by the lexical analyzer in the statement

```
yylval = yytext [0];
```

To give another example, here is a simple calculator language, called **digit,** which performs arithmetic on one-digit numbers and prints the results. Type the following grammar into the file **digit.y**:

```
%token DIGIT
%%
session :       calcn
        |       session calcn
        ;

calcn   :       expr '\n' /* print results */
                        {printf ("%d\n", $1);}
        ;

expr    :       term '+' term
                        {$$ = $1 + $3;}
        |       term '-' term
                        {$$ = $1 - $3;}
        ;

term    :       DIGIT
                        {$$ = $1;}
        ;

%%
#include <stdio.h>
yylex ()
{
    int c;
    c = 0;

    while (c == 0) { /* ignore control chars and space */
        c = getchar();
        if (c <= 0) return (c); /* could be EOF */
        if (c == '\n') return (c); /* set c to ignore */
```

```
            if ((c <= '9') && (c >= '0')) {
                yylval = c - '0';
                return (DIGIT);
            }
            if (c <= ' ') c = 0;
        }
        return (c);
    }
```

This creates the **yacc** specification file. To turn it into a program, type

```
yacc digit.y
cc y.tab.c -ly -o digit
```

To invoke the compiled progra, type:

```
digit
```

And to test it, type the following:

```
1+2
2+2
8+9
```

digit will reply, respectively:

```
3
4
17
```

To exit from **digit**, type **<ctrl-C>**.

digit is essentially an interpreter — results are calculated as numbers are typed in. When you type in

```
1+1
```

the parser recognizes the construct

```
term '+' term
```

and executes the statement that adds two numbers together. The two numbers each in turn came from the construct

```
term    :       DIGIT
```

and the value of the digit came from **yylex**. When the statement **calcn** is recognized, the value is printed as the result. Thus, the calculations are performed at the time that the constructs are recognized. If a compiler were being generated, the actions would likely build some form of intermediate code, or expression tree, as in:

```
expr    :       term '+' term
                        {$$=tree (plus, $1, $3);}
```

Structured Values

All the examples thus far have shown action values as simple **int** types. This is not sufficient for a large interpreter or compiler, because at different points in the language a value can represent a constant values, a pointer to code generation trees, or symbol table information.

To solve this problem, **yacc** allows you to define the values of **$$** and **$n** as a *union* of several types. This is done in the definitions section with the **union** statement. For example, to declare action values as an integer, tree pointer, or a symbol-table pointer, you would use the following code:

```
%union {
    int cval;
    struct tree_t tree;
    struct sytp_t sytp;
}
```

This says that action values can be a constant value **cval**, a code tree pointer **tree**, or a symbol-table pointer **sytp**.

To ensure that the correct types are used in assignments and calculations in actions in the generated C program, each token whose value will be used is declared with the appropriate type:

```
%token <tree> A B
%token <cval> CONST
```

In addition, the rules themselves can have a type declaration, as they also can pass action values. Their type is declared in the **%type** statement:

```
%type <sytp> variable
```

This declares the nonterminal **variable** to reference the **sytp** field of the value union.

The values referenced in the action statements do not need to be qualified (unless they are referencing a field of one of the union elements). **yacc** generates the necessary qualification for the references, based upon the type information provided in the **%type** and **%token** statements.

Keep in mind that productions that do not have explicit actions will default to an action of

```
$$ = $1
```

which might cause a type clash when compiling the generated parser. This is more likely to arise during debugging, when you have defined the types but have not put in the actions.

Handling Ambiguities

The ideal grammar for a language is readable and unambiguous. If the grammar is readable, its users will find it easy to use. If the language is unambiguous, the parser generator will parse the programs correctly. However, many common programming language constructs are ambiguous. Consider the following definition of an **if** statement:

```
statement      :      if_statement
               |      others
if_statement   :      IF cond THEN statement
               |      IF cond THEN statement ELSE statement
```

Consider a program that contains a statement

```
if a > b then if c < d then a = d else b = c;
```

The parser does not know by the grammar specification which **if_statement** the **else** belongs with. At the point of the **else**, the parser could correctly recognize it as part of the first **if** or the second **if**. The indentations illustrate the interpretation of the ambiguity associating the **else** with the first **if**.

```
if a > b then
    if c < d then
        a = d;
else
    b = c;
```

Associating it with the second **if**:

```
if a > b then
        if c < d then
                a = d;
        else
                b = c;
```

One solution to this ambiguity is to modify the language and rewrite the grammar. Some programming languages (including the COHERENT shell) have a closing element to the **if** statement, such as **fi**. The grammar for this approach is:

```
statement      :      if_statement
               |      others
if_statement   :      IF cond THEN statement FI
               |      IF cond THEN statement ELSE statement FI
```

Another ambiguity arises from a grammar for common binary arithmetic expressions. The following sample specifies binary subtraction:

```
exp     :       TERM
        |       exp '-' exp
        ;
```

For the program fragment

```
a - b - c
```

the parser can correctly interpret the expression as

```
(a - b) - c
```

or as

```
a - (b - c)
```

While for the **if** example, the language can be reasonably modified to remove the ambiguity, it is unreasonable in the case of expressions. The grammar can be rewritten for **exp** but it is less convenient.

How yacc Reacts

Because some ambiguities, such as the ones detailed above, are common, **yacc** automatically handles some of them.

The ambiguity exemplified by the **if then else** grammar is called a *shift-reduce* conflict. The parser generator can either choose to shift, meaning to add more elements to the parse stack, or to reduce, meaning to generate the smaller production. In the terms of **if**, the shift would match the **else** with the first **then**. Alternatively, the reduce choice will match the **else** with the latest (rightmost) unmatched **then**.

Unless otherwise specified, **yacc** resolves shift-reduce conflicts in favor of the shift. This means that the **if** ambiguity will be resolved in favor of matching the **else** with the rightmost unmatched **then**. Likewise, the expression

```
a - b - c
```

will be interpreted as

```
a - (b - c)
```

Additional Control

yacc provides tools to help resolve some of these ambiguities. When **yacc** detects shift-reduce conflicts, it consults the precedence and associativity of the rule and the input symbol to make a decision.

For the case of binary operators, you can define the associativity of each of the operators by use of the defining words **%left** and **%right**. These appear in the definition section with **%token**.

The usual interpretation of

```
a - b - c
```

is

```
(a - b) - c
```

which is called *left* associative. However, the shift/reduce conflict inherent in

```
exp '-' exp
```

is resolved in favor of the reduce, or in a right-associative manner:

```
a - (b - c)
```

To signal **yacc** that you want the left-associative interpretation, enter the grammer as:

```
%left '+' '-'
%token TERM
%%
expr    :       TERM
        |       expr '-' expr
        |       expr '+' expr
        ;
```

Some operators, such as assignment, require right associativity. The statement

```
a := b + c
```

is to be interpreted as

```
a := (b + c)
```

The **%right** keyword tells **yacc** that the following terminal is to right associate.

Precedence

Most arithmetic operators are left associative. For example, with the grammar

```
%right =
%left '-' '+' '*' '/'
%%
expr    :       expr '-' expr
        |       expr '*' expr
        |       expr '+' expr
        |       expr '/' expr
        |       expr '=' expr
        ;
```

The expression

```
a = b + c * d - e
```

based on associativity alone will be evaluated

```
a = (((b + c) * d) - e)
```

which is not according to custom. We normally think of * as having higher precedence than **+** or **-**, meaning that it is evaluated before other operators with the same associativity. The evaluation preferred is

```
a = (b + (c * d) - e)
```

To generate a parser with this evaluation, use several lines of **%left**, one line for each level of precedence. Each line containing **%left** describes tokens of the same precedence. The precedence increases with each line. Thus, to get the common notion of arithmetic precedence, use a grammar of

```
%right =
%left '-' '+'
%left '*' '/'
%%
expr    :       expr '-' expr
        |       expr '*' expr
        |       expr '+' expr
        |       expr '/' expr
        |       expr '=' expr
        ;
```

This method of **%left** and **%right** gives tokens a precedence and an associativity. This can eliminate ambiguities where these operators are involved. But what about the precedence of rules or nonterminals?

To specify the precedence of rules, the **%prec** keyword at the end of the rule sets the precedence of the rule to the token following the keyword. To add unary minus to the grammar above, and to give it the precedence of multiply, use **%prec *** at the end of the unary rule.

```
%right =
%left '-' '+' '*' '/'
%%
expr    :       expr '-' expr
        |       expr '*' expr
        |       expr '+' expr
        |       expr '/' expr
        |       expr '=' expr
        |       '-' expr %prec *
        ;
```

If associativity is not specified, **yacc** will report the number of shift/reduce conflicts. When associativity is

specified with **%left**, **%right** or **%nonassoc**, this is considered to reduce the number of conflicts, and thus the number of conflicts reported will not include the count of these.

Error Handling

Parsers generated by **yacc** are designed to parse correct programs. If an input program contains errors, the LALR(1) parser will detect the error as soon as is theoretically possible. The error is identified, and the programmer can correct the error and recompile.

However, in most programming environments, it is unacceptable to stop compiling after the detection of a single error. **yacc** parsers attempt to go on so that the programmer may find as many errors as possible.

When an error is detected, the parser looks for a special token in the input grammar named **error**. If none is found, the parser simply exits after issuing the message

```
Syntax error
```

If the special token **error** is present in the input grammar error recovery is modified. Upon detection of an error, the parser removes items from the stack until **error** is a legal input token and processes any action associated with this rule. **error** is the lookahead token at this point.

Processing is resumed with the token causing the error as the lookahead token. However, the parser attempts to resynchronize by reading and processing three more tokens before resuming normal processing. If any of these three are in error, they are deleted and no error message is given. Three tokens must be read without error before the parser leaves the error state.

A good place to put the **error** token is at a statement level. For example, the **calc.y** example in chapter 2 defines a statement as

```
stmnt   :       stat
        |       stat '\n'
        |       error '\n'
        ;
```

Thus, any error on a line will cause the rest of the line to be ignored.

There is still a chance for trouble, however. If the next line contains an error in the first two tokens, they will be deleted with no error message and parsing will resume somewhere in the middle of the line. To give a truly fresh start at the beginning of the line, the function **yyerrok** will cause the parser to resume normal processing immediately. Thus, an improved grammar is

```
stmnt   :       stat
        |       stat '\n'
        |       error '\n'
                        {yyerrok;}
        ;
```

will cause normal processing to begin with the start of the next line.

Error recovery is a complex issue. This section covers only what the parser can do in recovering from syntax errors. Semantic error recovery, such as retracting emitted code, or correcting symbol table entries, is even more complex, and is not discussed here.

yacc reserves a special token **error** to aid in resynchronizing the parse. After an error is detected, the stack is readjusted, and processing cautiously resumes while three error-free tokens are processed. **yyerrok** will cause normal processing to resume immediately. The token causing the error is retained as the lookahead token unless **YYCLEARIN** is executed.

Summary

yacc is an efficient and easy-to-use program to help automate the input phase of programs that benefit by strict checking of complex input. Such programs include compilers and interactive command language processors.

yacc generates an LALR(1) parser, that implements the grammar specifying the structure of the input. A simple lexical analyser routine can be hand-constructed to fit in among the rules, or you can use the COHERENT command **lex** to generate a lexical analyzer that will fit with the parser.

As the structured input is analyzed and verified, you assign meaning to the input by writing semantic **actions** as part of the grammatical rules describing the structure of the input.

yacc parsers are capable of handling certain *ambiguities*, such as that inherent in typical **if then else** constructs. This simplifies the construction of many common grammars.

yacc provides a few simple tools to aid in error recovery. However, the area of error recovery is complex and must be approached with caution.

Helpful Hints

Until you have mastered **yacc**, the best way to build your program is to do it a piece at a time. For example, if you are writing a Pascal compiler, you might start with the grammar

```
%token PROG BEG END OTHER
program :       PROG tokens BEG END '.'
        ;
tokens  :       OTHER
        |       tokens OTHER
        ;
```

and with a simple lexical analyzer of:

```
PROGRAM         return (PROG);
BEGIN           return (BEG);
END             return (END);
.               return (yytext [0]);
```

With the generated program, you can easily test the grammar by feeding it simple programs. Then add items to both the lexical analyzer and **yacc** grammar. With this approach, you can see the parser working, and if it behaves differently than you expect, you can more easily pinpoint the cause.

If you have difficulty understanding what actions your parser is taking, **yacc** will produce for you a complete description of the generated parser. To use this, you should be familiar with the way LALR(1) parsers work. To get this verbose output, specify the **-v** option on the command line. The result will appear in the file **y.output**.

In addition, you can have the parser give you a token-by-token description of its actions while it does them, by specifying the debug option **-d**. This also generates the file **y.output**, which is helpful in reading the debug output. The debug code is generated when the **-d** option is used, but is not activated unless the **YYDEBUG** identifier is defined. Include some code in the definitions section to activate it:

```
%{
    define YYDEBUG
%}
```

Your parser can turn on and off the debugging at execution time by setting the variable **YYDEBUG**: one for on, zero for off.

A frequent cause of grammar conflicts is the empty statement. You should use it with caution. **yacc** generates empty statements when you specify actions in the middle of a rule rather than at the end; for example:

```
def     :       DEFINE {defstart();}
                        identifier {defid ($2);}
        ;
```

yacc generates an additional rule:

```
$def    :       /* empty */
                        {defstart();}
        ;
def     :       DEFINE $def identifier {defid ($2);}
        ;
```

The resulting empty statement can cause parser conflicts if there are similar rules and the empty statement is not sufficient to distinguish between them.

Where to Go From Here

The Lexicon article for **yacc** summarizes its command syntax and features. The tutorial for **lex**, the COHERENT lexical analyzer, describes how to combine **lex** with **yacc** to build applications simply.

bc Desk Calculator Language

This tutorial introduces **bc**, the calculator language for COHERENT. If you have not used **bc** before, this tutorial will introduce you to its features and functions. If you are familiar with **bc**, you can use it as a reference.

bc is a language that can calculate to high precision. It automatically adjusts the number of digits in a number to represent it correctly. It is like having a powerful calculator at your fingertips.

Entry and Exit

The **bc** calculator for COHERENT is easy to use. Whenever you wish to invoke **bc** all you do is type its name (**bc**), followed by a stroke of the carriage return key. When you are finished using the calculator and wish to exit, just type the word 'quit' or **<ctrl-D>**. **bc** exits and returns control to COHERENT.

Example of Simple Use

bc performs calculations on formulas that you type into it. The formulas are laid out as you would naturally write them. For example, to invoke **bc**, have it add 2+2, and then exit, type:

```
bc
2 + 2
```

bc replies:

```
4
```

Then, leave **bc** by typing:

```
quit
```

bc is an arbitrary precision calculator: the number of digits carried by **bc** depends upon the requirements of the calculation, and is automatically expanded by **bc**. Thus, **bc** will never overflow. The number of digits it carries is limited only by the amount of available computer memory. For example, invoke **bc** and then try this calculation:

```
2^500
```

The circumflex '^' character signifies a superscript; thus, we are asking **bc** to raise 2 to the 500th power. After a moment, **bc** will reply:

```
32733906078961418700131896968275991522166\4
20460430647894832913680961337964046745548\8
32700923259041571508866841275600710092172\5
65458853930533285275893766
```

You have probably already noticed one nice thing about this calculator: you don't have to include a print statement as part of your command, because **bc** automatically prints the results onto your terminal screen. When **bc** sees any expression, like "2+2" or "37-7", it prints the result.

bc provides the common arithmetic operators for add, subtract, multiply, and divide, as illustrated by the following commands:

```
7 + 5
7 - 5
7 * 5
7 / 5
```

bc also provides the remainder operator '**%**'. To get a sense of how it works, type:

```
7 % 5
5 % 7
```

Here, **bc** prints the *remainder* of the first number divided by the second; in the case of the first example, **bc** prints 2, and in the second prints 5. As you saw above, **bc** also includes the exponentiation operator '**^**'.

With **bc**, you can also enter numbers with fractional parts. Type the following to illustrate:

```
9.999 * 9.999
```

bc replies:

```
99.980
```

You can save temporary calculations or repeated constants in *variables*. The following example shows you first how to define variables, and second how to use them:

```
a = 1.1
b = 2.2
a
b
a * b
```

Variable names can be longer than one letter.

The basic calculations in the above examples show only part of what **bc** can do. The following section describes simple statements — the assignment of variables and abbreviations — that allow you to perform complex calculations easily.

Simple Statements

Although you can use **bc** as a simple calculator for manipulating numbers, you can take advantage of its greater power by using *variables*. Variables, as noted above, store parts of calculations or constants that you will use repeatedly in calculations. Variable names are simply "words" that you make up. Here are some examples of possible variable names:

```
a
b
totaltaxesdue
ratio
```

To use variables, simply give them a value, use them in a calculation in place of a number, or print them out.

To see how a variable can save you repetitive typing, and protect you from possible errors, invoke **bc** and type the following:

```
x = 9.999
x
x * x
x = x * x
x
```

The following gives the example with **bc**'s replies *in italics*:

```
x = 9.999
x
9.999
x * x
99.980
x = x * x
x
99.980
```

bc did not reply to the assignment statements **x=9.999** and **x=x*x**. However, it did print the value of **x** when requested, and the results of arithmetic using **x**.

Calculations executed with hand-held calculators, with programming languages like C, or with **bc** often use the following formula:

```
x = x + 1
```

To decrease the likelihood of error, **bc** offers you a shorthand expression for this common phrase:

```
x += 1
```

What it means is, "add one to **x**". Type the following example into **bc** to see how this expression works:

```
x = 1
x * x
x += 1
x * x
x += 1
```

Likewise, **bc** provides an abbreviation for:

```
x = x - 2
```

The form should now be familiar:

```
x –= 2
```

The number to the right of the **-=** or **+=** operator can be replaced with a variable or even another calculation. When you type:

```
i = 4
x = 48
x –= i
x
```

bc replies:

```
44
```

Alternatively, if you type:

```
i = 4
x = 48
x –= i * i
x
```

then **bc** replies:

```
32
```

Similar abbreviations are provided for multiplication, division, remainder, and exponentiation. Here is a summary of this class of operation.

a **+=** 2	Replace *a* with a plus 2
b **+=** *a*	Replace *b* with *b* plus *a*
b **–=** *a*	Replace *b* with *b* minus *a*
c ***=** *b*	Replace *c* with *c* multiplied by *b*
c **/=** *a*	Replace *c* with *c* divided by *a*
c **%=** *b*	Replace *c* with remainder of *c* divided by *b*
d **^=** *3*	Replace *d* with *d* raised to the third power

bc also has an operator that increases a variable by one: '**++**'. When you type:

```
a = 1
++a
```

then **bc** replies:

```
2
```

To use this operator in an expression, combine it with a variable anywhere that a variable would normally be used. For example, entering

```
b = 1
a = 3
b = ++a
a
b
```

yields:

```
4
4
```

The '**++**' operator can also be put after a name. The resulting value in the expression is the value of the name *before* it is incremented. However, after the expression is evaluated, the name will have an incremented value. The

following example shows the use of '**++**' both before and after a name:

```
a = 1
b = 1
a++
++b
a
b
```

bc replies:

```
1
2
2
2
```

Operators are used in this manner:

```
a = 1
b = 2
c = a++ + ++b
```

Similar to '**++**' is '—'. It behaves the same way, except that rather than adding one, it subtracts one.

Numbers with Fractions

Most of the examples presented earlier use whole numbers (integers). However, **bc** can use numbers with fractional parts. This section discusses the use of fractional numbers in **bc** and their precision under different operations.

The Scale of Numbers

The number of digits to the left of the decimal point carried by **bc** depends upon the requirements of the calculation. If you calculate a large number, as in:

```
2^500
```

the result will contain as many digits as needed to express the product.

The number of digits to the right of a decimal point is called the *scale* of the number. Scale depends upon the operation that produces the number of digits, and a variable called **scale** that will be described shortly.

To illustrate simple uses of numbers with fractions, invoke **bc** and then type:

```
a = .01
b = 0.99
a + b
```

bc replies:

```
1.00
```

Addition and Subtraction

bc will dynamically adjust the number of digits in the calculation. It deals similarly with fractional numbers. To the following example

```
a = 0.01
b = 0.001
a + b
```

bc reply:

```
.011
```

In addition and subtraction, the scale of the result is the *larger* of the scales of the two numbers involved. Results are not truncated in addition or subtraction operations.

Scale During Multiplication

Other arithmetic operations act differently with numbers that contain fractions. In the multiplication of two numbers, the scale of the product will at least equal the larger of the scales of the two numbers. For example, the input:

```
1.1 * 1.11
```

results in:

```
1.22
```

Setting the Scale of Results

To increase the number of fractional digits for higher accuracy, **bc** provides the built-in variable **scale**. The following example illustrates the **scale** variable:

```
scale = 3
1.1 * 1.11
```

The result from this example is:

```
1.221
```

Note, however, the scale of the product of a multiplication procedure never exceeds the sum of the scales of the two numbers being multiplied. For example,

```
scale = 10
1.1 * 1.11
```

yields the result:

```
1.221
```

If the variable **scale** is less than the sum of the scales of the numbers being multiplied, then the product will have a scale equal to that of the variable **scale**. For example,

```
scale = 4
1.11 * 2.222
```

yields:

```
2.4664
```

The scales of the operands are 2 and 3. The larger scale is 3, so the result of a multiplication will have a scale of at least 3, no matter what **scale** is set to. Also, the sum of the scales is 5, so the result will never have more than 5 digits to the right of the decimal point. In this example, **scale** has been set to a scale of 4. Therefore, the result has four digits to the right of the decimal point.

Scale for Divisions

For division and remainder, the scale of the result is determined only by the value of the variable **scale**. For example,

```
scale = 13
14 / 13
14 % 13
```

yields:

```
1.0769230769230
.0000000000010
```

For non-whole numbers, as well as for integers, the definition of remainder is chosen so that the relationship

```
dividend = (divisor * quotient) + remainder
```

is true.

Scale From Exponentiation

bc sets the **scale** of a result of exponentiation as if repeated multiplications had been performed. Thus, for

```
5.992 ^ 5
```

the scale is chosen as if you typed:

```
n = 5.992
n * n * n * n * n
```

That is, the default is the scale of the largest (or, in this case, the only) number being multiplied; and scale cannot exceed the sum of the scales of the numbers being multiplied. Thus, the scale of the product in this example has a default setting of 3, and can be reset up to 15.

What Is the Current Scale?

The variable **scale** is just like other variables: you can assign values to it, as above. Because it is like regular variables, you can also use it in operations, as in this example:

```
scale += 1
```

You can also print its value:

```
scale
```

The value of the **scale** variable is zero until you explicitly change it.

The if Statement

The statements shown so far have been either assignment statements, giving a new value to a variable; or an expression, which prints the resulting value. Several other kinds of statements are available. These give you power to write programs that make decisions and perform iterative computations.

Using the if Statement

To see the **if** statement in action, type the following example into **bc**:

```
x = 3
if (x < 5) x
if (x > 5) -x
```

bc replies:

```
3
```

If the input is:

```
x = 6
if (x < 5) x
if (x > 5) -x
<return>
```

bc replies:

```
-6
```

The part of the **if** statement in parentheses, such as **(x > 5)**, determines whether **bc** executes the statement that follows it, such as **-x**. If the expression is false, the following statement is not executed. If the expression is true, the following statement is executed.

Comparisons

The decision expression in an **if** statement is enclosed in parentheses. The decision can be based upon a comparison of two operands, or numbers. The kinds of comparisons that can be done are:

==	First operand equal to second
!=	First operand not equal to second
<=	First operand less than or equal to second
<	First operand less than second
>=	First operand greater than or equal to second
>	First operand greater than second

The **if** statement can include the sorts of the simple statements already shown. You can also include an **if** statement, as well as the **while**, **do**, and **for** statements, which will be discussed below. The following example illustrates the use of an **if** statement within an **if** statement:

```
a = 2
b = 6
if (a >= 2) if (b > a) a + b
<return>
```

bc replies, simply:

```
8
```

Because both of the **if** conditions were true, **bc** proceeded to add **a** and **b**. Note that nested **if** statements must appear on the same line. Therefore,

```
if (a == 3) if (b > a) a + b
```

does not print the result of **a + b** because not both conditions were true. However

```
if (a == 3)
if (b > a) a + b
```

prints the result of **a + b** because **bc** treats **if** statements one by one, and the second **if** statement's condition is true.

Grouped Statements

You can place more than one statement after the expression part of the **if** statement by using grouping braces '{' and '}'. This can be useful if you want to perform several calculations based on the result of an **if** statement comparison. The following example prints the value of **a** and **b** if the value of **b** is less than the value of **a**:

```
a = 1
b = .99
if (a > b) {
    a
    b
}
```

bc replies:

```
1
.99
```

Any statement may be enclosed within the group braces, as the following example shows:

```
a = 1
b = .99
if (a > b) {
    a
    b
    if ((a + b) >= 2) a + b
}
```

Many Statements Per Line

To this point, all of our examples typed each statement on its own line. This includes the group braces '{' and '}', the latter of which must appear on a line by itself. You can, however, place several statements on one line if you separate them with semicolons. If you do this, remember that the semicolon rather than the carriage return separates the statements. For example, if you type:

```
a = 1;b = 2;c = 3
a;b;c
```

bc replies:

```
1
2
3
```

You can use this in combination with the group braces:

```
a = 1;b = 2;c = 3
if ((a + b) >= c) {
    a; b; c; a + b; }
```

The reply from **bc** is:

```
1
2
3
3
```

This example can be compressed even further by putting all of the **if** statement on one line:

```
a = 1;b = 2;c = 3
if ((a + b) >= c) { a; b; c; a + b; }
```

You do not need to follow the '}' with a semicolon.

The while Statement

The **while** statement repeats calculations. This is useful in successive approximation calculations. The following example of the **while** loop prints the numbers one through ten:

```
i = 1
while (i <= 10) {
        i
        i = i + 1
}
```

bc replies:

```
1
2
3
4
5
6
7
8
9
10
```

The statement

```
i = i + 1
```

adds 1 to the variable **i**. The expression

```
(i <= 10)
```

compares **i** with ten. While **i** is less than or equal to ten, the **while** loop executes. When **i** is increased to greater than ten, the loop stops executing.

bc checks the comparison expression for the **while** loop before the loop is entered for the first time. If the comparison fails, the loop is not executed at all; otherwise the processing repeats as long as the comparison is true. For example, the following statements do not print anything:

```
i = 0
while (i > 1) i
quit
```

Abbreviations in the while Statement

If we recall the assignment statements from the previous section, we can shorten the **while** counting-to-ten example to:

```
i = 1
while (i <= 10) {
        i
        i += 1
}
```

The result remains the same — a list of numbers from one to ten.

Another abbreviation of the example uses the '**++**' operator. The variable **i** is incremented, then tested in the **while** expression, which simplifies the entire example to:

```
i = 0
while (++i <= 10) i
```

Before the **while** is executed, **i** is set to zero. Then, the **while** expression increments the value of **i** before it is used or compared, Thus, the first value compared, then printed, is one.

Finally, the example calculation can be shortened to one line. If a variable in **bc** is used before it is initialized, it will have the value of zero. For example:

```
zip
```

prints:

```
0
```

Using this in our counting-to-ten example yields:

```
while (++n <= 10) n
```

The for Statement

for is a statement that controls the execution of other **bc** statements. You should use **for** to write a formula to control the number of times a value is computed.

The previous section demonstrated how to print the numbers one to ten using a **while** statement. The following does the same task with a **for** statement:

```
for (i=1; i <= 10; ++i) i
```

Three Parts of the for Statement

The **for** statement is more complex than the **while** statement; its controlling expressions have three parts.

The first part, shown here in italics

```
for (i=1; i <= 10; ++i) i
```

sets up the initial condition. The second part

```
for (i=1; i <= 10; ++i) i
```

tests whether more iterations should be performed. **bc** performs this test *before* it executes the statements that are subordinate to the **for** statement. If the test fails, no more iterations are performed.

The third part

```
for (i=1; i <= 10; ++i) i
```

is performed at the end of each iteration. In practically every instance, this part of the **for** statement modifies the value of the variable that the second part tests.

Taken together, these statements (1) set **i** to zero; (2) check whether **i** is less than or equal to ten; (3) if **i** proves to be so, prints **i**, and then increases it by one.

The following example of the **for** statement adds the squares of the numbers one through ten, prints each square, and then prints the sum of the squares at the end.

```
sum = 0
for (n=1; n <= 10; ++n) {
    sq = n * n
    sq
    sum += sq
}
sum
```

The result is:

```
1
4
9
16
25
36
49
64
81
100
385
```

Similarities Between the for and while Statements

To illustrate the similarity between the **for** statement and the simpler **while** statement, the following rewrites the above example, substituting the **while** for the **for**:

```
sum = 0
n = 0
while (++n <= 10) {
        sq = n * n
        sq
        sum += sq
}
sum
```

Functions in bc

bc allows you to name routines that you use repeatedly. You can then call them by name without having to retype them; obviously, this can be a great time-saver. These named routines are called *functions*. This section shows you how to define and use functions for your **bc** calculations.

Example of Function Use

The following example defines a function that calculates the area of a circle from its radius.

```
scale = 5
pi = 3.14159
define area (radius) {
        r2 = radius * radius
        return (pi * r2);
}
area (1.00)
area (2.00)
area (56)
```

The results will be:

```
3.14159
12.56636
9852.02624
```

The **define** keyword tells **bc** that you are defining a function. The name of the function follows. Then, in parentheses, come the *parameters* of the function. In this example, the only parameter, or *argument*, of the function is **radius**. Most functions have arguments, but they are not mandatory.

The **return** statement defines the value of the function. In the **area** example, the expression

```
area (1.00)
```

references the function **area**. **bc** then performs the calculation described by your definition of the function **area**. The number

```
1.00
```

is substituted wherever the parameter **radius** is shown.

The statement

```
r2 = radius * radius
```

is then executed, yielding this result:

```
1.00
```

Then, the statement

```
return (pi * r2)
```

calculates the area and returns its value. The statement

```
area (1.00)
```

then has the value calculated in the return statement.

Functions Using Other Functions

Functions in **bc** perform calculations using the same expressions as the rest of the **bc** program. This includes the use of functions. The **area** program can be written using another function, **sq**, to calculate the square of a number:

```
scale = 5
pi = 3.14159
define sq (number) {
        return (number * number)
        }
define area (radius) {
        return (sq (radius) * pi)
        }
area (1.00)
area (2.00)
area (56)
```

Again, the results will be identical:

```
3.14159
12.56636
9852.02624
```

Functions That Call Themselves

Not only can functions call other functions and perform regular calculations; a function can use itself in calculations. An example of this is the Fibonacci calculation:

```
define fib (f) {
        if (f == 0) return (0)
        if (f == 1) return (1)
        if (f > 1) return (fib (f - 1) + fib (f - 2))
}
fib (5)
fib (20)
```

Fibonacci numbers are defined in the following way: Fibonacci number zero is zero; similarly, Fibonacci number one is one. Any other Fibonacci number is defined as the sum of the two previous Fibonacci numbers. Fibonacci numbers are defined only for non-negative integers.

The defined function **fib** follows this definition by returning zero if the number requested is zero and one if the argument is one. If the number is neither of these, then the function calls itself to calculate the previous two numbers of the series and adds them together.

The auto Statement

Many functions that call other functions, including themselves, may require variables that are not changeable by the rest of the program. This is signalled to **bc** by the **auto** statement:

```
auto var1, var2
```

This declares **var1** and **var2** as local to the function that contains them.

To illustrate the use of **auto**, the following **bc** program calculates the factorial of a number:

```
define factorial (number) {
        auto value, i
        value = 1
        for (i = 1; i <= number; ++i) value *= i
        return (value)
}
value = 3
factorial (value)
i = 99
factorial (20)
value
i
```

The result is:

```
6
2432902008176640000
3
99
```

The first number, 6, results from:

```
factorial (value)
```

The second number is from:

```
factorial (20)
```

The last two numbers are from **value** and **i**, and are included to demonstrate that the variables in the function **factorial** appearing in this statement:

```
 auto value, i
```

are separate from the variables of the same name in the rest of the program.

If the function calls itself, as the **fib** example does above, any variable names noted in the **auto** statement are handled separately for each call of the function.

Programs in a File

Because its programs can be quite complex, **bc** lets you keep them in files. This lets you build a library of **bc** programs and functions that can be called up easily.

Using a Program From a File

To illustrate the use of programs stored in a file, type the following example into file **fib.bc** using the editor of your choice. The program defines the function **fib**:

```
define fib (f) {
        if (f == 0) return (0)
        if (f == 1) return (1)
        if (f > 1) return (fib (f - 1) + fib (f - 2))
}
```

To use a **bc** program that has been stored in a file, enter the file name on the **bc** command line, like this:

```
bc fib.bc
```

The function definition will be read in by **bc** and ready for your use. To use the function, simply type the function name with parameters.

So, if you type:

```
bc fib.bc
fib (6)
```

bc will reply:

```
8
```

Using Libraries

You can enter several useful programs in their own files and call them into **bc** at the same time. The following example creates another function that calculates the sum of the squares of integers up to a given number. Use an editor to type the following into a file named **sumsq.bc**:

```
define sumsq (number) {
        auto i, sum
        sum = 0
        for (i = number; i > 0; --i) sum += i ^ 2
        return (sum)
}
```

Now, you can use the **sumsq** function to print the sum of the squares for each number from one to ten:

```
bc sumsq.bc
for (i = 1; i <= 10; ++i) sumsq (i)
```

The result is:

```
1
5
14
30
55
91
140
204
285
385
quit
```

You can use the two functions stored in a file to print the difference between the sum of the squares of numbers, and the Fibonacci number:

```
bc fib.bc sumsq.bc
for (i = 1; i <= 10; ++i) sumsq (i) - fib (i)
quit
```

The result of this questionable computation is:

```
0
4
12
27
50
83
127
183
251
330
```

The bc Library

COHERENT provides an extended library to go with **bc**. It includes the following functions:

atan(*z*) arctangent of *z*
cos(*z*) cosine of *z*
exp(*z*) exponential function of *z*
j(*n*,*z*) *n*th order Bessel function of *z*
ln(*z*) natural logarithm of *z*
pi the value of pi to 100 digits
sin(*z*) sine of *z*

The library is stored in file **/usr/lib/lib.b**. To use the library, invoke the **bc** command with the **-l** option.

To show how the library can be used in your work the following example computes the sine of an angle of one-third radian with scale set to 20:

```
bc -l
scale = 20
sin (1/3)
quit
```

The result is:

```
.32719469679615224418
```

Summary

The Lexicon entry for **bc** summarizes its commands, features, and libraries. It will also refer you to related commands and functions.

Introduction to the m4 Macro Processor

m4 is a macro processor for the COHERENT system. It is a powerful and flexible text processing tool. You can tell it, with a great degree of generality, to search for macro names and replace them with other strings. Macros can also take arguments.

m4 provides a useful front end for programming languages such as fourth-generation lanaguages (4GLs) which commonly have no built-in macro facility. **m4** also has powerful facilities for manipulating files, making decisions conditionally, selecting substrings, and performing arithmetic, so it is useful for processing forms.

The command

m4 [*file* ...]

invokes **m4**. **m4** reads each *file* in the order given on the command line; if no *file* is given, **m4** reads from the standard input. The *file* '-' also indicates the standard input; this allows you to perform interactive input while **m4** is processing files. **m4** reports any *file* that it cannot open, and eliminates it from the input stream.

m4 writes its output to the standard output stream. As with other COHERENT commands, the optional output redirection specification **>***outfile* on the command line redirects the output into *outfile.* To leave **m4**, type **<ctrl-D>**.

Definitions and Syntax

m4 reads text one line at a time from its input stream. When it reads a line of text, it scans the line for a macro that you have defined. A legal macro name is a string of alphanumeric characters (letters, digits, underscore '_'), the first of which is not a digit. **m4** recognizes the macro name only if it is surrounded by nonalphanumeric characters (i.e., spaces or newline characters) on both sides.

When **m4** finds a macro, it removes it from the input stream and replaces it with its definition. It then writes the resulting modified text (called *replacement text*), onto the input stream. **m4** then reads another line from the input stream, and continues processing.

Text that is contained within single quotation marks is quoted (i.e., is contained between a grave mark ` on the left and an apostrophe ' on the right). All other text is *unquoted*. **m4** searches only unquoted text for macros.

A *macro call* can be either a macro or a macro immediately followed by a set of arguments:

```
macroname(arg1, ..., argn)
```

A set of arguments must start with a left parenthesis that follows the macro immediately (i.e., no space can come between the macro and the left parenthesis). The entire argument set must be enclosed by balanced, unquoted parentheses: parentheses may appear within the text of an argument, but they must always come in balanced pairs. A single left or right parenthesis may be passed by quoting it, e.g. `(' or `)'.

Arguments are separated by commas that are neither within apostrophes nor within an inner set of unquoted parentheses. **m4** strips from each argument all leading unquoted spaces, tabs, and newlines. It processes the text of each argument in the same manner that it processes ordinary text; that is, it removes, evaluates, and replaces any recognized macro calls *before* it stores the argument text for possible use within the replacement text. If you wish to pass a macro name or an entire macro call as an argument, it must be quoted. **m4** stores the values of the first nine arguments for possible use in the replacement text. It processes arguments after the ninth, but throws away the results.

m4 does not search quoted text for macros. Instead, it removes the quotation marks and copies the text to the standard output unchanged. Quotes can be nested; that is, quoted text can contain other blocks of quoted text. **m4** removes only the outermost level of quotation marks each time it reads a piece of quoted text. This aids in delaying macro expansion in text until the second (or later) time the text is read by **m4**.

m4 includes numerous predefined macros, which perform various functions. The remainder of this document describes the predefined macros in detail. The Lexicon entry for **m4** summarizes each predefined macro.

Defining Macros

The macro

```
define(`name', `definition')
```

defines a macro *name* and its replacement text *definition*. **m4** replaces every subsequent unquoted occurrence of *name* with *definition*, as described above. For example, the **m4** input

```
define(`her', `COHERENT')
To know, know, know her
Is to love, love, love her ...
```

produces the output

```
To know, know, know COHERENT
Is to love, love, love COHERENT ...
```

name should usually be quoted. If it is not quoted and it is being redefined, **m4** sees its old *definition* as the first argument to **define**, which will not have the intended effect. Similarly, *definition* should be quoted if the macro names that occur in it should not be replaced.

Any legal macro name may be the first argument of a **define**. If you redefine a predefined macro, its original function is lost and cannot be recovered.

As noted above, **m4** recognizes a macro name only if it is surrounded by non-alphanumeric characters. For example,

```
define(`her', `COHERENT')
Coherent software is reliable software.
```

produces the output

```
Coherent software is reliable software.
```

m4 does not recognize the characters **her** in the word **Coherent** as a macro name.

The value of the **define** macro is the null or empty string (the string which contains no characters). In other words, **m4** puts nothing (the null string) back on its input stream when it processes a **define** call.

Like predefined macros, user-defined macros may take arguments. **m4** replaces the string *$n* in the macro definition with the value of the *n*th argument, where *n* is a digit (1 to 9). It replaces $0 with the macro name. If the argument set contains fewer than *n* arguments, **m4** replaces *$n* with the null string. **m4** uses functional notation to specify argument sets. Unlike a normal function, however, an **m4** macro does not require a fixed number of arguments. The same macro may be called with or without an argument set, or with argument sets containing different numbers of arguments.

The following macro concatenates its arguments:

```
define(`cat', $1$2$3$4$5$6$7$8$9)
```

Then

```
cat(one, `two', ``three'', `four, four ',
    five(also,),,seven)
```

becomes

```
onetwothreefour, four five(also,)seven
```

A more complex definition is:

```
define(`comma', ``$0 (which looks like `,')'')
```

This turns each subsequent unquoted occurrence of

```
comma
```

into

```
comma (which looks like `,')
```

Two sets of quotation marks around the replacement text are necessary. When **m4** reads this call to macro **define**, the resultant argument text is:

TUTORIALS

```
comma
```

for the *name* and

```
`$0 (which looks like `,')'
```

for the *definition*. When **m4** sees the text

```
comma that is not quoted
```

it evaluates and replaces the now-defined macro name **comma** to produce the text

```
`comma (which looks like `,')' that is not quoted
```

on the *input* stream. Because **comma** appears inside a set of quotation marks, **m4** does not treat it as a macro name. For the same reason, the string ',' also passes through unmodified. The final output is:

```
comma (which looks like `,') that is not quoted
```

When the predefined macro **dumpdef** is used without arguments, it returns the names and definitions of all defined macros. For each macro, it returns its quoted name, a tab character, and then its quoted definition; no definition is given for a predefined macro. When used with arguments,

```
dumpdef(name)
```

returns the quoted definition of each macro name that appears as an argument.

The predefined macro

```
undefine(`name')
```

removes a macro definition. As noted for **define** above, the argument must be quoted to have the desired effect. **undefine** ignores arguments which are not defined macro names. The value of the **undefine** call is the null string. If a predefined macro is undefined, its original function cannot be recovered.

Input Control

The predefined macro **changequote** changes the quote characters. For example:

```
changequote( {, })
```

makes the quote characters the left and right braces. It also removes the effect of the previously defined quotation characters. Missing arguments default to ` for open quotation and ' for close quotation. Thus, **changequote** without arguments restores the original quote characters ` and '. If the arguments are identical, the nesting ability of quotation marks is temporarily lost. Instead, the first instance of the new quote character turns on quoting and the next instance turns off quoting. The value of the **changequote** call is the null string.

The predefined macro **dnl** (delete to newline) "eats" all characters from the input stream up to and including the next newline and returns the null string. It is particularly useful in a string of **define** macro calls. Although **m4** replaces each **define** by the null string, newlines often separate macro definitions, and **m4** copies the newlines to the output stream unchanged. Two ways of using **dnl** are:

```
define(this, that)dnl
define(something, else)dnl

dnl(define(this, that), define(something, else))
```

The first examples use **dnl** without arguments. The final example uses **dnl** with an argument set, which **m4** processes (performing each **define**) and subsequently ignores. The following section describes an alternative (and generally preferable) method of eliminating extraneous newlines in a sequence of **define** calls.

m4 includes two decision-making macros: **ifdef** and **ifelse**.

ifdef checks whether a macro is defined. It has the following form:

```
ifdef(macro,defvalue,undefvalue)
```

If *macro* is defined, **ifdef** returns *defvalue*; otherwise, it returns *undefvalue*.

ifelse compares pairs of arguments. It has the following form:

```
ifelse(arg1,arg2,arg3, ... , arg9)
```

ifelse compares *arg1* with *arg2*. If they are the same, it returns *arg3*. If not, and if *arg4* is the last argument, it

returns *arg4*. Otherwise, it repeats the process, comparing *arg4* with *arg5*, and so on. Like other **m4** macros, this takes a maximum of nine arguments.

In addition to each *file* specified in the command line, any other accessible file may be included in the input stream with the predefined macro

```
include(file)
```

m4 replaces this macro call on the input stream with the entire contents of the specified *file*. If *file* cannot be accessed, **include** causes a fatal error; **m4** prints an error message and exits. The alternative predefined macro

```
sinclude(file)
```

functions exactly like **include**, except that it does not print an error message and stop processing if *file* is inaccessible.

Output Control

m4 maintains ten output streams, numbered zero through nine. Stream 0 is the standard output, where **m4** normally directs its output. Streams 1 through 9 are temporary files. The predefined macro

```
divert(n)
```

diverts output away from stream 0, appending it instead to stream *n*. Any *n* outside the range 0 to 9 causes output to be thrown away until the next **divert** call. **divert** without any arguments or with a nonnumeric argument is equivalent to **divert(0)**. The value of a **divert** call is the null string.

The preceding section described the use of **dnl** to eliminate extraneous newlines on the output stream when processing a sequence of **define** calls. A more readable method of eliminating the newlines is to precede the definitions with **divert(-1)** and follow them with **divert**. **m4** then diverts the extraneous newlines to the nonexistent stream -1.

The predefined macro

```
undivert(streams)
```

fetches text diverted to one or more temporary streams. It appends the text from the specified *streams* in the given order to the *current* output stream. **m4** does not allow diverted text to be undiverted back to the same stream. **undivert** with no arguments undiverts all diversions in numerical order. The value of **undivert** is the null string; undiverted text is *not* scanned for macro calls, but is simply moved from one place to another. **m4** automatically undiverts all diversions in numerical order to the standard output (stream 0) at the end of processing.

To illustrate the use of **divert** and **undivert**, invoke **m4** and type:

```
define(`count', $1$2)
```

And to see what macro **count** does, type:

```
count(one, three)
```

The output on the screen reads:

```
onetwo
```

Now type:

```
divert(1)
```

This diverts device 1 (the standard output) into a temporary file. Now type:

```
count(one, three)
```

Nothing appears on the screen. **divert** sent the output of the macro **count(one, three)** into a temporary file. Thus, the output is not lost, as you might have thought. To demonstrate the existence of that output, type:

```
divert
```

to reset the standard output to be the screen. See for yourself. Now, when you type

```
count(one, four)
```

m4 replies on the screen:

```
onefour
```

As you can see, the standard output is again directed to the screen. To retrieve the diverted output of **count(one, three)**, and send it to the screen, type:

```
undivert(1)
```

which produces:

```
onethree
```

The predefined macro **divnum** returns the current diversion number.

The predefined macro

```
errprint(message)
```

sends the given *message* to the standard error stream. The value of **errprint** is the null string.

String Manipulation

The predefined macro

```
substr(string, start, count)
```

returns a substring of a string of characters. The first argument *string* can be anything. The second argument *start* is a number giving the starting position of the desired substring in *string*. Position 0 is the leftmost character of *string*, position 1 is the next character to the right, and so on. If *start* is negative, the orientation switches to the right. Position -1 is the rightmost character of *string*, position -2 is the character to its left, and so on. The third argument *count* specifies the length and direction of the substring. Zero returns the null string. A positive *count* returns a substring consisting of the character addressed by *start* and *count*-1 characters to the right of it. A negative number does the same thing, but to the left. If *count* is omitted, it is assumed to be of the same sign as *start* and large enough to extend to the end of *string* in that direction. If *start* is omitted, it is assumed to be 0 if *count* is positive or omitted, or -1 if *count* is negative. For example:

```
define(`alpha', `abcdefghijklmnopqrstuvwxyz')
substr(alpha, , )
```

returns

```
abcdefghijklmnopqrstuvwxyz
```

Here both *start* and *count* are omitted and are therefore assumed to be 0 and 26, respectively.

```
substr(alpha, 0, 6)
substr(alpha, , 6)
```

both return

```
abcdef
```

Similarly,

```
substr(alpha, , -6)
substr(alpha, 21, )
```

both return

```
uvwxyz
```

Finally,

```
substr(alpha, -6, )
substr(alpha, 0, 21)
```

both return

```
abcdefghijklmnopqrstu
```

The predefined macro

```
translit(string, characters, replacements)
```

transliterates single characters within a string. It returns *string* with every occurrence of a character specified in

characters replaced with the corresponding character from *replacements*. If there is no corresponding character, **translit** simply deletes the character. For example:

```
define(liquorjugs, `pack my box with five dozen liquor jugs')
translit(liquorjugs, aeiou, 1234)
```

returns:

```
p1ck my b4x w3th f3v2 d4z2n l3q4r jgs
```

Numeric Manipulation

m4 can simulate the long integer variables typical of most programming languages by using **define** as the assignment operator. Whenever the defined macro name appears unquoted, **m4** immediately replaces it by its numeric value.

The predefined macros **incr** and **decr** return their argument incremented or decremented by 1. Thus,

```
define(`x', 1234)
incr(x)
```

returns:

```
1235
```

Note that **incr** and **decr** do not change the value of the simulated variable **x**, or of any other variable. They return only that value plus or minus 1; **x** itself retains its value of **1234**.

incr and **decr** initialize to zero all arguments that are omitted or not a valid number. Thus, the example

```
 incr(a34/87)
```

returns **1**; but

```
incr(123.67)
```

returns **124**. As you can see, **incr** truncates floating-point numbers. The same applies to a variable that you have **define**d to have a floating-point value.

More generally, the predefined macro

eval(*expression*)

evaluates an integer-value arithmetic *expression* and returns the resulting value. The operators available, in order of decreasing precedence, are:

( )	Parentheses for grouping
+ -	Unary plus, negation
^ **	Exponentiation
* / %	Multiplication, division, modulus
+ -	Addition, subtraction
> < >= <= == !=	Comparisons
!	Logical negation
&& &	Logical and
|| |	Logical or

The comparisons and logical operators return either 0 (false) or 1 (true). **eval** performs all arithmetic in **long** integers. **eval** reports an error if its argument is not a well-formed expression.

The predefined macro

len(*string*)

returns a numeric value corresponding to the length of *string*.

The predefined macro

index(*string*, *pattern*)

returns a numeric value corresponding to the first position where *pattern* appears in *string*. If it does not appear, **index** returns -1. Both *pattern* and *string* may be arbitrary strings of any length.

TUTORIALS

The following example defines a macro **repeat** that repeats its first argument the number of times specified by its second argument.

```
define(`repeat',
     `ifelse(eval($2<=0),1,,`repeat($1,decr($2) )'$1)')
```

The definition is recursive; that is, **repeat** calls itself within its own definition. The entire definition is quoted to defer the evaluation of **ifelse** from when **m4** encounters the definition to when it encounters a **repeat** macro call. Similarly, the recursive **repeat** call is quoted to defer its evaluation within the **ifelse**. **eval** checks if the first argument is less than or equal to 0; if so, it returns 1 (true) and **ifelse** returns the null string. Otherwise, **decr** decrements the count, so each successive recursive call has a smaller second argument, and each call appends a copy of the first argument to the previous result. For example:

```
repeat(`Ho! ',3)
```

produces

```
Ho! Ho! Ho!
```

From this example, you can see that the lowered value of the second argument — generated by the macro **decr**— is "kept in mind" successively. Nevertheless, **decr** and **incr** never change the value of a variable. For example, consider:

```
define(`turns', 10)
```

We now have a variable called **turns** whose value is ten. Typing

```
repeat(`Ho! ', turns)
```

produces:

```
Ho! Ho! Ho! Ho! Ho! Ho! Ho! Ho! Ho! Ho!
```

Within **repeat**, **decr** lowered the current value of the second argument (i.e., **turns**), until it becomes zero. But when we type

```
turns
```

we see:

```
10
```

As you can see, the value of **turns** remained ten, despite that variable's having been used in a **decr** statement.

COHERENT System Interface

The predefined macro

```
maketemp(string)
```

creates a unique file name for a temporary file. *string* is a six-character string that is normally initialized to **XXXXXX**; **maketemp** replaces all of the **X**s with a pattern of six numerals that form a unique file name in the directory where temporary files are being written. It is the same as the C library routine **mktemp**. It returns the null string if its argument is less than six characters long.

The predefined macro

```
syscmd(command)
```

performs the given COHERENT *command* and returns the null string. It is the same as the C library routine **system**.

A common use of **syscmd** is to create a file which **m4** subsequently reads with an **include**. For example, to get the output from the COHERENT **date** command:

```
define(`tempfile', maketemp(/tmp/m4XXXXXX))
define(`get_date',
        `syscmd(date >tempfile)'`include(tempfile)')
```

In subsequent input, **m4** replaces each occurrence of **get_date** with the system date information. The definition of **tempfile** is unquoted, so **m4** executes the **maketemp** call only once (when it processes the **define**), and it creates only one temporary file. On the other hand, the definition of **getdate** is quoted, so **m4** executes **syscmd** and **include** to get the current time and date each time it processes a call to **get_date**. The temporary file should be

removed with

```
syscmd(rm tempfile)
```

at the end of the **m4** program.

The following example is more complex. It defines a macro **save**, which appends a macro definition to a file:

```
define(`save',`syscmd(`cat>>$2 <<\#
define(`$1',`dumpdef(`$1')`)
#
')')
```

The arguments to **define** are the *name*

```
save
```

and the *definition*

```
syscmd(`cat >>$2 <<\#
define(`$1',`dumpdef(`$1')`)
#
')
```

(Note that the body of macro **syscmd** uses the shell operator **<<** to create a "here document". For more information on here documents, see the tutorial *Intorducing sh, the Bourne Shell*.) A typical call of this macro is:

```
save(`sample',`defs.m4')
```

which saves the macro definition of **sample** in a COHERENT file **defs.m4** containing macro definitions. When **m4** processes this call, the argument of **syscmd** becomes

```
cat >>defs.m4 <<\#
define(`sample',
```

followed by the definition of **sample** returned by **dumpdef**, followed by

```
)
#
```

Then **syscmd** executes the COHERENT **cat** command to append the here document delimited by # to the macro definition file **defs.m4**. The leading # delimiter of the here document is quoted with \ to prevent interpretation by the COHERENT shell. Because **save** uses the character # to delimit the here document, it does not work correctly for macro definitions containing #. For example,

```
save(`save',`defs.m4')
```

does not work as expected.

Note that you can only use **save** when you run **m4** interactively — you cannot use it in a script. Furthermore, **save** does not always save a definition literally. For example:

```
save(`tempfile', `defs.m4')
```

saves the **tempfile** definition in **defs.m4** as:

```
define(`tempfile', `/tmp/m400074a') #
```

where, as you can see, the **XXXXXX** has been replaced with a hexadecimal number (which may differ from the one you). Likewise, the definition of **get_date** will look like this:

```
define(`get_date', `syscmd(date >tempfile)include(tempfile)') #
```

To load a saved definition into **m4**, simply type **m4** at the shell's command-line prompt to invoke it interactively; and then type:

```
sinclude(defs.m4)
```

From now on, you can use any definition that you had saved into file **defs.m4**.

Errors

m4 reports all errors to the standard error stream. An error produces a line of the form

```
m4: line: message
```

where *line* is a decimal line number and *message* describes the error. For example, the error message

```
m4:  7:  illegal macro name: ab*c
```

indicates an attempt to **define** a macro with the illegal macro name **ab*c** in line 7 of the input stream.

The following error messages may occur:

```
cannot open file
eval: invalid expression
eval: missing or unknown operator
eval: missing value
illegal macro name: name
out of space
/tmp open error
unexpected EOF
```

The *file* or *name* will be the file name or macro name which caused the error, or **{NULL}** if the required argument is omitted.

m4 does not recognize (and therefore does not report) the most common of **m4** errors, namely invoking recursive macro definitions that never terminate. A simple example is the definition

```
define(`recursive', `recursive')
```

When **m4** subsequently encounters a call of **recursive** in its input stream, it replaces it on the input stream with its definition. Because the definition is another call to **recursive**, **m4** replaces it in turn with its definition; the process never terminates. More complicated examples may involve many macro definitions and may be difficult to discover. If **m4** enters an endless loop, you can terminate it from the keyboard by typing the interrupt character (normally **<ctrl-C>**) or the kill character (normally **<ctrl-\>**). If **m4** enters an endless loop while being run in the background, you can terminate it with the **kill** command.

For More Information

The Lexicon entry for **m4** gives a summary of its functions and options.

The make Programming Discipline

make is a utility that guides the building of complex things out of one or more simpler things. The "complex thing" can be practically any sort of file that you create regularly, such as a report or a program.

Under COHERENT, **make** is most commonly used to control the building of complex C programs; and it is in this context that **make** shows its power most easily. This tutorial introduces the features of **make**, and discusses how to use it to help you build complex C programs easily and efficiently.

How Does make Work?

To understand how **make** works, it is first necessary to understand how a C program is built: how COHERENT takes you from the C source code that you write to the executable program that you can run on your computer.

The file of C source code that you write is called a *source module*. When COHERENT compiles a source module, it uses the C code in the source module, plus the code in the header files that the code calls to produce an *object module*. This object module is *not* executable by itself. To create an *executable file*, the object module generated from your source module must be handed to a linker, which links the code in the object module with the appropriate library routines that the object module calls, and adds the appropriate C runtime startup routine.

For example, consider the following C program, called **hello.c**:

```
main()
{
        printf("Hello, world\n");
}
```

When the C compiler compiles the file that contains C code shown above, it generates an object module called **hello.o**. This object module is not executable because it does not contain the code to execute the function **printf()**; that code is contained in a library. To create an executable program, you must hand **hello.o** to the linker **ld**, which copies the code for **printf** from a library and into your program, adds the appropriate C runtime startup routine, and writes the executable file called **hello**. This third file, **hello**, is what you can execute on your computer.

The term *dependency* describes the relationship of executable file to object module to source module. The executable program *depends* on the object module and one or more libraries. The object module, in turn, depends on the source module and its header files (if any).

A program like **hello** has a simple set of dependencies: the executable file is built from one object module, which in turn is compiled from one source module. If you changed the source module **hello.c**, creating an updated version of **hello** would be easy: you would simply compile **hello.c** to create **hello.o**, which you would link with the library and the runtime startup to create **hello**. COHERENT, in fact, does this for you automatically: all you need to do is type

```
cc hello.c
```

and the C compiler takes care of everything.

On the other hand, the dependencies of a large program can be very complex. For example, the executable file for the MicroEMACS screen editor is built from several dozen object modules, each of which is compiled from a source module plus one or more header files. Updating a program as large as MicroEMACS, even when you change only one source module, can be quite difficult. To rebuild its executable file by hand, you must remember the names of all of the source modules used, compile them, and link them into the executable file. Needless to say, it is very inefficient to recompile several dozen object modules to create an executable when you have changed only one of them.

make automatically rebuilds large programs for you. You prepare a file, called a **makefile**, that describes your program's chain of dependencies. **make** then reads your **makefile**, checks to see which source modules have been updated, recompiles only the ones that have been changed, and then relinks all of the object modules to create a new executable file. **make** both saves you time, because it recompiles only the source modules that have changed, and spares you the drudgery of rebuilding your large program by hand.

Try make

To see how **make** works, try compiling a program called **factor**. It is built from the following files:

```
atod.c
factor.c
makefile
```

All three are kept in directory **/usr/src/sample**. To use them, copy the following files into your current directory. (By the way, first make sure that you do not already have a file named **makefile** in your current directory, or the following commands will overwrite it.)

```
cp /usr/src/sample/atod.c .
cp /usr/src/sample/factor.c .
cp /usr/src/sample/makefile .
```

Now, type **make**. **make** begins by reading **makefile**, which describes all of **factor**'s dependencies. It then uses the **makefile** description to create **factor**. The following appear on your screen:

```
cc -c factor.c
cc -c atod.c
cc -f -o factor factor.o atod.o -lm
```

Each of these messages describes an action that **make** has performed. The first shows that **make** is compiling **factor.c**, the second shows that it is compiling **atod.c**, and the third shows that it is linking the compiled object modules **atod.o** and **factor.o** to create the executable file **factor**.

When **make** has finished, the shell prompt returns. To see how your newly compiled program works, type

```
factor 100
```

factor calculates the prime factors of its argument **100**, and print them on the screen:

```
2 2 5 5
```

To see what happens if you try to re-make your file, type **make** again. **make** will run quietly for a moment, and then exit. **make** checked the dates and times of the object modules and their corresponding source modules and saw that the object modules had a time later than that of the source modules. Because no source module changed, there was no need to recompile an object module or relink the executable file, so **make** quietly exited.

To see what happens when one of the source modules changes, try the following. Use the MicroEMACS screen editor to open the file **factor.c** for editing. Insert the following line into the comments at the top, immediately following the **/***:

```
* This comment is for test purposes only.
```

Now type **<ctrl-Z>** to save the file and exit. Type **make** once again. This time, you will see the following on your screen:

```
cc -O -c factor.c
cc -o factor factor.o atod.o -f -lm
```

Because you altered the source module **factor.c**, its time was later than that of its corresponding object module, **factor.o**. When **make** compared the times of **factor.c** and **factor.o**, it noted that **factor.c** had been altered. It then recompiled **factor.c** and relinked **factor.o** and **atod.o** to re-create the executable file **factor**. **make** did not touch the source module **atod.c** because **atod.c** had not been changed since the last time it was compiled.

As you can see, **make** simplifies the construction of a C program that uses more than one source module.

Essential make

Although **make** is a powerful program, its basic features are easy to master. This section will show you how to construct elementary **makefile**s.

The makefile

When you invoke **make**, it searches the directories named in the environmental variable **PATH** for a file called **makefile** or **Makefile**. (You can tell **make** to read a file other than **makefile** or **Makefile**; see the description of **make**'s **-f** option, below.) As noted earlier, the **makefile** is a text file that describes a program's dependencies. It also describes the type of program you wish to build, and the commands for building it.

A **makefile** has three basic parts.

First, the **makefile** describes the executable file's dependencies. That is, it lists the object modules needed to create the executable file. The name of the executable file is always followed by a colon ':' and then by the names of files from which the target file is generated.

For example, if the program **feud** is built from the object modules **hatfield.o** and **mccoy.o**, you would type:

```
feud:    hatfield.o mccoy.o
```

If the files **hatfield.o** and **mccoy.o** do not exist, **make** knows to create them from the source modules **hatfield.c** and **mccoy.c**.

Second, the **makefile** holds one or more *command* lines. The command line gives the command to compile the program in question. The only difference between a **makefile** command line and an ordinary **cc** command is that a **makefile** command line *must* begin with a space or a tab character.

For example, the **makefile** to generate the program **feud** must contain the following command line:

```
        cc -o feud hatfield.o mccoy.o
```

For a detailed description of the **cc** command and its options, refer to the entry for **cc** in the Lexicon.

Third, the **makefile** lists all of the header files that your program uses. (If you don't know what a header file is, see the entry for **#include** in the Lexicon.) These files are given so that **make** can check if any had been modified since your program was last compiled. For example, if the program **hatfield.c** used the header file **shotgun.h** and **mccoy.c** used the header files **rifle.h** and **pistol.h**, the **makefile** to generate **feud** would include the following lines:

```
hatfield.o: shotgun.h
mccoy.o: rifle.h pistol.h
```

Thus, the entire **makefile** to generate the program **feud** is as follows:

```
feud: hatfield.o mccoy.o
        cc -o feud hatfield.o mccoy.o

hatfield.o: shotgun.h
mccoy.o: rifle.h pistol.h
```

A **makefile** can also contain *macro definitions* and *comments*. These are described below.

Building a Simple makefile

The program **factor** is built from two source modules, **factor.c** and **atod.c**. No header files are used. The **makefile** contains the following two lines:

```
factor: factor.o atod.o
        cc -o factor factor.o atod.o -f -lm
```

The first line describes the dependency for the executable file **factor** by naming the two object modules needed to build it. The second line gives the command needed to build **factor**. The option **-lm** at the end of the command line tells **cc** that this program needs the mathematics library **libm** when the program is linked. No header file dependencies are described because these programs use no special header files.

Comments and Macros

make ignores all lines that begin with a pound sign '#'. This lets you embed comments within a **makefile**, to "document" the file so that whoever reads it will know what it is for. For example, you may wish to include the following comments in your **makefile** for **factor**:

```
# This makefile generates the program "factor".
# "factor" consists of the source modules "factor.c" and
# "atod.c".  It uses the standard mathematics library
# "libm", but it requires no special header files.
# "-f" lets you use printf for floating-point numbers.

        factor: factor.o atod.o
                cc -f -o factor factor.o atod.o -lm
```

Anyone who reads this file will know immediately what it is for by looking at the comments.

make also lets you define macros within your **makefile**. A *macro* is a symbol that represents a string of text. Usually, a macro is defined at the beginning of the **makefile** using a *macro definition statement*. This statement uses the following syntax:

```
SYMBOL = string of text
```

Thereafter, when you use the symbol in your **makefile**, it must begin with a dollar sign '$' and be enclosed within parentheses. (If the macro name is only one character long, the parentheses are not required.) A macro name can use both upper-case and lower-case characters.

Macros eliminate the chore of retyping long strings of file names. For example, with the **makefile** for the program **factor**, you may wish to use a macro to substitute for the names of the object modules out of which it is built. This is done as follows:

```
# This makefile generates the program "factor".
# "factor" consists of the source modules "factor.c" and
# "atod.c".  It uses the standard mathematics library
# "libm", but it requires no special header files.
# "-f" lets you use printf for floating-point numbers.

OBJ = factor.o atod.o
factor: $(OBJ)
        cc -o factor $(OBJ) -f -lm
```

The macro **OBJ** is used in this **makefile**. If you use a macro that has not been defined, **make** substitutes an empty string for it. The use of a macro makes sense when generating large files out of a dozen or more source modules. You avoid retyping the source module names, and potential errors are avoided.

Note that you can define macros in the **makefile**, in the environment, or as a command-line argument. A macro defined as a command-line argument always overrides a definition of the same macro in the environment or in the **makefile**. Normally, a definition in a **makefile** overrides a definition of the same macro name in the environment; however, the **-e** option to **make** forces definitions in the environment to override those in the **makefile**.

Setting the Time

As noted above, **make** checks to see which source modules have been modified before it regenerates your C program. This is done to avoid wasteful recompiling of source modules that have not been updated.

make determines that a source module has been altered by comparing its date against that of the target program. For example, if the object module **factor.o** was generated on March 16, 1992, 10:52:47 A.M., and the source module **factor.c** was modified on March 20, 1992, at 11:19:06 A.M., **make** will know that **factor.c** needs to be recompiled because it is *younger* than **factor.o**.

Building a Large Program

As shown earlier, **make** can ease the task of generating a large program. The following gives a **makefile** that can be used to generate the screen editor MicroEMACS:

```
# makefile for "MicroEMACS"

CFLAGS = -O
LFLAGS = /usr/lib/libterm.a
OBJ=ansi.o basic.o buffer.o display.o file.o \
        fileio.o line.o main.o random.o region.o \
        search.o spawn.o termio.o vt52.o window.o \
        word.o tcap.o

me: $(OBJ)
        cc -o me $(OBJ) $(LFLAGS)

$(OBJ): ed.h
```

Note that this **makefile** has been simplified for the purposes of this tutorial; the actual **makefile** that builds the COHERENT edition of MicroEMACS is considerably more complex.

The first line in the above **makefile** gives commentary that describes the file does. The next five lines define macros that are used on the target and command lines. The first macro will be discussed in the following section. The second macro substitutes for the name of a special library that is needed to create this program. The third macro, which is three lines long, stands for the names of the source modules that produce MicroEMACS. A backslash '\' must be used to tell **make** that the macro's definition extends onto the next line.

The next line names the target file (**me**) and the files used to construct it, here represented by the macro **OBJ**.

Next comes the command line, which gives the compilation to be performed. This line *must* begin with a space or a tab.

The last line lists the header file **ed.h**, which is required by all of the files used to generate MicroEMACS.

Command-Line Options

Although **make** is controlled by your **makefile**, you can also control **make** by using command-line options. These allow you to alter **make**'s activity without editing your **makefile**.

Options must follow the command name on the command line and begin with a hyphen, '-', using the following format. The square brackets merely indicate that you can select any of these options; do *not* type the brackets when you use the **make** command:

```
make [ -deinpqrst ] [ -f filename ]
```

Each option is described below.

-d Debug option: **make** describes all of its decisions. You can use this to debug your **makefile**.

-e Environment option: force definitions in the environment to override those in the **makefile**. For example, if the **makefile** defines

```
foo=makefoo
```

and the environment defines

```
foo=envfoo
```

then **$(foo)** expands to **makefoo** if you use the command **make** but expands to **envfoo** if you use the command **make -e**.

-f *filename*
File option: Tell **make** that its commands are in a file other than **makefile**. For example, the command

```
make -f smith
```

tells **make** to use the file **smith** rather than **makefile**. If you do not use this option, **make** searches the directories named in the environmental variable **PATH**, and then the current directory for a file entitled **makefile** or **Makefile** to execute.

-i Ignore errors: **make** ignores error returns from commands and continues processing. Normally, **make** exits if a command returns an error status.

-n No execution: **make** tests dependencies and modification times but does not execute commands. This option is especially helpful when constructing or debugging a **makefile**.

-p Print: **make** prints all macro definitions and target descriptions.

-q Quit option: Return a zero exit status if the targets are up to date. Do not execute any commands.

-r Rules option: **make** does not use the default macros and commands from **/usr/lib/makemacros** and **/usr/lib/makeactions**. These files will be described below.

-s Silence: **make** does not print each command line as it is executed.

-t Touch: **make** changes the modification time of each executable file and object module to the current time. This suppresses recreation of the executable file, and recompilation of the object modules. Although this option is used typically after a purely cosmetic change to a source module or after adding a definition to a header file, it must be used with great caution.

Other Command Line Features

In addition to the options listed above, you may include other information on your command line.

First, you can define macros on the command line. A macro definition must *follow* all command-line options. Arguments, including spaces, must be surrounded by quotation marks, as spaces are significant to the shell. For example, the command line

```
make -n -f smith "OPT=-DTEST"
```

tells **make** to run in the *no execution* mode, read the file **smith** instead of **makefile**, and define the macro **OPT** to mean **-DTEST**.

The ability to define macros on the command line means that you can create a **makefile** using macros that are not yet defined; this greatly increases **make**'s flexibility and makes it even more helpful in creating and debugging large programs. In the above example, you can define a command line as follows:

```
cc $(OPT) example.c
```

When you define the macro **OPT** on the command line, then the program is compiled using the **-DTEST** option, which defines the preprocessor variable **TEST**.

As noted above, a macro defined on the command line always overrides an identically named macro defined either in the environment or in the **makefile**.

Another command-line feature lets you change the name of the *target file* on the command line. Normally, the target file is the executable file that you wish to create, although, as will be seen, it does not have to be. As will be discussed below, a **makefile** can name more than one target file. **make** normally assumes that the target is the first target file named in **makefile**. However, the command line may name one or more target files at the end of the line, after any options and any macro definitions.

To see how this works, recall the program **factor** described above. **factor** is generated out of the source modules **factor.c** and **atod.c**. The command

```
make atod.o
```

with the **makefile** outlined above would produce the following **cc** command line:

```
cc -c atod.c
```

if the object module **atod.o** did not exist or were outdated. Here, **make** compiles **atod.c** to create the target specified in the **make** command line, that is, **atod.o**, but it does not create **factor**. This feature allows you to apply your **makefile** to only a portion of your program.

The use of special, or *alternative*, target files is discussed below.

Advanced make

This section describes some of **make**'s advanced features. For most of your work, you will not need these features; however, if you create an extremely complex program, you will find them most helpful.

Default Rules

The operation of **make** is governed by a set of *default rules*. These rules were designed to simplify the compilation of a typical program; however, unusual tasks may require that you bypass or alter the default rules.

To begin, **make** uses information from the files **/usr/lib/makemacros** and **/usr/lib/makeactions** to define default macros and compilation commands. **make** uses the commands in **makemacros** and **makeactions** whenever the **makefile** specifies no explicit regeneration commands. The command line option **-r** tells **make** not to use the macros and actions defined in **makemacros** and **makeactions**.

As shown in earlier examples, **make** knows by default to generate the object module **atod.o** from the source module **atod.c** with the command

```
cc -c atod.c
```

The macro **.SUFFIXES** defines the suffixes **make** knows about by default. Its definition in **makemacros** includes both the **.o** and **.c** suffixes.

make's files **makemacros** and **makeactions** use pre-defined macros to increase their scope and flexibility. These are as follows:

$< This stands for the name of the file or files that cause the action of a default rule. For example, if you altered the file **atod.c** and then invoked **make** to rebuild the executable file **factor**, **$<** would then stand for **atod.c**.

$*** This stands for the name of the target of a default rule with its suffix removed. If it had been used in the above example, **$* would have stood for **atod**.

$<** and **$* work *only* with default rules; these macros will not work in a **makefile**.

$? This stands for the names of the files that cause the action and that are younger than the target file.

$@ This stands for the target name.

You can use the macros **$?** and **$@** in a **makefile**. For example, the following rule updates the archive **libx.a** with the objects defined by macro **$(OBJ)** that are out of date:

```
libx.a: $(OBJ)
        ar rv libx.a $?
```

For more information on archives, see the Lexicon entry for the command **ar**.

makemacros also contains default commands that describe how to build additional kinds of files:

- **AS** and **ASFLAGS** call the assembler **as** to assemble **.o** files out of files with the suffix **.s**, which **make** assumes hold assembly language.
- **YACC** and **YFLAGS** call **yacc** to build **.o** or **.c** files from files with the suffix **.y**, which **make** assumes hold **yacc** source code.
- **LEX** and **LFLAGS** call **lex** to build **.o** or **.c** files from files with the suffix **.l**, which **make** assume hold **lex** source code.

You can change the default rules of **make** by changing them in **makeactions** and changing the definition of any of the macros as given in **makemacros**.

Source File Path

If a file is not specified with an absolute path name beginning with '/', **make** first looks for the file in the current directory. If the file is not found in the current directory, **make** searches for it in the list of directories specified by the macro **$(SRCPATH)**. This allows you to compile a program in an object directory separate from the source path.

For example

```
export SRCPATH=/usr/src/local/me
make
```

or alternatively

```
make SRCPATH=/usr/src/local/me
```

builds objects in the current directory as specified by the **makefile** from sources kept in directory **/usr/src/local/me**. To test changes to a program built from several source files, copy only the files you wish to change to the current directory; **make** will use the local sources and find the other sources on the **$(SRCPATH)**.

Note that **$(SRCPATH)** can be a single directory, as in the above example, or a list of directories. In the latter case, each entry in the list must be separated by a colon ':', as described in the Lexicon entry for the function **path()**.

Double-Colon Target Lines

An alternative form of target line simplifies the task of maintaining archives. This form uses the double colon "::" instead of a single colon ':' to separate the name of the target from those of the files on which it depends.

A target name can appear on only one single-colon target line, whereas it can appear on several double-colon target lines. The advantage of using the double-colon target lines is that **make** remakes the target by executing the commands (or its default commands) for the *first* such target line for which the target is older than a file on which it depends.

For example, for the program **factor** described earlier, assume that two versions of the source modules **factor.c** and **atod.c** exist: fBfactora.c plus **atoda.c**, and **factorb.c** plus **atodb.c** The **makefile** would appear as follows:

```
OBJ1 = factora.o atoda.o
OBJ2 = factorb.o atodb.o
```

```
factor:: $(OBJ1)
        cc -c $(OBJ1) -lm

factor:: $(OBJ2)
        cc -c $(OBJ2) -lm
```

This **makefile** tells **make** to do the following: (1) Check if either **factora.o** or **atoda.o** is younger than **factor**. (2) If either one is, regenerate **factor** using this version of these files. (3) If neither **factora.o** nor **atoda.o** is younger than **factor**, then check to see if either **factorb.o** or **atodb.o** is younger than **factor**. (4) If either of them is, then regenerate **factor** using the youngest version of these files.

This technique allows you to maintain multiple versions of source files in the same directory and selectively recompile the most recently updated version without having to edit your **makefile** or otherwise trick the system.

You cannot target a file in both a single-colon and a double-colon target line.

Special Targets

A few target names have special meanings to **make**. The name of each special target begins with '.' and contains upper-case letters.

The target name **.DEFAULT** defines the default commands **make** uses if it cannot find any other way to build a target. The special target **.IGNORE** in a **makefile** has the same effect as the **-i** command line option. Similarly, **.SILENT** has the same effect as the **-s** command line option.

Errors

make prints "*command* exited with status *n*" and exits if an executed *command* returns an error status. However, it ignores the error status and continues processing if the **makefile** command line begins with a hyphen '**-**' or if the **make** command line specifies the **-i** option.

make reports an error status and exits if the user interrupts it. It prints "**can't open** *file*" if it cannot find the specification *file*. It prints "**Target** *file* **is not defined**" or "**Don't know how to make** *target*" if it cannot find an appropriate *file* or commands to generate *target*. Other possible errors include syntax errors in the specification file, macro definition errors, and running out of space. The error messages **make** prints are generally self-explanatory. The section **Error Messages**, at the end of this manual, lists **make's** error messages and describes them briefly.

Exit Status

make normally returns a status of zero if it succeeds, and of one if an error occurs. With the **-q** option (described above), **make** returns zero if all files are up to date and two if they are not up to date.

Alternative Uses

make is a program that helps you construct complex things from a number of simpler things.

make usually is used to build complex C programs: the executable file is made from object modules, which are made from source modules and header files. However, you can also use **make** to build any file that is constructed from one or more source modules. For example, an accountant can use **make** to generate monthly reports from daily inventories: all the accountant has to do is prepare a **makefile** that describes the dependencies (that is, the name of the monthly report he wishes to create and the names of the daily inventories from which it is created), and the command required to generate the monthly report. Thereafter, to recreate the report, all the accountant has to do to generate a monthly report is type **make**.

In another example, the **makefile** can trigger program-maintenance commands. For example, the target name **backup** might define commands to copy source modules to another directory; typing **make backup** saves a copy of the source modules. Similar uses include removing temporary files, building archives, executing test suites, and printing listings. A **makefile** is a convenient place to keep all the commands used to maintain a program.

The following example shows a **makefile** that defines two special target files, **printall** and **printnew**, to be used with the source files for the program **factor**.

```
# This makefile generates the program "factor".
# "factor" consists of the source modules "factor.c" and
# "atod.c".  It uses the standard mathematics library
# libm, but it requires no special header files.

OBJ = factor.o atod.o
SRC = factor.c atod.c

factor: $(OBJ)
        cc -o factor $(OBJ) -lm

# program to print all the updated source modules
# used to generate the program "factor"


printall:
        pr $(SRC) | lpr
        >printnew

printnew: $(SRC)
        pr $? | lpr
        >printnew
```

In this instance, typing the command

```
make printall
```

forces **make** to generate the target **printall** rather than the target **factor**, which is the default as it appears first in the **makefile**. The **pr** and **lpr** commands are then used to print a listing of all files defined by **SRC**. The macro **OBJ** cannot be used with these commands because it would trigger the printing of the object files, which would not be of much use. It also creates an empty file **printnew**. This new file serves only to record the time the listing is printed. This tactic is performed in order to record the time that the listing was last generated so that **make** will know what files have been updated when you next use **printnew**.

Typing the command

```
make printnew
```

forces **make** to generate the target **printnew** rather than the default target **factor**. **printnew** prints only the files named in the macro **SRC** that have changed since any files were last printed.

Where To Go From Here

The Lexicon article on **make** summarizes **make**'s options and features. The source code included with the COHERENT system and with the COHware packages include **makefile**s. Studying them will show you how **make** has been used to control the building of large, real-world applications.

nroff, The Text-Formatting Language

nroff is the COHERENT system's text-formatting language. You write a file that mingles the text you want formatted with commands to control the formatting. **nroff** then uses the commands to format the text, and writes the formatted text onto the standard-output device.

This tutorial describes how to work with **nroff.** It assumes you are familiar with the basic features of the COHERENT system. In particular, you should know what a *command* is, what a *file* is, and how to create and edit a file. If you are not familiar with these concepts, read *Using the COHERENT System* before you read this tutorial.

The Lexicon also contains a number of articles that relate to **nroff**. In particular, you should read the article for **printer**, which describes how you can print text under the COHERENT system.

What is nroff?

nroff is the text processor for COHERENT. A *text processor* is a utility that accepts commands and text, and uses the commands to format the text on a page. The commands may call for simple formatting, such as indenting each new paragraph five spaces, to complex formatting of columns and entire pages.

A file that contains text mixed with **nroff** commands is called a *script*. For example, the following **nroff** script

```
.nr Z 0 5
.nf
I tire of love,
.ti \n+Z
I sometimes tire of rhyme;
.ti \n-Z
But money makes me happy
.ti \n+Z
All the time!
.fi
```

produces the following printed text:

```
I tire of love,
        I sometimes tire of rhyme;
But money makes me happy
        All the time!
```

An **nroff** script allows you to change your output very easily. For example, change the minus sign '**-**' in line 7 of the **nroff** to a plus sign '**+**', and the formatted text suddenly becomes:

```
I tire of love,
        I sometimes tire of rhyme;
                But money makes me happy
                        All the time!
```

As you can see, **nroff** is a powerful and versatile formatter.

In truth, however, **nroff** is both a text formatter and a text formatting *language*. With **nroff**, you can write your own text-formatting commands to handle automatically the unique requirements of whatever formatting you need.

nroff Input and Output

Input is what you give to **nroff**. *Output* is what **nroff** returns to you. If you simply type

```
nroff
```

then **nroff** accepts input from your keyboard, and prints its output on your screen. For example, if you want **nroff** to process the contents of a file named **script.r**, type the command line

```
nroff script.r
```

nroff then takes the file **script.r**, processes it, and in a few moments it displays the formatted text on your screen. Note that the suffix **.r** is used by convention to indicate that a file contains an unprocessed **nroff** script.

You can save **nroff**'s output by *redirecting* it into another file. For example, you can redirect **nroff**'s processed output of the file **script.r** into the file named **target** by using the following command:

```
nroff script.r > target
```

Printing nroff Output

The COHERENT system's implementation of **nroff** currently can be used with any variety of printer. COHERENT, however, fully supports three varieties of printer: Epson-compatible dot-matrix printers, printers that use the Hewlett-Packard Page Control Language (PCL) (including the Hewlett-Packard LaserJet and DeskJet families of printers), and any printer that has implemented the PostScript page-control language. The following descriptions assume that you have plugged your printer into a parallel port on your computer, and have installed COHERENT correctly so that it can access your printer.

To print **nroff** output on an Epson-compatible printer, use the commands **epson** and **lpr**. For example, to print the **nroff** output that you have directed into file **text.out**, use the following command:

```
epson text.out | lpr
```

Or, you can pipe the output of **nroff** directly into **epson**, as follows:

```
nroff -ms text.r | epson | lpr
```

In the above example, **text.r** is your input, and **-ms** invokes the **ms** package of macros.

To print on a printer that uses PCL, use the commands **hp** and **hpr**. For example, to print the file **text.out** on a PCL printer, use the command:

```
hp text.out | hpr -B
```

The option **-B** to **hpr** suppresses the printing of a banner page. If you wish, you can pipe the output of **nroff** directly into **hp**, as follows:

```
nroff -ms text.r | hp | hpr -B
```

To access a printer that uses PostScript, use the command **hpr**, but do not use the command **hp**. Also, you use must the **-p** switch to **nroff**, which tells it to generate PostScript output. For example, the following command processes file **text.r** into PostScript output, and passes that output to a PostScript printer:

```
nroff -p -ms text.r | hpr -B
```

All of the above commands are described in their respective entries in the Lexicon.

You can also print the output of **nroff** through the **lp** spooler. For information on that spooler, see its entry in the Lexicon. For a summary of how the COHERENT system manages printers, see the Lexicon entry for **printer**.

nroff Limitations

Because **nroff** is a text-formatting language rather than a text-formatter *per se,* it makes no assumptions about how you want to lay out your page. It does not automatically leave margins at the top and bottom of pages; it does not automatically number pages; it does not automatically format paragraphs. You must use or create a set of formatting commands, called *macros,* to generate these features. This tutorial will teach you how to write macros that can solve nearly every conceivable formatting problem. As you have seen, too, your copy of COHERENT comes with a set of predefined macros, the **-ms** macro package.

The ms Macro Package

A macro package called **-ms** is included with your copy of **nroff**. It provides macros to format paragraphs, produce headers and footers (the areas at the top and bottom of pages, respectively), and perform most other page-formatting tasks. **-ms** is easy to use. The command

```
nroff -ms
```

tells **nroff** to accept input from your keyboard, process it using the **-ms** macro package, and print the output on your screen. The command

```
nroff -ms script.r
```

tells **nroff** to process **script.r** with the **-ms** package and print the output on your terminal; while the command

```
nroff -ms script.r >target
```

redirects the output of **nroff** into the file **target**; and

```
nroff -ms script.r | lpr
```

prints the output on the line printer.

Working with the **-ms** macro package is a good way to gain confidence in working with **nroff** commands. Soon you will learn the correct way to encode **nroff** commands in your scripts.

Using this Tutorial

The only way to learn about **nroff** is to use it. You should type all the examples in this tutorial into your computer and observe how they work. You should also alter the example and examine how your changes affect what **nroff** produces. Don't hesitate to experiment! You can learn more from analyzing why something unexpected happens than you can from simply copying an example that works as you were told it would.

The first section describes how to use **nroff** with the **-ms** macro package. The second section describes how to perform sophisticated formatting. For most users, this chapter contains all the information they need to know.

The rest of the tutorial describes how **nroff** actually works with the input text to produce its output. This will teach you how to write your own **nroff** macros for your special word processing needs.

The ms Macro Package

As explained above, **nroff** is the text formatter for COHERENT. You give **nroff** a *script* — that is, text interspersed with commands that control its processing; **nroff**, in turn, formats your text in the manner dictated by your commands.

nroff's most outstanding feature is its flexibility: you can control line length, page offset, page length, paragraph format, beginning- and end-of-page format, and every other aspect of formatting a document.

nroff has built into it a set of basic commands, called *primitives*, that are used to control formatting. A basic formatting function might require several primitives. For example, formatting a new paragraph requires one primitive to force the printing of the fragment of a line left at the end of the previous paragraph; another primitive to skip a blank line; and a third primitive to indent the first line of the new paragraph. If you were to type directly into your script all the primitives required to control every feature of your document, formatting would be a very difficult task, and mistakes would be common.

Fortunately, another feature of **nroff** makes it easier for you to prepare input: **nroff** allows you to bundle together a group of primitives and give the bundle its own name. Such a bundle is called a *macro*. Whenever you want all the commands in that bundle to be executed, you simply insert the name of the macro into the text. For example, you might group the primitives needed to format a paragraph, and call that bundle **PP**. Then, instead of retyping the primitives, all you need to do is insert the command **.PP** before the start of a paragraph.

-ms is a package of macros that are ready for you to use. When you include the option **-ms** on the **nroff** command line, **nroff** automatically uses the the macros that have been defined in the **-ms** package. These macros will take care of setting line length and page length, numbering pages, formatting paragraphs, and all other formatting tasks. You do not need to know how **nroff**'s primitives are used in the macros; you only need to know the names of the macros and what they do, so that you can insert them correctly into your text.

Using the **-ms** package is a good way to become accustomed to preparing input for **nroff,** so that the features of the primitives will not seem so alien when you eventually choose to work with them. When you become familiar with **nroff**, you may wish to your own macro packages, to handle the unique requirements of different types of documents. For now, however, you will find that the **-ms** package will get you up and running with **nroff**.

Text and Commands

nroff input includes both *text* and *commands.* The commands control the processing of the text. **nroff** distinguishes between text and commands by looking at the first character of each input line. If that character is a period or an apostrophe, the line is a *command*; otherwise, it is *text*.

Earlier in this tutorial, you used the **-ms** package to format a text file that had already been prepared for you. To become more accustomed to using **nroff,** try entering the following text into a file that can be formatted later. Use a text editor (either **ed** or MicroEMACS) to create a file named **script2.r** that contains the following text. It is important for this exercise that you break up the lines as they are shown here:

```
London.  Michaelmas Term lately over,
and the Lord Chancellor sitting in
Lincoln's Inn Hall.  Implacable November weather.
As much mud in the streets, as if the waters
had but newly retired from the face of the
earth, and it would not be wonderful to meet
a Megalosaurus, forty feet long or so, waddling
like an elephantine lizard up Holborn Hill.
```

Note that this file contains no commands; every line is a text line. Process the file with the command:

```
nroff script2.r | more
```

The output is piped to **more** so that it will not all rush past your screen. **nroff** will process the text, and in a moment you will see the following:

```
London.  Michaelmas Term lately over, and the Lord Chancellor sitting in Lincoln's Inn Hall.
Implacable November weather.  As much mud in the streets, as if the waters had but newly retired
from the face of the earth, and it would not be wonderful to meet a Megalosaurus, forty feet long
or so, waddling like an elephantine lizard up Holborn Hill.
```

When you see this example, the spacing will be different; the spacing for the examples in this tutorial is adjusted to conform to the rest of the tutorial text. Notice that **nroff** automatically adjusts the spacing between words to justify the right margin, even though the input text has a ragged right margin. Each output line contains 65 characters, and each output page contains 66 lines.

Now try processing **script.r** again, this time with the **-ms** macro package. Type

```
nroff -ms script.r | more
```

As you can see, **nroff** again adjusted the spacing to keep a strict right margin. Each line was indented with ten leading spaces, followed by 65 characters of text. The pages output by both the **nroff** command and the **nroff -ms** command both contain 66 lines, but the page built with the **-ms** package left blank lines at the top of the page and printed the page number in a blank space at the bottom of the page. When **nroff** constructs its output, it assumes that your printer prints ten characters per inch (Pica, or 10-pitch spacing) and six lines per inch. Given these assumptions, each page of output from **nroff -ms** fits onto an 8.5 by 11 inch page, with an inch of blank space at the top, at the bottom, and on each side.

As this example shows, **nroff** adjusts the spacing between words to keep a strict right margin. When you type in the text, don't worry about the right margin. You must, however, keep a strict left margin, because when **nroff** encounters a line of text that begins with blank spaces, it breaks the line it was working on and begins a new, indented line.

Also, do not hyphenate words; if you do, **nroff** treats each part as a separate "word" (the first ending with the hyphen character), rather than keeping them joined, as you want.

nroff normally interprets as a command every line that begins with a *period* or an *apostrophe*. However, to include an initial apostrophe or period as a literal part of your document, you must place the characters **\&** before the period or apostrophe.

The remainder of this will show you how to use commands in input text to change the appearance of the output. You can control many aspects of the printed document simply by including the appropriate commands within your text.

Command Names

The name of every **nroff** primitive consists of two lower-case letters. Some commands can also include additional information, or *arguments*. For example, **.sp** is the command to leave vertical space between output lines. The command line

```
.sp
```

leaves one space, whereas

```
.sp 2
```

leaves two spaces. The information that follows the command name on the command line is an argument. Each macro defined in the **-ms** macro package is named with one or two upper-case letters. For example, **.PP** is the name of the macro that begins a new paragraph.

TUTORIALS

Paragraphs

Every time you want to begin a new paragraph, enter the *paragraph* command **.PP;** that is, place the command line **.PP** in the text. To test this macro, enter the following text under the name **script3.r**:

```
.PP
It is a truth universally acknowledged,
that a single man in possession of a good fortune,
must be in want of a wife.
.PP
However little known the feelings or views of such
a man may be on first entering a neighbourhood, the
truth is so well fixed in the minds of the surrounding
families, that he is considered as the rightful
property of some one or the other of their daughters.
```

When you process this text with the command

```
nroff -ms script3.r | more
```

the result resembles the following:

```
            It is a truth universally acknowledged, that a single man in possession of a good
fortune, must be in want of a wife.

            However little known the feelings or views of such a man may be on first entering a
neighbourhood, the truth is so well fixed in the minds of the surrounding families, that he is
considered as the rightful property of some one or the other of their daughters.
```

As the output shows, the **.PP** command inserts a blank line before beginning a new paragraph, and indents the first line of the new paragraph by half an inch.

The **-ms** package also provides another paragraph format: the **.IP** command. This macro creates an *indented paragraph.* The **.PP** macro indents only the first line of each paragraph; however, **.IP** indents every line *except* the first. For example,

```
.IP
This is an indented paragraph.
All the lines are indented by
the same amount.
.PP
This is a normal paragraph.
nroff indents the first line
but does not indent the following lines.
```

gives the output

```
        This is an indented paragraph. All the lines are indented by the same amount.

        This is a normal paragraph.  nroff indents the first line but does not indent the
following lines.
```

Several options are available for the basic **.IP** macro. You can add two *arguments* to it. **nroff** interprets the first argument after the **.IP** as a *tag* to the paragraph, and it interprets the second argument as the amount of indentation you want. For example,

```
.IP A. 8
This is the first line of text.
nroff indents the following lines by the same
amount as the first.
The indent is eight spaces.
The paragraph includes a tag in the indent.
```

produces

```
A.      This is the first line of text. nroff indents the following lines by the same amount as
        the first. The indent is eight spaces. The paragraph includes a tag in the indent.
```

You must make sure the indent leaves enough spaces for the tag. If the tag contains blank spaces, enclose it within quotation marks. To see how this works, enter the following script under the title **script4.r**:

```
.IP "King Lear:" 16
Is man no more than this?
Consider him well.
Thou owest the worm no silk,
the beast no hide,
the sheep no wool,
the cat no perfume...
Unaccommodated man is no more
but such a poor, bare, forked
animal as thou art.
```

When processed with the command

```
nroff -ms script4.r >script4.p
```

you see:

```
King Lear:      Is man no more than this? Consider him well. Thou owest the worm no silk, the
                beast no hide, the sheep no wool, the cat no perfume... Unaccommodated man is no
                more but such a poor, bare, forked animal as thou art.
```

As this example shows, this form of the **.IP** macro can be used to format the script for a play.

If you do not want a tag, but merely wish to set the indentation to something other than the default setting of five spaces, then use a pair of quotation marks with nothing between them for the first field:

```
.IP "" 8
```

If you forget the quotation marks, you will not get what you expect: **nroff** will interpret '8' as a tag and use the normal indentation of five spaces.

Once you set the amount of indentation, the new indentation stays in effect until you change it again. For example, if you format a paragraph with

```
.IP "" 8
```

and follow it with another paragraph that begins with **.IP**, **nroff** will also indent the second paragraph by eight spaces. The indentation will remain in effect until you explicitly change it — for example, by beginning a paragraph with

```
.IP "" 6
```

which resets the indent to six spaces.

Normally, **nroff** measures the paragraph indentation from the left margin. Another variation of **IP** allows you to measure the indentation of a new indented paragraph from the left-hand edge of a previous indented paragraph, thus producing *relative indentation*. To do this, enclose the new paragraph between the macros **RS** and **RE** (for **r**elative indent **s**tart and **r**elative indent **e**nd). Copy the following script into the file **script5.r**:

```
.IP
And it came to pass in an eveningtide,
that David arose from off his bed ...
and from the roof he saw a woman washing
herself; and the woman was very beautiful
to look upon.  And David sent and enquired
after the woman.  And one said,
.RS
.IP
Is not this Bathsheba, the daughter of Eliam,
the wife of Uriah the Hittite?
.RE
.IP
And David sent messengers and took her; and
she came in unto him, and he ...
and she returned unto her house.
```

When processed through **nroff** with the command

```
nroff -ms script5.r >script5.p
```

the output resembles the following:

```
And it came to pass in an eveningtide, that David arose from off his bed ... and from the roof he
saw a woman washing herself; and the woman was very beautiful to look upon.  And David sent and
enquired after the woman.  And one said,

        Is not this Bathsheba, the daughter of Eliam, the wife of Uriah the Hittite?

And David sent messengers and took her; and she came in unto him, and he ... and she returned unto
her house.
```

You can include any number of indented paragraphs between **.RS** and **.RE**. Also, you can specify tags and different indents just as for ordinary indented paragraphs. You can even nest **.RS** and **.RE** pairs inside each other to produce multiple relative indents. Just remember that an **.RS** must always be balanced by an **.RE**. Type the following into the file **script6.r** to see how **nroff** handles nested flashbacks:

```
.IP
In England during World War II, a captain tells the
story of his Free French bomber squadron.
.RS
.IP
In the early days of the war, a French ship picks up
five men adrift in a small boat.  One tells of their
life on Devil's Island.
.RS
.IP
A convict tells others of his past.
.RS
.IP
Publication of anti-Nazi material leads to arrest on
false charges.
.RE
.IP
The convicts escape to help France in the war.
.RE
.IP
When France surrenders, the crew overpowers pro-Vichy
officers and heads for England instead of Marseilles.
.RE
.IP
The captain concludes his story as the bombers return
from a mission.
```

When you process this file with the **-ms** package, the output file **script6.p** should resemble the following:

```
In England during World War II, a captain tells the story of his Free French bomber squadron.

        In the early days of the war, a French ship picks up five men adrift in a small boat.  One
        tells of their life on Devil's Island.

                A convict tells others of his past.

                        Publication of anti-Nazi material leads to arrest on false charges.

                The convicts escape to help France in the war.

        When France surrenders, the crew overpowers pro-Vichy officers and heads for England
        instead of Marseilles.

The captain concludes his story as the bombers return from a mission.
```

As you can see, each **.RE** command peels away the current layer of indentation and moves you into the previous one. To return to an even earlier level, you must input the appropriate number of **.RE** commands before you begin a paragraph.

A third type of paragraph is the *quoted* paragraph. It produces a paragraph that is indented on both on the right side and on the left side, in order to set off a quotation from the surrounding text. To produce such a paragraph, precede it with the **.QS** macro and follow it with the **.QE** macro. To break the quotation into different sections, insert a blank line in the text before each line that you want to begin a new section. For example, type the following example as **script7.r**:

```
Form of Tender of Rescue from Strange Young Gentleman
to Strange Young Lady at a Fire.
.QS
Although through the fiat of a cruel fate, I have been
debarred the gracious privilege of your acquaintance,
permit me, Miss [here insert name, if known], the
inestimable honor of offering you the aid of a true
and loyal arm against the fiery doom which now
o'ershadows you with its crimson wing.  [This form
to be memorized, and practiced in private.]
.QE
Should she accept, the young gentleman should offer
his arm - bowing, and observing "Permit me" -
and so escort her to the fire escape and deposit
her in it.
```

After processing with the **-ms** package, the output file **script7.p** should resemble the following:

```
Form of Tender of Rescue from Strange Young Gentleman to Strange Young Lady at a Fire.

      Although through the fiat of a cruel fate, I have been debarred the gracious
      privilege of your acquaintance, permit me, Miss [here insert name, if known],
      the inestimable honor of offering you the aid of a true and loyal arm against the
      fiery doom which now o'ershadows you with its crimson wing.  [This form to be
      memorized, and practiced in private.]

Should she accept, the young gentleman should offer his arm - bowing, and observing "Permit me" -
and so escort her to the fire escape and deposit her in it.
```

Section Headings

The *section heading* macro **.SH** prints a heading or title. For example:

```
.SH
Section Headings
```

The heading may be more than one line long; consequently, you should follow a section heading with a **.PP** or an **.IP** macro. **nroff** leaves a blank line before the heading and prints the heading flush with the left margin in **boldface** type, as described below in the section on Fonts.

The *numbered heading* macro **.NH** produces consecutively numbered section headings. For example:

```
.NH
Guess What's Coming to Dinner?
.NH
Guess Why I Won't be There?
```

produces

```
1.   Guess What's Coming to Dinner?

2.   Guess Why I Won't Be There?
```

You can number subsection headings by entering a number from two to five to the **.NH** macro. The number indicates the level of section headings; for example, **.NH 2** numbers subsection headings, **.NH 3** numbers sub-subsection headings. For example:

```
.NH
Guess What's Coming to Dinner?
.NH 2
Guess What it Looks Like?
.NH 3
Teeth Like That Might Frighten the Children!
.NH 2
What Does it Eat?
.NH
Guess Why I Won't be There?
```

produces:

```
1.   Guess What's Coming to Dinner?

1.1   Guess What it Looks Like?

1.1.1   Teeth Like That Might Frighten the Children!

1.2   What Does it Eat?

2.   Guess Why I Won't be There?
```

The number on the **.NH** command line is *not* the number that appears before the heading; instead, it controls how many "parts" appear in the number. For example, **.NH 3** produces a three-part number, such as **2.5.3**, whereas **.NH 4** produces a four-part number, such as **7.4.5.2**.

You can reset the entire numbering scheme by using the command **NH 0**; for example,

```
.NH 0
Through The Mandelbrot Set With Rod and Gun
```

produces

```
1.   Through The Mandelbrot Set With Rod and Gun
```

with numbering starting at one.

Title Page

If you want your output to begin with a title page, begin the input with the following.

```
.TL
Title of Document (may be more than one line)
.AU
Name(s) of Author(s) (may be more than one line)
.AI
Institution(s) of Author(s)
.AB
Abstract (line length 5.5 inches)
.AE
```

The **.TL** macro indicates the *title,* the **.AU** macro indicates the *author,* the **.AI** macro indicates the *author's institution,* and the **.AB** macro precedes the *abstract*. The **.AE** macro, for **a**bstract **e**nd, marks the end of the abstract. If you do not want some of these headings to appear, simply omit the relevant macros. Begin the body of the document immediately after the **.AE** macro. The body must begin with a formatting command, such as **.PP** or **.SH.**

Note that the *end abstract* macro **.AE** also prints today's date automatically. To do so, **nroff** reads the date as encoded in COHERENT. Before you use these macros, be sure that you have set the correct date in COHERENT.

To see how these macros work, type the following script into file **script8.r**:

```
.TL
Tickling in the Therapy of
von Muenchausen's Syndrome
.AU
P. R. Sanserif
.AI
The Department of Parapsychology
The University of Southern North Dakota
```

```
at Hoople
.AB
Study of 150 subjects (75 men and 76 women)
indicated that hard tickling may prove beneficial
to patients with von Muenchausen's syndrome.
Applications for a seven-figure grant have been
made to continue research in this area.
.AE
.PP
Due to complications in our experiment, this paper
has now been withdrawn.
```

After processing with the **-ms** macro package, you will see that in the outputfile **script8.p**, **nroff** placed the text on the same page as the title information. You may or may not want this to happen. If you do not, one solution is to insert two additional commands between the **.AE** macro and the body of your text:

```
.PP
.bp
```

Headers and Footers

The *header* macro controls the format of the top of each page. It automatically skips one inch at the top of the page. The *footer* macro controls the format of the bottom of each page. It stops printing text one inch above the bottom of the page, and prints the page number.

It is easy to print either a page header or a page footer. Both the page header and the page footer are three-part titles: **nroff** prints the first part on the left side of the page, the second part in the middle, and the third part on the right side of the page. The parts of the header title are named:

LT: left, top
CT: center, top
RT: right, top

and the parts of the footer title are named:

LF: left, footer
CF: center, footer
RF: right, footer

These parts are called *strings.* A later section of this tutorial describes strings in detail. Normally, these strings are undefined, except for **CF**, which prints the current page number; therefore, the header macro normally prints nothing, and the footer macro prints only the page number in the center of the block of space at the bottom of each page. However, you can set any portion of the header or footer to print what you like. To set the left portion of the header, for example, type the following:

```
.ds LT "Walnuts in History"
```

Note that you do *not* type a period before the **LT**. After you define **LT** in this fashion, **nroff** will print

```
Walnuts in History
```

at the top of each page on the left-hand side. If you want the date to appear on the right-hand side of the header, type:

```
.ds RT "\*(Ds"
```

The string **Ds** is automatically initialized to today's date, as set on your COHERENT system. A later section of this tutorial will present strings in detail. For now, all you need to know is that whenever you want **nroff** to insert today's date into your script automatically, just type the entry ***(Ds**. This entry does *not* have to be at the beginning of a line to work.

Use the same procedure to define the strings in the footer title. If you want something other than the page number to appear in the position allocated to **CF**, use the **.ds** primitive to redefine **CF**. If you want nothing to appear there, type

```
.ds CF ""
```

Wherever you want the current page number to appear in the header or footer, use the symbol '%'. For example, if you want the page number to appear in the upper right-hand corner of each page, type

```
.ds RT "Page %"
```

Be sure to type in all of the macros to define headers and footers *before* you begin to type in your text. Otherwise, your headers and footers will not appear on the first page of the formatted output.

To see how this works, try editing the file **script1.r**. At the top, insert the macro

```
.ds RT "\*(Ds"
```

and reprocess the file using the **-ms** macro package. Each output page should have today's date written in the upper right-hand corner.

Fonts

nroff normally prints ordinary, or "Roman", characters. In addition, **nroff** can print **boldface** and *italic* characters. Each of the three styles of type — Roman, boldface, and italic — is called a *font*, in keeping with typesetting terminology.

nroff prints each boldface and italic character by generating a special three-character output sequence. It prints the boldface character **c**, for example, by printing a 'c', then the backspace character **<ctrl-H>**, and then another 'c'. This sequence emaphasizes 'c' by forcing your printer to print it twice. **nroff** represents an italic character *c* with the underscore character '_', followed by the backspace character **<ctrl-H>,** followed by 'c'.

Because of these special representations, the appearance of **nroff** boldface and italic fonts depends on the device on which you see the output. On your terminal, the **<ctrl-H>** backspaces the cursor, and the third character of each sequence replaces the first; therefore, boldface and italic characters appear the same as Roman characters. On a printer, the appearance depends on the characteristics of the printer. The COHERENT system provides programs to print boldface and italic characters appropriately on certain devices.

The **-ms** macro package includes three commands for easy printing in specific fonts: the *boldface* command **.B**, the *italic* command **.I**, and the *Roman* command **.R**. To print a single word in boldface, do the following:

```
The last word is printed in
.B boldface.
```

Likewise for italics:

```
The last word is printed in
.I italics.
```

These example printed a word in a different font. You can print several words in a different font by enclosing the words within quotation marks on the command line:

```
This sentence ends with
.B "three bold words".
```

You can also switch fonts by using one of the font commands with nothing after it on the command line. For example,

```
.B
This entire sentence is printed in boldface.
.R
```

or

```
.I
This entire sentence is printed in italics.
.R
```

In these examples, the Roman font command **.R** is needed to return to the normal font after completing the boldface or italic text.

On rare occasions, you might want different parts of one word to be in different fonts. You cannot use the **-ms** macros to produce mixed-font words directly. A later section of this tutorial gives additional information about **nroff** fonts. As explained there, the input

```
This manual describes \fBnroff\fR's powerful features.
```

produces the output:

This manual describes **nroff**'s powerful features.

The word **nroff** is boldface but the following apostrophe and 's' are Roman.

Special Characters

A few characters have special meaning to **nroff.** You should be aware of these characters if you want **nroff** to process your text properly.

As mentioned earlier, the period and the apostrophe introduce **nroff** command lines. Each is a special character if it is the *first non-space character* on an input line. If you wish to use a period or an apostrophe at the start of an input line simply as part of your text, you must precede it with a backslash and ampersand "**\&**". For example, the input

```
The footnote command
.DS
\&.FT
.DE
generates footnotes for you automatically.
```

produces the output

```
The footnote command

.FT

generates footnotes for you automatically.
```

Neither the period nor the apostrophe is a special character unless it is the first non-space character on a line.

The most important special character for **nroff** is the backslash '\'. It changes the meaning of the following character or characters. If you simply want a backslash to appear as part of your text, you must follow it with the letter 'e'; that is, use "\e" in your input to have '\' appear in your output. Later sections of this tutorial describe other special uses for backslash.

Footnotes

You can place footnotes between the *footnote start* command **.FS** and the *footnote end* command **.FE,** as in the following example:

```
.FS
*MicroKVETCH Electronic Nag is a
copyrighted trademark of Caveat Emptor
Software, Inc.
.FE
```

You should insert each footnote into your text where the reference to it occurs; **nroff** will see to it that the footnote appears at the bottom of the correct page. Footnotes should be inserted as follows:

```
The notion that we have been visited
by visitors from outer space may seem
outlandish(1)
.FS
1. Raucus J, O'Hooligan R: "Viruses
from Venus?" \fIJ Earth Med Assoc\fR,
1985;36:412-414.
.FE
but reason compels us to exclude no ...
```

The journal article cited in the footnote will appear at the bottom of the page, with the journal name in italics.

Displays and Keeps

A *display* is a portion of text, such as a graph or a table, that should appear in the output exactly as it is typed in the input. **nroff** normally alters the spacings between elements in your text, which, of course, would destroy the appearance of a display. Therefore, **nroff** has macros to tell it that a portion of text is a display, and so not to alter spacings between elements or split it between two pages. These macros are the *display start* macro **.DS** and the *display end* macro **.DE**. You should place your display between these macros, as follows:

```
.DS
The text of the display goes here,
exactly
as
you
want
it
to appear in the output.
.DE
```

The **.DS** macro comes in three varieties. The *display start centered* macro **.DS C** centers every line of your display. Because **nroff** centers each line individually, both right and left margins are ragged. The *display start block-centered* macro **.DS B** takes the entire display at once and centers it. You can think of this as simply shifting the display to the right by an appropriate amount. The *display start indented* macro **.DS I** indents the entire display by half an inch.

If your display is longer than one page, do not use **.DS** or any of its variants. Instead, begin the display with one of the following.

The *centered display* macro **.CD** centers each line of the display. The *block-centered display* macro **.BD** considers the entire display as a block and centers it. The *left display* macro **.LD** performs no indenting or centering, but simply begins each line at the left margin. Finally, the *indented display* macro **.ID** indents each line by half an inch. If you begin the display with one of these macros, do *not* end it with **.DE**; rather, just type **.PP** or **.SH** or whatever other macro is needed at that point.

To see how displays work, type the following into the file **script9.r** and process it with the **-ms** macro package:

```
.PP
.DS C
Tyger! Tyger! burning bright
In the forests of the night,
What immortal hand or eye
Could frame thy fearful symmetry?
Burma Shave
.DE
```

When the output file **script9.p** is read, the results will appear as follows:

```
Tyger! Tyger! burning bright
In the forests of the night,
What immortal hand or eye
Could frame thy fearful symmetry?
Burma Shave
```

You must remember one important fact when you use display macros: the normal length of output lines is 6.5 inches, but if the display contains lines longer than this **nroff** simply prints them as they are. If a line is too long to fit onto the page, what occurs afterwards depends upon the output device. If you are displaying the output on the screen, the text will be displayed as far as possible to the right, then the remainder will be wrapped around onto the next line, without indentation. On most printers, however, the chunk of text that extends past the right margin will simply be lopped off and thrown away. In any event, the effect is usually quite unsightly. The only restriction on what you can safely put in a display, then, is that lines should be no longer than 6.5 inches. If you are using an indented display, lines should be no longer than six inches.

A *keep* is a display macro: you put text between the *keep start* macro **.KS** and the *keep end* macro **.KE** when you want it all kept on the same page. If you put a block of text between these macros that proves to be longer than one page, **nroff** moves the excess text onto a new page.

The major difference between the keep and the display is that normal processing occurs in the keep: **nroff** adjusts spacings between words, hyphenates words, justifies lines, and performs all other formatting tasks, just as it normally does.

Other Commands

Several of **nroff**'s primitives can be used with the **-ms** macro package. The primitive

```
.sp N
```

skips *N* lines on the output page; for example, **.sp 4** skips four lines.

The *begin page* primitive **.bp** tells **nroff** to begin a new page, no matter where it is on the current page.

The remaining sections of this tutorial provide more information about these other **nroff** primitives.

Introducing nroff's Primitives

The rest of this tutorial describe **nroff**'s *basic commands* — the commands that are "built into" **nroff**, and from which macros are assembled. These basic commands, or *primitives*, form **nroff**'s text formatting language. Once you have mastered the primitives, you will be able to write macros to control automatically even the most difficult text formatting tasks.

The rest of this tutorial includes a number of exercises. You should type them into your system and execute them as described in the tutorial; this will greatly increase the rate at which you master **nroff**. None of the following examples should be processed with the **-ms** macro package; the purpose of this portion of the tutorial is to teach you how to create you own text processing routines, rather than how to use ones that have already been written.

Page Format

When deciding how to process text, you must first decide how to position the text on the printed page. You must control line length, left and right margins, page offset (i.e., how far from the left edge of the page each line begins), and page length. Controlling these functions is quite easy with the appropriate **nroff** commands.

The *line length* primitive **.ll** controls the line length; and the *page offset* command **.po** controls the page offset. If you are writing an **nroff** script, you should include these commands before the beginning of your text, so that **nroff** can put them into effect immediately. The following example uses a line length of three inches and a page offset of two inches. Type this into your system under the name **ex1.r**. Note, by the way, that the text to the right of the characters '\"' is a *comment*, and there is no need for you to type it into your system:

```
.ll 3i          \" set line length
.po 2i          \" set page offset
Along outside of the front fence ran the country
road, dusty in the summertime, and a good place for
snakes -- they liked to lie in it and sun themselves;
when they were rattlesnakes or puff adders, we killed
them; when they were black snakes, or racers, or belonged
to the fabled "hoop" breed, we fled, without shame; when
they were "house snakes", or "garters", we carried them
home and put them in Aunt Patsy's work basket for a
surprise; for she was prejudiced against snakes, and
always when she took the basket in her lap and they
began to climb out of it it disordered her mind.
```

Process this script by typing the command

```
nroff ex1.r >ex1.p
```

From this point on, you should *not* use the **-ms** macro package with your **nroff** examples. When you display the output stored in the file **ex1.p**, you will see that the length of each line is three inches, and each line begins two inches from the left-hand margin.

As you noticed, line length and page offset were set in *inches*. **nroff** output can be controlled using a number of different units of measurements, including inches, number of characters, or lines, or *machine units*. A following section discusses **nroff** units of measurement in detail.

As noted above, this example contains two *comments*. **nroff** ignores any text that appears on a line after "\"". You should use comments, for the benefit of anyone who must read your **nroff** script (including yourself). The above example used the comments

```
\" set line length
\" set page offset
```

to help you understand the **.ll** and **.po** commands. Judicious comments can make a complex script much easier to understand.

Breaks

Before you look at the *break* primitive **.br**, it is helpful to examine how **nroff** constructs a finished line of output. Suppose, for example, that you tell **nroff** that you want each output line to be five inches long. **nroff** takes your input one word at a time, and attempts to squeeze that word into the space that has not yet been taken up in the line. When **nroff** finally picks up a word that is too large to fit into the amount of space left in the line, it either puts the word aside entirely, or hyphenates the word and places the hyphenated portion into the line. **nroff** then inserts extra blank spaces between the words to justify the line. The *break* primitive **.br**, however, tells **nroff** to print whatever words have already been put into the line, even if they do not form a complete line, and without performing right justification.

The idea of a break might seem strange at first, but you are familiar with a simple example: the end of a paragraph. You do not want the start of a new paragraph to be on the same line as the end of the previous paragraph: you want to print the end of the previous paragraph whether or not it fills a complete line; and you want to begin the new paragraph on a new line. As you will learn later, some **nroff** commands cause breaks automatically; you should be aware of this when you use them.

Fill and Adjust Modes

Two terms describe how **nroff** processes your input to create its output: *filling*, and *adjusting* or *justifying*. Unless you order it not to, **nroff** operates in the *fill* and *adjust* modes. The *no-fill* primitive **.nf** tells **nroff** to stop using fill mode. The *fill* primitive **.fi** tells it to resume using the fill mode. In a similar way, the *adjust* primitive **.ad** tells **nroff** to use adjust mode, whereas the *no adjust* primitive **.na** tells it to use no-adjust mode.

As mentioned above, **nroff** by default is in both fill mode and adjust mode, so you do not need to begin your script with **.fi** and **.ad** if you want **nroff** to fill and adjust your text. However, if you turn off filling and adjusting by using the **.nf** and **.na** commands, you must use the **.fi** and **.ad** commands to turn filling and adjusting back on.

When you use **.nf** to turn off fill mode, **nroff** no longer tries to fill lines to a fixed line length. It prints each line of input text exactly as received. However, a sufficiently long line of text would run off the right-hand edge of the page if **nroff** were to print it as entered. If the input line cannot fit on one line, **nroff** prints as much as it can fit on one line, then breaks the line and prints the rest on the next line with no page offset.

In adjust mode, **nroff** inserts extra spaces between words to justify lines of text, as described above. When **nroff** is in no-fill mode, it is automatically in no-adjust mode: with no fixed line length, there is no need to insert extra spaces. *Moral:* you can fill without adjusting, but you cannot adjust without filling.

If you request filling but not adjusting, **nroff** fills the output line as described earlier, but does not insert extra spaces between words; that is, it does not try to keep an even right margin. Every output line either is shorter than the line length you specified, or exactly as long.

The **.ad** primitive includes several options. If you use the command **.ad** without an argument, **nroff** keeps strict left and right margins. The primitive **.ad l** justifies the left margin only; **.ad r** justifies the right margin only; and **.ad b** justifies both margins (this, of course, is the default). Finally, **.ad c** centers output lines while keeping their lengths less than or equal to the line length, as set with the **.ll** command.

Remember that **nroff** ignores adjustment requests if you are in no-fill mode. If **nroff** is in fill mode and you request any variety of adjustment, it adjusts accordingly until you issue either a no-fill or a no-adjust command. If you give a no-fill command, only a fill command restores adjustment; any plea for a different kind of adjustment is ignored while **nroff** is in no-fill mode.

To see how this works, type the following script under the name **ex2.r**, and process it as above:

```
.ll 6.75i
.sp            \" space
When we were alone, I introduced the subject
of death, and endeavored to maintain that the fear
of it might be got over.  I told [Johnson] that
David Hume said to me, he was no more uneasy to
think that he should not be after this life, than
that he had not been before he began to exist.
.sp
.na            \"no adjust
JOHNSON:  "Sir, if he really thinks so,
his perceptions are disturbed;
he is mad: if he does not think so, he
```

```
lies .... When he dies, he at
least gives up all he has."
.sp
.ad r          \"right-adjust
BOSWELL:  "Foote, sir, told me that
he was not afraid to die."
.sp
.nf            \"no-fill
JOHNSON:  "It is not true, sir.
Hold a pistol to Foote's
breast or to Hume's breast,
and threaten to kill them,
and you'll see how they behave."
.sp
.fi            \"fill
BOSWELL:  "But may we not fortify our minds for
the approach of death?"
.sp
JOHNSON:  "No, sir, let it alone.  It matters not
how a man
dies, but how he lives.  The act of dying is not of
importance, it lasts so short a time .... A man
knows it must be so, and submits.
It will do him no good to whine."
```

When you process this input with **nroff,** your output should look like this:

```
When we were alone, I introduced the subject of death, and endeavored to maintain that the fear
of it might be got over.  I told [Johnson] that David Hume said to me, he was no more uneasy to
think that he should not be after this life, than that he had not been before he began to exist.

JOHNSON: "Sir, if he really thinks so, his perceptions are disturbed; he is mad: if he does not
think so, he lies .... When he dies, he at least gives up all he has."

BOSWELL: "Foote, sir, told me that he was not afraid to die."

JOHNSON:  "It is not true, sir.
Hold a pistol to Foote's
breast or to Hume's breast,
and threaten to kill them,
and you'll see how they behave."

BOSWELL: "But may we not fortify our minds for the approach of death?"

     JOHNSON: "No, sir, let it alone.  It matters not how a man dies, but how he lives.  The act of
  dying is not of importance, it lasts so short a time .... A man knows it must be so, and submits.
It will do him no good to whine."
```

After the **.na** primitive, **nroff** fills but does not adjust the second paragraph. After **.ad r**, it fills and right adjusts the third paragraph. After **.nf**, it neither fills nor adjusts the fourth paragraphs. Finally, after **.fi**, it fills the fifth and sixth paragraphs and uses the **.ad r** adjust option that was in effect previously.

Under certain extreme conditions, **nroff** cannot adjust a line even though it is in adjust mode. If, for example, you specified a line length of one inch, a seven-letter or eight-letter word would then take up most of a line. When such a word was then followed by a word that could not fit into the line after it, **nroff** would begin a new line with the second word rather than violate the right margin by inserting the into the line. When a line has only one word in it, **nroff** obviously cannot adjust the line by inserting extra spaces between words; therefore, the right margin is left uneven, as though **nroff** were in no-adjust mode.

Defining Paragraphs

What happens if you copy text from several pages of a book into a file without adding any formatting commands, and then process the file with **nroff**? There is no page offset, because **nroff**'s default page-offset setting is zero; and the processed lines are set to the default length of 6.5 inches (65 Pica characters).

More interesting things happen with paragraphs. Suppose you skip a line between paragraphs and begin each paragraph by indenting five spaces. The blank line in the input text causes a break, and forces **nroff** to print a blank line. The last line of each paragraph is unadjusted, and a blank line appears before the next paragraph. Initial blank spaces in a line of input also cause a break. In this example, the breaks caused by initial blank spaces at the beginning of each paragraph do nothing, because the preceding blank line forces out the last line of the preceding paragraph. **nroff** always considers initial blank spaces in a line to be significant, and preserves them in the output.

To see how blank lines and initial spaces affect **nroff**'s output, copy the following example and run it through **nroff:**

```
     Here is a little text so you can see
whether nroff will ignore the initial
indentation
            in this very very long sentence.
Here is a little bit more text.

     And here is something to mimic
the beginning of a new paragraph.
```

The output should look like this:

```
Here is a little text so you can see whether nroff will ignore the initial indentation
   in this very very long sentence.  Here is a little bit more text.

And here is something to mimic the beginning of a new paragraph.
```

Instead of leaving a blank line in the text, you could use the *space* primitive **.sp 1,** which causes a break and inserts one blank line into the output. In a similar way, **.sp 5** causes a break and inserts five blank lines in the output. Edit the example and replace the blank line with the command line:

```
.sp 1
```

You will see that it has the same effect. You can also use the form **.sp**; **nroff** assumes you want one space if you omit the argument.

Most **nroff** input consists of many paragraphs that contain text, and you probably want each paragraph to have the same format in the output. Rather than formatting each paragraph explicitly, as in this example, you can use the *macro* facility of **nroff** to define a sequence of commands to format a paragraph. Macros are described in detail later in this tutorial.

Centering

The *center* primitive **.ce** centers one or more lines of text. For example, you can center a two-line heading as follows:

```
.ce 2
Heading Printed
In Center of Page
```

If you use the **.ce** command with no argument, **nroff** assumes a default argument of one, and centers only the next line of input. The command **ce 0** cancels any earlier centering command that is in operation.

Tabs

If your **nroff** input includes tables, you may find it convenient to use tabs to separate items in a line of the table. **nroff** recognizes the **<tab>** character and expands it into spaces. If you use tabs to format a table, remember to use no-fill mode; otherwise, **nroff** tries to fill and adjust your output lines.

By default, **nroff** sets a tab stops after every eight characters. You can use the *tab* primitive **.ta** to change the positions of the tab stops. For example,

```
.ta 10 20 30 40 50 60
```

sets tab stops ten characters apart rather than eight. **.ta** can also be used to fix tab stops in inches rather than after a number of characters; for example,

```
.ta 0.8i 2.0i
```

sets tab stops after 0.8 inches and 2.0 inches on the output line. This is quite helpful when you are designing a table.

You can use the *tab character* command **.tc** to change the character **nroff** prints between its current position and the next tab stop. Enter the following text to see how this primitive works:

```
.ta 9 19 29 39
.tc *
.nf
<tab>1<tab>2<tab>3<tab>4
```

The output file, **ex3.p**, should appear as follows:

```
*********1*********2*********3*********4
```

Page Breaks

The *begin page* primitive **.bp** causes a break and forces **nroff** to the next output page. By default, **nroff** assumes a page length of 11 inches (66 lines). You can change the page length with the *page length* command **.pl.** For example,

```
.pl 2i
```

specifies a two-inch page length.

At this point, the question arises about how **nroff** handles top and bottom page margins, number pages, and other aspects of page layout. The answer is it merely keeps track of the current output page number and the current line number on the current output page; designing top and bottom margins, page headers and footers, and other aspects of page layout is up to you.

Can **nroff** execute a set of commands whenever it reaches a certain position on the page? This would solve the problem of producing top and bottom margins, and you would not have to guess where to insert the commands in your script. In fact, you can tell **nroff** to do this, by using *traps.* The next section of this tutorial describes macros and traps and how to use them to format a page.

Macros and Traps

This section presents **nroff** macros: how to write them, how to tell **nroff** to execute them at a give point on every output page, and how to install a macro file under the COHERENT system

As with previous sections, this one uses a number of exercises. Working the exercises will help you master **nroff** quickly. When you format the exercise scripts, do not use the **-ms** option. Also, it is not necessary for you to copy the comments into your system; they are here to help you understand what each **nroff** command does, but they have no effect on how the script executes.

What Is a Macro?

To become familiar with the idea of a *macro*, consider the problem of formatting a paragraph. Whenever you come to a new paragraph, you want **nroff** to skip a line and indent the first line five spaces. Because **nroff** preserves blank lines and initial indentations, you could force **nroff** to break your text into paragraphs simply by inserting a blank line and spaces directly into your text. The same effect, however, can be achieved by inserting following set of **nroff** commands

```
.br             \" break
.sp             \" skip a line
.ti 5           \" indent next line 5 spaces
```

between the end of each paragraph and the start of the next paragraph. You should recognize the first two commands: **.br** causes a break, so that **nroff** prints the last line of the previous paragraph even though it might not be a complete line; **.sp** skips a line before the next paragraph begins. The third command is the *temporary indent* command **.ti**, which tells **nroff** to indent the next line; the number indicates how many spaces to indent. The following exercise, **ex4.r**, demonstrates how this works:

```
.ll 3i          \" line length
.po 3i          \" page offset
```

```
.ti 5             \" indent next line
Adam was human--this explains it all.  He did
not want the apple for the apple's sake, he
wanted it because it was forbidden.  The mistake
was in not forbidding the serpent; then he would
have eaten the serpent.
.br               \" break
.sp               \" skip a line
.ti 5             \" indent next line
Training is everything.  The peach was once a bitter
almond; cauliflower is nothing but cabbage with a
college education.
.br
.sp
.ti 5
Habit is habit, and not to be flung out of the window
by any man, but coaxed downstairs a step at a time.
```

After you have processed this file, the output file **ex4.p** should resemble the following:

```
     Adam was human--this explains it all.  He did not want the apple for the apple's sake,
he wanted it because it was forbidden.  The mistake was in not forbidding the serpent; then he
would have eaten the serpent.

     Training is everything.  The peach was once a bitter almond; cauliflower is nothing but
cabbage with a college education.

     Habit is habit, and not to be flung out of the window by any man, but coaxed downstairs a
step at a time.
```

Now, in a small file it would be easy to type all of the **nroff** primitives directly into your input text; however, what if your file is very long, with hundreds of paragraphs? Every time you wanted to begin a paragraph, you would have to include that set of commands within the text. You would save considerable agony if you could bundle these commands together under a common *name*; then you could simply put that *name* into your text whenever you wanted **nroff** to perform these commands, rather than typing the commands themselves over and over again.

As you probably have guessed by now, you can do just that; the set of commands is called a *macro*. The following shows the selections from Pudd'nhead Wilson's calendar set with a macro called **.PP** that takes care of formatting each paragraph. The following exercise, **ex5.r**, shows how to bundle together the **nroff** primitives for formatting paragraphs into the **.PP** macro:

```
.de PP            \" define the PP macro
.br               \" break the line
.sp               \" insert a blank line
.ti 5             \" indent next line 5 spaces
..                \" two periods ends the macro definition
.PP               \" execute PP macro
Adam was human--this explains it all.  He did
not want the apple for the apple's sake, he
wanted it because it was forbidden.  The mistake
was in not forbidding the serpent; then he would
have eaten the serpent.
.PP
Training is everything.  The peach was once a bitter
almond; cauliflower is nothing but cabbage with a
college education.
.PP
Habit is habit, and not to be flung out of the window
by any man, but coaxed downstairs a step at a time.
```

As you can see, using a macro can save you a considerable amount of work when you prepare your script.

Introducing Traps

Now, consider the problem of formatting the beginning and ending of each page of output. You could define what are traditionally called *header* and *footer* macros, which contain the commands you want performed at the top and bottom of each page. However, how can you tell **nroff** when to execute these macros? You cannot possibly know where to call these macros in the input text, because you cannot know where any given text line will appear in the output until you have processed it through **nroff**. This problem is solved by using *traps.*

nroff keeps track of its vertical position on each output page. You can set a *trap* that tells **nroff** to execute a macro at a particular vertical position on every page. When a line of output reaches or extends past the position that is specified in your *trap*, **nroff** then executes the commands named in the trap command before processing any more input text.

You can set a trap by using the *when* command **.wh**. For example, if you want **nroff** to call your header macro **.HD** at the very top of each page, the command

```
.wh 0 HD            \" set header trap
```

sets a trap for the macro **.HD** at vertical position 0 (the very top of the page) of every output page. The macro **.HD** will then be executed every time **nroff** begins a new page. To have your footer macro **.FO** execute one inch from the bottom of each page, use the command

```
.wh -1i FO          \" set footer trap
```

The negative number tells **nroff** to measure distance from the *bottom* of the page rather than from the top; the **i** is an abbreviation for inches. (**nroff** recognizes various units of measurement; this will be described in more detail later.)

Headers and Footers

Suppose you want to design the output page by defining the header and footer macros. A simple header macro merely skips an inch of space at the top of each page; a simple footer macro forces printing to stop an inch from the bottom of each page and prints the page number. **nroff** does not print page numbers automatically, but it does automatically keep track of which output page it is on. It stores the page number internally in a *number register* that you can access with the symbol '%'. (A later section gives more information about number registers and how to use them.)

The following gives a simple footer macro that prints the page number:

```
.de FO              \" define footer macro FO
'sp 4v              \" skip four vertical lines (no break)
.tl ''- % -''       \" print hyphen, page number, hyphen
'bp                 \" jump to new page
..                  \" end macro definition
```

There are several points of interest raised by this macro.

First, notice that some commands are preceded with an apostrophe rather than with a period. The use of the apostrophe instead of the period tells **nroff** to suppress the break these commands normally cause. You might run into problems if you define your header macro as follows:

```
.de HD              \" header macro
.sp 1i              \" skip an inch (break)
..
```

You want this to leave a blank space of one inch at the top of each page; however, the **.sp** command causes a break, so that if a word were left over from the last line on the preceding page, **nroff** would print it at the very top of the next page. The effect would be quite unsightly. However, if you use **'sp** instead of **.sp** in the macro, **nroff** suppresses the break and does not print the partial word until *after* it performs the macro commands. The same is true for the footer macro: you do not want anything unplanned to be printed in the blank space at the bottom of the page. You should always be conscious of these considerations when you use commands that cause breaks.

Another new item in the above example is the *title* command **.tl,** which prints a three-part title. A three-part title contains a left part (aligned to the left margin of the page), a center part (centered), and a right part (aligned to the right margin). The command name **.tl** is followed by four apostrophes: **nroff** prints the characters between the first two apostrophes as the left part of the title line, those between the second and third apostrophes as the center part, and those between the third and fourth apostrophes as the right part of the three-part title. If you do not want **nroff** to print anything in one of these positions, simply put nothing between the appropriate pair of quotes.

In the above example, the **.tl** primitive tells **nroff** to print nothing in the left and right portions of the footer title line, but to print the page number in the center. If you want an apostrophe to appear as a part of the title, precede it with the backslash character '\'.

nroff considers the length of the title line to be independent of the length of normal output lines; therefore, you must set it with the *length of title* primitive **.lt** unless you want **nroff** to use the default title length of 6.5 inches. For example, to set the length of the title to five inches, use the command

```
.lt 5i
```

In light of all you now know, you should give Pudd'nhead Wilson's calendar the treatment it deserves:

```
.ll 3i              \" set line length to 3 inches
.po 2i              \" set page offset to 3 inches
.pl 9i              \" set page length to 9 inches
.wh 0 HD            \" set the header trap
.wh -1i FO          \" set the footer trap
.de HD              \" define header macro HD
'sp 1i              \" skip 1 inches of space
..                  \" end macro definition
.de FO              \" define footer macro
'sp 2               \" skip 2 lines
.tl ''- % -''       \" define footer title
'bp                 \" begin new page
..                  \" end macro definition
.de PP              \" define paragraph macro
.sp 1               \" skip 1 line of space
.ti 5               \" indent the first line 5 characters
..                  \" end macro definition
.PP
Adam was human--this explains it all.  He did
not want the apple for the apple's sake, he
wanted it because it was forbidden.  The mistake
was in not forbidding the serpent; then he would
have eaten the serpent.
.PP
Training is everything.  The peach was once a bitter
almond; cauliflower is nothing but cabbage with a
college education.
.PP
Habit is habit, and not to be flung out of the window
by any man, but coaxed downstairs a step at a time.
```

As a point of technique, always set header and footer traps early in your input script; otherwise, **nroff** may not print the header on the first page.

Macro Arguments

You can affect how macros function by passing them modifiers, called *arguments*. An argument may be a bit of text that is arranged in a special way by the macro, or it may be a number or other parameter that dictates exactly what the macro does.

As an example of how a macro can handle arguments, consider a macro to format the list of ingredients for a recipe. You want the ingredients to be printed as follows:

```
3 cups of pumpkin
1 cup of milk
1 cup of sugar
1 tsp of ground ginger
1 tbl of cinnamon
```

Each of these lines has the same format: the amount of ingredient, the unit of measurement, the word "of", and the name of the ingredient. You can create a macro (call it **.RE**, for **re**cipe) that encodes the format of these lines and contains three "slots": one slot for the amount, one for the unit of measurement, and one for the name of the ingredient. Each time you use the macro, you indicate what you want to go into each slot, and **nroff** substitutes it

for you. The macro **.RE** can be constructed as follows:

```
.de RE              \" define macro RE
\\$1 \\$2 of \\$3   \" set RE's arguments
..                  \" end definition
Here is some text.
.nf                 \" don't fill the recipe
.RE 3 cups pumpkin
.RE 1 cup milk
.RE 1 cup sugar
.RE 1 tsp "ground ginger"
.RE 1 tbl cinnamon
.fi                 \" resume filling
Here is some more text.
.bp                 \" begin a new page, to force printing
```

When you call a macro that takes arguments, the arguments must appear on the same line as the macro command itself. A macro may have up to nine arguments; they are denoted by **\$1**, through **\$9**, respectively: the first field after the macro name is called **\$1**, the second **\$2**, and so on.

If you want to use as an argument a string of characters that includes blank spaces, you must enclose the string within quotation marks, as with the words "ground ginger", in the example above. If you forget to include the quotation marks, **nroff** regards each word in the string as a separate argument, and treats them accordingly.

Note that macros that are called by traps cannot accept arguments.

Double vs. Single Backslashes

If you carefully examine the definition of **RE**, you will see that it identifies each argument with two backslashes:

```
\\$1 \\$2 of \\$3
```

Whenever you identify an argument within a macro, always preface it with two backslashes, rather than one. The reason is that **nroff** in effect processes a macro *twice*: when it first reads it, and later when you call it within your text. Prefacing an argument with *one* backslash tells **nroff** that you want to expand that argument when the macro is first read; prefacing it with two backslashes tells **nroff** that you want to expand it when the macro is called in your text. In nearly every circumstance, you want to expand the arguments in your text, so you should use two backslashes. As you will see, this rule also applies to the use of *strings* and *number registers* within macros.

To see how this works, consider again the **.RE** macro:

```
.de RE
\\$1 \\$2 of \\$3
..
Here is some text.
.nf
.RE 3 cups pumpkin
.RE 1 cup milk
.RE 1 cup sugar
.RE 1 tsp "ground ginger"
.RE 1 tbl cinnamon
.fi
Here is some more text.
.bp
```

Using two backslashes, as above, allows you to redefine what **$1**, **$2**, and $3 mean many times throughout your text, to generate the following output:

```
Here is some text.
3 cups of pumpkin
1 cup of milk
1 cup of sugar
1 tsp of ground ginger
1 tbl of cinnamon
Here is some more text.
```

If you used only one backslash, however, your output would appear as follows:

TUTORIALS

```
Here is some text.
  of
  of
  of
  of
  of
Here is some more text.
```

nroff could not expand the argument calls (**\$1** etc.), because you had not yet defined them; therefore, it threw them away; and because all of the argument calls had been thrown away, **nroff** then threw all the arguments away. All that was left was word **of**.

Designing and Installing Macros

Now that you have been shown how to write a macro, the next step is to design some macros and install them, so you can call them over and over again.

The first step in designing a macro is to analyze the problem that you want to solve. Suppose that in this instance you want to print a list of names. Each name will consist of a first name, a last name, and the department with which he is associated, and the list will be printed in columns; for example:

```
Firstname      Lastname       Department
```

Moreover, you want to be able to switch the order in which the columns appear without having to retype your list; for example:

```
Lastname       Firstname      Department
```

or

```
Department     Lastname       Firstname
```

In effect, then, you want three macros: one for each of the three orders of columns shown above.

When you have finished designing your macros, they should look something the following. Type the following into the file **tmac.lst**; note that the symbol **<tab>** represents a *tab* character, and should not be entered literally:

```
.\" List macros. $1 represents first name,
.\" $2 last name, $3 department
.de LA
.nf
.ta 1.5i 2.75i
\\$1<tab>\\$2:<tab>\\$3
.Rt
..
.de LB
.nf
.ta 1.5i 2.75i
\\$2,<tab>\\$1:<tab>\\$3
.Rt
..
.de LC
.nf
.ta 1.5i 2.75i
\\$3:<tab>\\$2,<tab>\\$1
.Rt
..
```

The first lines are *comments*, so that anyone who looks at these macros will know what they do. The first command line, introduced with the **.de** command, names each macro. These names were selected after checking the file **tmac.s**, which is where the **-ms** macro package is kept, to confirm that they are not used elsewhere. Naturally, using the same macro name in two different places can lead to a great deal of trouble.

The next command, **.nf**, turns off the **nroff**'s normal right justification, which otherwise would smear a table. The **.ta** command sets the tab characters at certain points on the page, measured from the left margin.

The next line gives the order in which the arguments appear. The arguments are separated by tab characters, and punctuation is inserted. The last command, **.Rt**, calls a macro in the file **tmac.s**; this macro resets **nroff** to its normal fill mode and returns the tab settings to normal. Note that these macros can be used only when you also use the **-ms** macro package.

After you have typed the macros into **tmac.lst**, carefully read over what you type to ensure that no there are no errors; if you find any, be sure to correct them. The final step is to move **tmac.lst** into the directory **/usr/lib**, which is where **tmac.s** is also kept.

To test your new macros, type the following text into the file **ex6.r**:

```
The following lists give the personnel who are involved in
this project:
.sp
.LA Ivan Morgan Engineering
.LA Marian Gusman Design
.LA George Meyer Electrical
.LA Catherine Scanlon "Metal Shop"
.LA Fred McElroy Carpentry
.LA Anne Assenmacher "Machine Shop"
.sp
.LB Ivan Morgan Engineering
.LB Marian Gusman Design
.LB George Meyer Electrical
.LB Catherine Scanlon "Metal Shop"
.LB Fred McElroy Carpentry
.LB Anne Assenmacher "Machine Shop"
.sp
.LC Ivan Morgan Engineering
.LC Marian Gusman Design
.LC George Meyer Electrical
.LC Catherine Scanlon "Metal Shop"
.LC Fred McElroy Carpentry
.LC Anne Assenmacher "Machine Shop"
.sp
We expect that they will receive your full cooperation.
```

The same set of names is used three times; the only difference is the macro call employed.

Now, process this file with the following command:

```
nroff -ms -mlst ex6.r >ex6.p
```

As you can see, when you installed **tmac.list** into **/usr/lib**, you could invoke it in the same way that you invoke **tmac.s** with **-ms**.

When you look at the output file **ex6.p**, you should see something that resembles the following:

```
The following lists give the personnel who are involved in this project:

Ivan              Morgan:           Engineering
Marian            Gusman:           Design
George            Meyer:            Electrical
Catherine         Scanlon:          Metal Shop
Fred              McElroy:          Carpentry
Anne              Assenmacher:      Machine Shop

Morgan,           Ivan:             Engineering
Gusman,           Marian:           Design
Meyer,            George:           Electrical
Scanlon,          Catherine:        Metal Shop
McElroy,          Fred:             Carpentry
Assenmacher,      Anne:             Machine Shop

Engineering:      Morgan,           Ivan
```

```
Design:           Gusman,          Marian
Electrical:       Meyer,           George
Metal Shop:       Scanlon,         Catherine
Carpentry:        McElroy,         Fred
Machine Shop:     Assenmacher,     Anne

We expect that they will receive your full cooperation.
```

As you grow proficient in writing **nroff** macros, you will probably find it most efficient to keep special macros in their own files; this will save time by ensuring that **nroff** does not have to process macros that are never called.

Strings

Suppose you are writing a script for **nroff** and, to relieve the tedium, decide to punctuate the text occasionally with a rousing cry of "FOOD FIGHT!!". If you plan to interject this phrase more than a few times in your script, you can take advantage of another labor-saving device, called a *string*. You can use a string name as an abbreviation for a long string of characters you use frequently. Like a macro, a string is a *name* that **nroff** associates with a *definition* that you supply. Wherever you put the name in your text, **nroff** prints the definition. Although macros refer to sets of *commands* that you define, strings refer to strings of *characters* that you define.

You define a string with the *define string* primitive **.ds:**

```
.ds FF "FOOD FIGHT!!"
```

The first field after the **.ds** gives the name of the sting, in this case **FF.** Like a macro name, a string name may be either one or two characters. The second field after the **.ds** gives the definition of the string, in this case

```
"FOOD FIGHT!!"
```

As in this example, you must enclose the definition within quotation marks if it contains spaces.

Be careful whenever you define a macro or a string. If you already have a macro or a string named **X** and you define a new macro or string named **X, nroff** forgets the previous meaning of **X.**

Once you have defined a string, you can refer to it anywhere in your text. The string itself appears in the output text wherever a reference to it appears in the input text. You refer to the string **FF** in the following fashion:

```
\*(FF
```

Use the left parenthesis '(' only when the name of the string is two characters long. If the string name is only a single character, such as **S,** refer to it as follows:

```
\*S
```

As an example, type the following script into **ex7.r**, and process it through **nroff**; do *not* use the **-ms** macro package:

```
.ds FF "FOOD FIGHT!!"
.ds W "WHOOPEE!!"
.ce
From Aristotle's "Poetics"
.br
.sp
A tragedy is the imitation of an action \*(FF
that is serious and also, \*W as having magnitude,
complete in itself, with incidents \*(FF
arousing pity and fear, wherewith to accomplish \*W
\*(FF its purgation of such emotions \*(FF \*(FF.
.bp
```

nroff adjusts the spacings between words in a string but does not hyphenate any word that is within a string. If you use a very short line length, such as two inches, and define a string that includes a three-inch long word, that word would not be hyphenated but would extend past the right-hand margin.

You cannot include a newline character in a string. However, you can spread the definition of a string out over more than one line with the aid of *concealed* newlines (preceded by the backslash character '\'). **nroff** ignores each concealed newline. For example, add the following string to the previous example:

```
.ds PR "GO TEAM \
GO!!!"
```

As you can see, **nroff** ignores concealed newlines anywhere in its input.

Strings Within Strings

You can define a string that has embedded within it a reference to another string. Whenever you refer to the bigger string in your text, **nroff** substitutes the definition of the smaller string for any reference to the smaller string. When you embed strings, though, you should use *two* backslashes to refer to the embedded string, for the same reason that you should use two backslashes to refer to an argument within a macro:

```
.ds S "This string \\*x has embedded \\*y strings"
```

To help understand this better, type following three scripts into your computer and format them with **nroff**. The first script contains proper references to embedded strings (using double backslashes); it works as expected:

```
.ds S "strings \\*X, strings \\*Y, strings \\*Z"
.ds X "here"
.ds Y "there"
.ds Z "everywhere"
\*S
```

The next script contains embedded references that use only single backslashes. Because the embedded strings are defined after the larger string, they are not available when **nroff** defines the larger string, and so the references are ignored:

```
.ds S "strings \*X, strings \*Y, strings \*Z"
.ds X "here"
.ds Y "there"
.ds Z "everywhere"
\*S
```

The third script again contains embedded references using single backslashes. This time, the embedded strings are defined *before* the larger string, and so are available when the larger string is defined:

```
.ds X "here"
.ds Y "there"
.ds Z "everywhere"
.ds S "strings \*X, strings \*Y, strings \*Z"
\*S
```

To avoid unnecessary worry, you should always play it safe and use double backslashes to refer to embedded strings.

Number Registers

You learned in previous sections that **nroff** keeps track of the output page number while it prints its output. You made use of this fact when you created a footer macro that printed page numbers. **nroff** also keeps track of other housekeeping information, such as the current line length, page offset, page length, and vertical position of the last output line. It keeps this information in storage locations called *number registers*.

You can use the *name* of a number register to refer to the number that is stored in it. When you place a reference to a number register in your text, **nroff** substitutes for the name whatever number is currently in the register.

Number register names are one or two characters long, just like macro and string names. You can have a number register with the same name as a string or a macro without confusing **nroff**, even though you cannot give a macro and a string the same name. However, *you* might become confused; **nroff** scripts usually are easier to understand if you keep all macro names, string names, and register names distinct.

Another difference between number registers, macros, and strings is that **nroff** itself does not define any macros or strings (although the **-ms** macro package does), but it does automatically define and update quite a few number registers. You can use these predefined number registers in much the same way that you use registers you define yourself, except that you cannot change their values.

To define a number register, you must specify the *register name* and the *initial value* for the register. The *number register* primitive **.nr** looks like this:

```
.nr X 5
```

Here **X** is the name of the register and **5** is the initial value to store in it. To refer to number register **X** in your text, use **\nX**; if the name is two characters long (for example, **XY)**, use **\n(XY**. This is exactly like the way you refer to a string, except that you use the letter 'n' instead of an asterisk '*'. When **nroff** sees a reference to number register **X**, it automatically substitutes the value stored in **X**. As you will see shortly, **nroff** can do arithmetic, and

learning to use number registers is an important part of learning to take advantage of **nroff**'s arithmetic abilities.

A reference to a number register can occur anywhere a number would normally occur. For example, if you set register **X** to 5, as above, you can set the line length to five inches as follows:

```
.ll \nXi
```

This command is essentially the same as

```
.ll 5i
```

if the current value of register **X** is 5.

A familiar problem arises when you refer to a number register inside a macro or a string definition. If you use just one backslash, **nroff** substitutes the value in the register for the reference when it first processes the macro or string. If you have not yet defined the number register in your script, **nroff** inserts **0** into the macro or string. Normally, you should use a a double backslash, such as **\\nX** or **\\n(XY**, when referring to a number register within a macro or string. Using the double backslash is particularly important if you *change* the value of the register throughout your script, and want the current value to appear in the macro or string each time you call it.

Try typing the following examples into your computer, and processing them with **nroff**. See if you can describe why **nroff** prints what it does in each case. The first example defines a string with a register reference preceded by a single backslash.

```
.ds S "Here is a number \nX"
.nr X 55
\*S
\nX
```

You should see the following output:

```
Here is number 0
55
```

nroff printed what it did because number register X had not yet been defined when it was called in string S; **nroff** therefore erased the reference to X and substituted zero for it. Number register X was then set to 55, which was printed when the register was specifically called later in the script.

The second example is similar, but now the number register is set *before* the string is called:

```
.nr Y 56
.ds T "Here is a number \nY"
\*T
\nY
```

Now the output is

```
Here is a number 56
56
```

The third example uses a double backslash for the register reference.

```
.ds U "Here is a number \\nZ"
.nr Z 57
\*U
.nr Z 58
\*U
```

This script produces the following:

```
Here is a number 57
Here is a number 58
```

The final example uses a single backslash again.

```
.nr W 59
.ds V "Here is a number \nW"
\*V
.nr W 60
\*V
```

The following is produced:

```
Here is a number 59
Here is a number 59
```

The last example illustrates the danger of using a single backslash to refer to a number register within a string definition. You defined the number register **W** before you defined the string **V**, so the value for **W** was available when **nroff** read the definition of **V**. **nroff** substituted the value when it reads the definition; the reference to the number register **W** is no longer there. You then change the value of **W**, but as you see in the next call of **V**, the change does not affect the number that appears in **V**. In contrast to this, notice in the third example that the double backslash in the definition of **U** allows the reference to number register **Z** to remain within the definition of string **U**. Whenever you change the value of **Z** and then call **U**, **nroff** substitutes the new value of **Z** for the reference to **Z** within **U**.

You can also use the **.nr** primitive to increase or decrease the value in a number register. For example, suppose you initially store the value five in **X**:

```
.nr X 5
```

Incrementing and Decrementing

You can change the value of **X** to 9 by adding 4, as follows:

```
.nr X +4
```

You can then change the value of **X** to 7 by subtracting 2:

```
.nr X -2
```

A plus or minus sign before a number on the **.nr** command line tells **nroff** to add or subtract the given amount to or from the value in the register. Because a negative number is always preceded by a minus sign whereas a positive number usually is not preceded by a plus sign, you can use **.nr** to set a register to a positive value in a way that cannot be imitated for negative values. For example, suppose you again start out with number register **X** set to a value of 5:

```
.nr X 5
```

If you immediately follow this with

```
.nr X 7
```

then **nroff** replaces the value of 5 with 7. The second **.nr** does not increase the value of **X** by 7 to produce 12; rather, it wipes out the previous value of 5 and replaces it by the value 7. The command line to increase **X** by 7 is

```
.nr X +7
```

If you again start with a value of 5 in **X** and want to change the value to -4, you cannot use the following command line:

```
.nr X -4
```

nroff interprets this as a command to *decrease* the current value of **X** by 4, which is not what you intended. This command places the value 1 in **X**, since 5-4=1. If **X** initially has a value of 5 and you want to change the value to -4, you could use the command

```
.nr X -9
```

You can also increase or decrease the value of a number register without using **.nr**. If number register **X** currently has the value 10, the reference **\n+X** increases the value in **X** by 1 to 11 and substitutes the new value for the reference. The value in **X** becomes 11; **nroff** replaces the next reference **\nX** by 11, whereas another reference **\n+X** increments the value in **X** to 12 and replaces the reference by 12. Similarly, if number register **XY** currently has the value 15, the reference **\n+(XY** increases the value in **XY** to 16 and replaces the reference by 16.

You can also decrease a register's value. The reference **\n-X** decreases the current value in **X** by 1 and substitutes the new value for the reference. Likewise, the reference **\n-(XY** decreases the current value in **XY** by 1 and substitutes the new value for the reference.

You can change the size of the increment or decrement by means of another option to the **nr** command. If you define **X** with

```
.nr X 1 5
```

then **nroff** sets the value of **X** to 1 and sets the increment value for **X** to 5. The next reference **\n+X** increments

the value in **X** from 1 to 6 (the '+' now causes **nroff** to add 5 to the current value of **X** rather than adding 1) and substitutes 6 for the reference. In the same manner, **\n-X** subtracts 5 from the current value of **X** and substitutes the new value for the reference. This is convenient if you plan to repeatedly increment or decrement **X** by the same fixed amount. If you wish to change the size of the increment, simply redefine **X** with another **.nr** that specifies the new initial and increment values. If you define a number register but do not specify an increment value, **nroff** assumes the increment value to be 1.

The following example of a macro illustrates a typical use of a number register and incrementing.

```
.nr W 1                                    \" set W to 1, inc by 1
.ds X "Here's Wrestler No. \\nW,"          \" define string X
.de B                                      \" define macro B
.br
\\*X \\$1!!!                               \" define arg to macro B
.nr b \\n+W                                \" increment W
..                                         \" end definition
.B "Alex 'Killer' Bovine"                  \" call B with arguments
.B "William 'Crusher' Risible"
.B "Vlad 'the Impaler' Acephalous"
.bp                                        \" force printing of page
```

to produce the following output:

```
Here's Wrestler No. 1, Alex 'Killer' Bovine!!!
Here's Wrestler No. 2, William 'Crusher' Risible!!!
Here's Wrestler No. 3, Vlad 'the Impaler' Acephalous!!!
```

A reference to a number register may appear any place a number can normally appear. For example:

```
.nr X \nY \nZ
```

sets register **X** to the value of register **Y** and sets the increment for **X** to the value of register **Z.**

As mentioned before, **nroff** performs arithmetic. It understands and evaluates properly formed arithmetic expressions involving numbers, references to number registers, the arithmetic operators '+', '-', '*', '/', '%', and parentheses. The first four operators represent addition, subtraction, multiplication, and division. The '%' is the *modulus* or *remainder* operator: the value of 7%3 is 1, which is the remainder when 7 is divided by 3.

One word of caution: **nroff** evaluates expressions from left to right without any preference for performing some operations before others. For example,

```
.nr X 5+4*3/9
```

stores 3 in **X. nroff** does not perform the multiplication and division before the addition, as you might expect.

Another important fact is that number registers hold only integers. If you write

```
.nr X 3.6
```

nroff truncates the value 3.6 and stores 3 in **X**. Also, an assignment such as

```
.nr X 3.9*3.9
```

stores 9 in **X**; **nroff** truncates each factor before it performs the multiplication. The assignment

```
.nr X 0.4*8
```

stores 0 in **X** rather than 3: truncation occurs before **nroff** performs the multiplication rather than after.

A final word of caution: when you use numbers with commands other than **.nr,** the results may *not* be what you expect. **nroff** understands several different units of measurement and converts between units automatically. The next section explains units and conversion in detail.

Units of Measurement

As mentioned above, **nroff** maintains many number registers during processing. For example, it stores the current page length in the register **.l** (Note that the period '.' is actually part of the name of this register.) If you set the line length to five inches with the command

```
.ll 5i
```

nroff stores the length in register **.l** automatically; however, if you print the value in register **.l** by entering **\n(.l**, you find the value is 600. What does this mean?

Many **nroff** commands require that you specify lengths or measurements as arguments. You are already familiar with many of these commands: for example, **.ll, .po, .pl,** and **.lt**. **nroff** accepts various units of measurement, but for purposes of calculation, it converts each into a basic unit called a *machine unit*, which is abbreviated **u**. A machine unit is 1/120 of an inch long. Because one inch is 120 machine units, the length of a five-inch line is 5 times 120, or 600 machine units.

The conversion table for units of measurement is as follows:

inch:	1**i** = 120**u**
vertical line space:	1**v** = 20**u**
centimeter:	1**c** = 47**u**
em:	1**m** = 12**u**
en:	1**n** = 6**u**
pica:	1**P** = 20**u**
point:	1**p** = 1**u**

Most of these are traditional typesetting terms.

As noted briefly earlier, **nroff**'s output actually consists of a sequence of characters. It is useful, though, to think of the output as being "printed" at ten characters per inch (Pica or 10-pitch spacing) and six lines per inch. Many output devices use this spacing. With these assumptions, **5i** is equivalent to five inches of printed output.

Every **nroff** command has a default unit of measurement. For example, the default unit for **.ll** is **m**, whereas the default unit for **.sp** is **v**. If you type

```
.ll 5
```

nroff interprets it not as five inches or five centimeters, but as 5**m**, which it converts to 5 times 12, or 60 machine units (60**u**).

nroff always assumes a unit specification as part of each number and automatically converts each number and its unit specification into machine units. If you append an explicit unit specification to the number, **nroff** uses it; if you do not, **nroff** uses the default unit for the command.

For example, suppose you write the following commands:

```
.nr X 2i
.ll \nX
```

What line length results? The first command stores the number 2 times 120, or 240, in register **X.** The second command is therefore equivalent to typing

```
.ll 240
```

However, the default unit for **.ll** is **m.** Because 1**m** equals 12**u**, **nroff** sets the line length to 12 times 240, or 2,880 machine units. If you wanted a line length of two inches to result from the above commands, you will be unpleasantly surprised, because 2**i** equals only 240**u**. Instead, you should write:

```
.nr X 2i
.ll \nXu
```

By including the **u** in the **.ll** primitive, you do not accidentally multiply your results by 12, as happened earlier.

You should think of the unit specification as a part of a number. Because **nroff** accepts so many different units of measurement, a number without a unit specification is ambiguous. What does '5' mean? Five inches? Centimeters? Ems? **nroff** must know what unit of measurement you are using. If you think of the unit specification as a part of the number, you will have less trouble with potentially mystifying situations like the following. As mentioned, number registers store only integers and **nroff** truncates each number in an arithmetic expression to an integer before evaluating the expression. Therefore, the following stores 0 in register **X**:

```
.nr X 0.4*9
```

But now try the following:

```
.nr X 0.4i
\nX
```

This does not store 0 in **X** like the previous command; it stores 0.4 times 120, or 48 in **X**. The 0.4 is not truncated

to 0 here! Truncation occurs *after* conversion to machine units, so **nroff** truncates 0.4**u** in the first example. But the number in the second example is given in inches **i** instead of machine units **u**. **nroff** converts it to **u** before truncating to get an integer.

As another example, the following stores 1 in **X**:

```
.nr X 0.01i
```

nroff converts 0.01 inches to 0.01 times 120, or 1.2**u**, and then truncates 1.2 to 1.

The following command illustrates that **nroff** understands *each* number in an arithmetic expression to have an attached unit specification, whether you supply one or not.

```
.ll 2*8
```

Recall that **nroff** stores the current line length in the register **.l;** if you type

```
\n(.l
```

you will receive the number 2,304. **nroff** interprets the 2 as 2**m** and the 8 as 8**m**, because the default unit for **.ll** is **m.** Then it converts each to machine units and multiplies to give the result: (2*12)*(8*12), or 2,304.

Consider one final example that illustrates the unusual consequences of seemingly innocent assignments. Suppose you set the page offset as follows:

```
.po 8/3
```

nroff stores the current page offset in register **.o**. To see what number it stores there, type

```
\n(.o
```

You see that the page offset is 2. Because the default unit for **.po** is **m,** the calculation is (8*12)/(3*12)=8/3, which **nroff** truncates to 2. Two machine units is equivalent to only 1/60 of an inch. This is not a physically reasonable value for most typewriter-like devices, so a page offset of 0 characters results. On the other hand,

```
.po 8/3u
```

produces a page offset of approximately 1/4 of an inch.

Conditional Input

Now that you have been introduced to number registers, you can use them in conjunction with powerful *conditional commands* to create more elaborate **nroff** scripts.

To see how conditional statements help you construct an **nroff** script, consider again the problem of creating header and footer macros. Earlier, you constructed macros that skipped space at the top of the page and printed the page number at the bottom of each page.

Suppose, however, that you are formatting a paper that has a title. You want to print the page number for page 1 at the bottom of the page, and to print the rest of the page numbers at the top of the page. Both the header and the footer need some kind of conditional mechanism to perform differently on the first page than on subsequent pages. On page 1, the header should skip to where the title will be printed; on other pages, the header should print the page number. On page 1, the footer should print the page number; on other pages, the footer should leave a block of blank space at the bottom of the page.

To execute commands conditionally, use the *if/else* commands **.ie** and **.el**, which are demonstrated in the following example. Note that the formation ’’, which is used with the **.tl** command, represents two apostrophes, *not* a quotation mark.

```
.de HD              \" define header
.ie \\n%=1 .A
.el .B              \" else do B
..
.de A               \" define macro A
.sp |1.0i           \" space down to 1 inches from top of page
..
.de B               \" define macro B
’sp 2v              \" skip 2 spaces
.tl ’’- % -’’       \" print page no.
’sp |1.0i           \" skip to 1 inch from top of page
..
```

```
.de FO              \" define footer
.ie \\n%=1 .C       \" if page no. is 1 then do C
.el .D              \" else do D
..
.de C               \" define macro C
'sp |-4v            \" move to 4 in. above bottom of page
.tl ''- % -''       \" print page no.
'bp                 \" begin new page
..
.de D               \" define macro D
'bp                 \" begin new page
..
```

As you can see, the **.ie** and **.el** commands always occur in pairs. **.ie** consists of three parts: the command name **.ie**, then a *condition* that **nroff** tests, followed by a *command* **nroff** performs if the condition is true. If the condition on the **.ie** command line is not true, **nroff** performs the command on the **.el** line instead.

In the example, each conditional invokes a macro on the command line. Actually, the conditional can specify *input* text rather than the command after the condition. If you want to execute several commands or include several text lines conditionally, enclose the lines with the special sequences '\{' and '\}'.

Note, too, that one other new element was introduced in the construction of these macros. Some of the **.sp** commands have a vertical bar immediately in front of the measurement; for example,

```
.sp |1.0i
```

Normally, when **nroff** sees a command like **.sp 1.0i,** it moves down one inch on the output page. The movement is relative to where **nroff** happens to be on the output page when it received the request. The vertical bar tells **nroff** that the following measurement is an *absolute* measurement, measuring either from the top of the page (if positive) or from the bottom of the page (if negative). Therefore,

```
.sp |1.0i
```

tells **nroff** to move to one inch from the top of the page;

```
.sp |(-4v)
```

tells it to move to four vertical spaces from the bottom of the page.

The **.if** primitive is formed exactly like **.ie**. Unlike **.ie**, which must always be used with **.el**, the **.if** command may be used by itself. If the condition on the **.if** line is true, **nroff** performs the command that follows the condition; if the condition is false, it ignores the command altogether.

This chapter ends with two substantial examples that incorporate most of what you have studied so far. To illustrate the use of conditionals, the first example begins each even paragraph of output with the phrase **Even Paragraph:** and begins each odd paragraph with the phrase **Odd Paragraph:**. Type this into the file **ex8.r**, and process it through **nroff** without using the **-ms** macro package, and as before, there is no need to copy the comments:

```
.wh 0 HD            \" set header trap
.wh -2i FO          \" set footer trap
.nr EO 1            \" set EO register to 1
.po 2i              \" page offset 2 inches
.pl 6i              \" page length 6 inches
.lt 4i              \" title length 4 inches
.ll 4i              \" line length 4 inches
.de HD              \" define header
'sp |(1i-1v)        \" space down to 1 inch minus 1 line
.tl ''\\*(WS''      \" set WS macro in title
'sp |1.5i           \" space down to 1.5 inches
..
.de FO              \" define footer
'sp |(3i+3v)        \" space down to 3 inches plus 3 lines
.tl ''- % -''       \" set page number in footer
'bp                 \" begin new page
..
.ds WS "From the Devil\'s Dictionary"
```

```
.                   \" define string WS
.de PP              \" define paragraph macro
.ie \\n(EO=0 .EP    \" if EO = 0 (even) then do EP
.el .OP             \" else do OP
..
.de EP              \" define EP (even paragraph)
.br
.nr EO 1            \" set register EO to 1
.sp 1v              \" skip 1 line
.ll 4i              \" set line length to 4 inches
.lt 4i              \" set title length to 4 inches
\\*E                \" insert string E
..
.ds E "Even Paragraph:"
.                   \" define string E
.de OP              \" define macro OP (odd paragraph)
.br
.nr EO 0            \" set register EO to 0
.sp 1v
.ll 3i              \" set line length to 3 inches
.lt 3i              \" set title length to 3 inches
\\*O                \" insert string O
..
.ds O "Odd Paragraph:"
.                   \" define string O
.PP
Debt, n. An ingenious substitute for the whip
and chain of the slave-driver.
.PP
Bore, n. One who talks when you wish him to listen.
.PP
Brandy, n. A cordial composed of one part
thunder-and lightning, one part remorse, two parts
bloody murder, one part death-hell-and-the-grave,
and four parts clarified Satan.
.PP
Responsibility, n. A detachable burden easily
shifted onto the shoulders of God, Fate, Fortune,
Luck, or one's neighbor.
```

This example uses an "even/odd" register called **EO** to determine whether you are beginning an even or an odd paragraph. To distinguish between even and odd paragraphs, it uses a line length of four inches for even paragraphs and one of of three inches for odd paragraphs. It changes the title length with each paragraph, so **nroff** centers the page number with respect to whichever kind of paragraph happens to occur at the bottom of a page.

The final example illustrates a loop constructed with the **if/else** commands. The first paragraph is six inches long with no page offset; each succeeding paragraph is one inch shorter with a page offset one inch longer. The line length of the sixth paragraph is one inch; the next paragraph renews the cycle with a six-inch line length. Type this into file **ex9.r**, and process it as you did the above example:

```
.nr PO 0 1          \" set register PO to 0, increment by 1
.de PP              \" define paragraph macro
.ie \\n(PO=6 .A     \" if register PO=6 then do A
.el .B              \" else do B
..
.de A               \" define macro A
.br
.nr PO 0            \" set register PO to 0
.nr LL 6-\\n(PO     \" set register LL to 6 minus PO
.ll \\n(LLi         \" set line length to LL inches
.po \\n(POi         \" set page offset to PO inches
.nr PO \\n+(PO      \" increment register PO
```

```
.sp               \" skip a space
..
.de B             \" define macro B
.br
.nr LL 6-\\n(PO   \" set LL to 6 minus PO
.ll \\n(LLi       \" set line length to LL inches
.po \\n(POi       \" set page offset to PO inches
.nr PO \\n+(PO    \" increment register PO
.sp               \" skip a space
..
.PP
Future, n. That period of time in which our affairs prosper,
our friends are true, and our happiness is assured.
.PP
Gallows, n. A stage for the performance of miracle plays, in
which the leading actor is translated into heaven.
.PP
Geneaology, n. An account of one's descent from an ancestor
who did not particularly care to trace his own.
.PP
Guillotine, n. A machine which makes a Frenchman shrug
his shoulders with good reason.
.PP
History, n. An account most false, of events
most unimportant, which are brought about by
rulers mostly knaves, and soldiers mostly fools.
.PP
Idiot, n. A member of a large and powerful tribe
whose influence in human affairs has always been
dominant and controlling.
.PP
Kiss, n. A word invented by the poets as a rhyme
for "bliss".
```

You should try this example to see verify that "loop" works as advertised.

Environments and Diversions

Another aspect of **nroff**'s power is the ability to shift from one *environment* to another.

The **nroff** *environment* is the overall manner in which **nroff** processes your input text. The environment's definition includes such aspects as line length, fill and adjust modes, and indentation.

nroff allows you to define three independent environments, called **0**, **1**, and **2**. In each, you can set as you wish such parameters as line length, filling, adjustment, and indentation. You can call a different environment with the **.ev** command; the parameters you define for the new environment control text processing until you change them within the present environment or shift to another environment.

Not all **nroff** parameters change when you switch to a new environment. For example, different environments do *not* have independent page offsets; the **.po** command affects all environments. Parameters that may be set to different values in different environments are *environmental parameters*; parameters that cannot be switched according to environment, like page offset, are *global parameters*. Macro and string definitions are global.

When you first call **nroff,** you are by default in environment 0. In all the examples used in this tutorial thus far, everything happened in environment 0. The following example illustrates how to switch back and forth between environments. Type in the following **ex10.r** and process it to see the output as you go along.

```
.po 1i            \" set global page offset to 1 inch
.ll 4i            \" set line length in ev 0 to 4 inches
.de PP            \" define paragraph macro
.sp
.ti 0.5i          \" indent first line 1/2 inch
..
.PP
The heart of the righteous studieth to answer,
```

```
but the mouth of the wicked poureth out evil things.
.br
.ev 1              \" switch to environment 1
.ll 3i             \" set line length to 3 inches
.ls 2              \" line spacing now double space
.PP
A froward man soweth strife, and a whisperer
separateth chief friends.
.br
.ev                \" return to previous ev (0)
.PP
It is naught, it is naught, sayeth the buyer;
but when he is gone his way, then he boasteth.
.br
.ev 1              \" switch to ev 1
.PP
Wealth maketh many friends; but the poor is separated
from his neighbors.
.br
.ev                \" return to ev 0
```

The first **.ll** command sets a line length of four inches in environment 0. After defining the paragraph macro **.PP** and an initial paragraph in environment 0, you switched to environment 1 with the command

```
.ev 1
```

You now enter a new environment. If you do not explicitly set environmental parameters, such as line length, **nroff** automatically uses default values for them. **nroff** assigns the same default values in environments 1 and 2 as it does in environment 0.

The line length in environment 1 is set to three inches with the output text double-spaced. The *line space* primitive

```
.ls 2
```

leaves one blank line between each output line. Thus, paragraphs processed in environment 0 have four-inch single-spaced lines, whereas paragraphs processed in environment 1 have three-inch double-spaced lines.

The example used the command line

```
.ev
```

without an argument to *leave* environment 1. This leaves environment 1 and restores (or "pops") the previous environment — in this case, environment 0. The next time you enter environment 1, you will not need to set the line length to three inches again: the value stays in effect in environment 1 until you specifically change it. The same is true of all environmental parameters.

To understand how **nroff** switches between environments, imagine that you have a set of plates, each marked with either a **0**, a **1**, or a **2**. You have as many plates of each type as you wish. You stack the plates on a table; the top plate represents your current environment. You begin with a '0' plate on the table, to represent the initial environment when you enter **nroff**.

Switching to environment 1 with the command **.ev 1** corresponds to placing a '1' plate on top of the '0' plate. You can again change the stack of two plates either by placing a new plate on top of the stack, or by removing the top plate from the stack: the former corresponds to calling a new environment, whereas the latter corresponds to restoring the previous environment with the command line **.ev**.

Because you can have as many plates of each type as you wish, you can call environment 1, then call environment 2, then restore environment 1, then call environment 0, and so on. The command **.ev** *N*, where *N* is 0, 1, or 2, places (or "pushes") a plate onto the stack; the command **.ev** removes (or "pops") the top plate from the stack.

To illustrate this, add the following text to the end of the previous example. Use a piece of paper and pencil to keep track of how the **.ev** commands add or remove environments. Because the line lengths are different in each environment, it should be easy to tell in which environment **nroff** has processed each paragraph:

```
.ev 2              \" introduce environment 2
.ll 5i             \" set line length
.in 1i             \" set indentation
```

```
.PP               \" paragraph in ev 2
A poor man that oppresseth the poor is like
a sweeping rain which leaveth no food.
.br
.ev 0             \" push ev 0
.PP
As a roaring lion, and a ranging bear; so is
a wicked ruler over the poor people.
.br
.ev 1             \" push ev 1
.PP
Wrath is cruel, and anger is outrageous;
but who is able to stand before envy?
.br
.ev 2             \" push ev 2
.PP
A good name is rather to be chosen than
great riches; and loving favour rather than
silver and gold.
.br
.ev 0             \" push ev 0
.PP
Pride goeth before destruction, and an haughty
spirit before a fall.
.br
.ev               \" return to ev 2
.ev               \" return to ev 1
.PP
He that answereth a matter before he heareth it,
it is folly and shame unto him.
.br
.ev               \" return to ev 0
.ev               \" return to ev 2
.PP
A merry heart doeth good like a medicine, but a
broken spirit drieth the bones.
.br
```

Buffers

Earlier, it was shown that **nroff** uses a buffer to assemble words from its input into output lines. Actually, each environment has its own buffer. Switching to a new environment does *not* cause a break. Suppose you are currently in environment 1 with an unfinished line in the buffer. When you give the command **.ev 2**, the unfinished line remains undisturbed in the environment 1 buffer until you return to environment 1. Text you process in the meantime in environment 2 or in environment 0 has no effect on the partial line in the environment 1 buffer, because **nroff** assembles text processed in other environments in different buffers.

In the following example, you process some text in environment 0 and then switch to environment 2. Any partial line collected in environment 0 when you switch to environment 2 waits patiently in the buffer until you return to environment 0 and issue the break command to flush the buffer. You then return to environment 2 and flush any partially filled line left when you restored environment 0. Enter the following into the file **ex11.r** and process it through **nroff**:

```
.ll 3i            \" set line length in ev 0
.po 2i            \" set page offset in ev 0
This is environment 0.
.ev 2             \" introduce ev 2
This is environment 2
.br               \" flush ev 2 buffer
.ev               \" return to ev 0
.br               \" flush ev 0 buffer
```

As you can see, the order of the two sentences is reversed from the way you entered them. If you were to delete the **.br** commands after the texts in **ex10.r**, the output would be very badly affected.

Headers and Footers

A common use of environment switching is for the creation of header and footer macros. As the following example demonstrates, the length of title set by the **.lt** command is an environmental parameter. The following constructs header and footer macros that print strings of asterisks in the margins above and below the text; type it into your computer as **ex12.r**:

```
.wh 0 HD           \" set header trap
.wh |2.5i FO       \" set footer trap
.de HD             \" define header macro
.ev 1              \" define ev 1
.lt 5i             \" set title length to 5 inches
'sp 3v             \" move down three spaces
.tl '****''****'   \" define header title
'sp 2v             \" skip two more spaces
.ev                \" pop environment
..
.de FO             \" define footer macro
'sp 2
.ev 1              \" push ev 1
.tl '****'%'****'  \" define footer title
.ev                \" pop environment
'bp                \" begin new page
..
.ll 4i             \" set line length in ev 0
.pl 3i             \" set page length
.in 1i             \" set indentation
.po 2i             \" set page offset
.de PP             \" define paragraph macro
.sp 1
.ti 0.5i           \" indent 1st line 1/2 inch
..
.PP
When in the course of human events ...
```

The following section explains why header and footer macros often use a different environment.

More About Fonts

As earlier described in some detail, **nroff** output includes representations for **boldface** and *italic* characters, in addition to the normal Roman characters. The visual appearance of boldface and italic characters depends on the device you use to print your **nroff** output.

If you want a word or a phrase to appear in boldface, enclose the word or phrase between **\fB** and **\fR**:

```
The last word of this sentence appears in \fBboldface\fR.
```

The sequence **\fB** tells **nroff** to print in boldface, whereas the sequence **\fR** tells **nroff** to return to the Roman font. Italics are used in a similar manner:

```
An entire phrase \fIappears in italics\fR.
```

To print more than a few words in a different font, you should use the *font* command **.ft:**

```
.ft I
Here is text you want to
appear in italics.
.ft R
```

The initial **.ft I** switches the print to italic font, and the concluding **.ft R** returns it to Roman font. As you might have suspected, the command **.ft B** switches to boldface.

You have two additional options when you use the **.ft** primitive. The command **.ft P** returns to the *previous* font. You can use **.ft P** within a macro or a string to return to the previous output font, even though you do not know which font was previously in effect. You can also use the sequence **\fP** to return to the previous font. The **.ft** primitive without an argument tells **nroff** to return to the Roman font.

In scripts that frequently change fonts, you should switch to a new environment for header and footer macros, in order to protect their font settings.

Diversions

The *diversion* is a powerful feature that allows you to suspend outputting lines until **nroff** has collected all of a block of text. For example, suppose you use **nroff** to format a chapter of a book. The chapter includes footnotes at various places in the text; you want **nroff** to collect these footnotes and print at the end of the chapter. To do this, **nroff** must store the processed footnote text somewhere until the end of the chapter, when you want it printed. Where do you store the text until the time comes for it to appear? To handle situations like this, **nroff** provides a *diversion* mechanism: you can *divert* text into temporary storage within a macro.

Diversion normally involves passing to a new environment to process the footnote without causing a break in the main environment. When the text of the diversion ends, **nroff** returns to the main environment, again without causing a break, and continues processing just as if the text of the note had never been in the input.

However, before you attempt to write a footnote macro, type the following text into the file **ex13.r**, and process it with **nroff**. This example illustrates the basic features of diversion. The example exchanges two paragraphs of text, so that **nroff** prints the second before the first.

```
.di X             \" divert the following to macro X
.sp
A soft answer turneth away wrath:
but grievous words stir up anger.
.br               \" send last line of paragraph to X
.di               \" end diversion
.sp
He that is slow to anger is better than the
mighty; and he that ruleth his spirit than he
that taketh a city.
.br
.sp
.X                \" print the paragraph diverted to X
```

The new command here is the *divert* primitive **.di**. The command **.di X** tells **nroff** to divert the text that follows into macro **X**; the matching **.di** with no argument marks the end of the diversion.

The break is necessary before the end of the diversion because **nroff** diverts *processed* text into the macro. Without the break, **nroff** would not divert any partially filled line in its buffer to **X**; the last few words of diverted text might not form a complete line in the buffer, so **nroff** might not divert them. However, if you break the input before you end the diversion, **nroff** will also divert those last few words.

As you saw earlier, the **.br** command must be used to flush that environment's buffer before switching environments.

The next example, **ex14.r**, illustrates a similar point.

```
.br               \" clear buffer
testword          \" put 'testword' into buffer
.di X             \" divert to X
Piracy, n. Commerce without its folly-swaddles,
just as God made it.
.br               \" divert last line
.di               \" end diversion
.X                \" print text in X
```

Here **nroff** diverts **testword** into **X** along with the text between **.di X** and **.di**. Why did this happen? The command **.di X** does *not* cause a break. Because you did not pass to a new environment in this example before you diverted, **nroff** formed the diversion text in the same buffer in which it stored **testword**. You did not break the input, so **nroff** appended the diverted text to **testword**.

To make sure **nroff** diverts only text between **.di X** and **.di** into **X**, do one of the following: If you want to process the diverted text within the current environment, empty the buffer by inserting the **.br** command before you start the diversion. If you switch to a new environment before starting the diversion, flush the buffer for the new environment before you begin to process diverted text.

Diverting processed text into a macro that already holds material will erase whatever had already been stored there. In some cases, such as the footnote example, you need to append information into the same macro. The *divert and append* variation **.da** of the diversion construction allows you to do so. The following example, **ex15.r**, demonstrates this command:

```
.ll 3i              \" set line length
.po 2i              \" set page offset
.de PP              \" define paragraph macro
.br
.sp 1
.ti 0.5i            \" indent first line 1/2 inch
..
.di X               \" divert the following into X
.PP
Litigation, n. A machine which you go into as a pig
and come out of as a sausage.
.br
.di                 \" end diversion
.X                  \" print what is in X
.br
.da X               \" divert and append material into X
.PP
Inventor, n. A person who makes an ingenious arrangement
of wheels, levers and springs, and believes it
civilization.
.br
.di                 \" end diversion
.X                  \" print what is now in X
```

In this example, you first diverted a single paragraph into the macro **X**. **nroff** stored in **X** the *processed* paragraph; in other words, the command line **.PP** is *not* stored in **X**; its *output* is. When you invoke **X** with the command line **.X**, **nroff** takes the processed text in **X** as input. To **nroff,** there is no difference between processed text and unprocessed text as input: it processes the contents of **X** in the current environment, just like any other text. Therefore, **nroff** processes diverted text *twice*: first when it stores the text within the macro, and again when you invoke the macro.

The fact that **nroff** processes diverted text twice can cause problems if you are not careful. Fortunately, nothing strange happens in the example above. You store a processed paragraph with lines three inches long in **X**. When you invoke **X**, the line length is three inches. Because each line in **X** is already exactly three inches long, nothing happens to it when reprocessed; the layout of the output paragraph is unchanged.

But now, consider what happens in the following example, **ex16.r**:

```
.ll 3i              \" set line length
.po 2i              \" set page offset
.de PP              \" define PP macro
.sp 1
.ti 0.5i            \" indent first line 1/2 inch
..
.di X               \" divert following into X
.ev 2               \" push environment 2
.ll 4i              \" set line length to 4 inches
.PP
Justice, n. A commodity which in a more or less
adulterated condition, the State sells to the
citizen as a reward for his allegiance, taxes
and personal service.
.br
.ev                 \" pop environment (return to ev 0)
.di                 \" end diversion
.X
```

A pargraph processed in environment 0 in this example has three-inch lines; you want your diverted paragraph to have four-inch lines. However, when you print the diverted paragraph with the command line **.X**, what happened?

nroff did *not* print four-inch lines. The four-line text lines set in environment 2 were reprocessed into three-inch lines when the diversion macro is called in environment 0.

There are two ways to prevent such disasters. First, if you wish to invoke **X** in the main environment, use no-fill mode:

```
.nf                \" begin no-fill mode
.X
.fi                \" return to fill mode
```

In no-fill mode, **nroff** outputs lines of input exactly as it receives them, so it keeps four-inch lines four inches long and does not change the format of the diverted text. The second strategy is to return to environment 2 and then invoke **X**; again, the format of the diverted paragraph does not change, because the line length in environment 2 is four inches.

```
.ev 2              \" push environment 2
.X
.ev                \" restore original environment
```

A Footnote Macro

The footnote macro that follows does not print notes at the bottom of each page; rather, it prints everything at the end of the chapter. In the processed text, number register **Fn** is used to keep track of the footnote number; the footnote number will be printed in square brackets where the footnote originally appeared in the text.

Type this macro into the file **ex17.r**. If you wish to use it in your text processing, transfer it to the directory **/usr/lib** under the name **tmac.fn**. Then, whenever you wish to use this macro, be sure to include the option

```
-mfn
```

when you invoke **nroff**:

```
.de FN             \" define macro FN
[\\n+(Fn]          \" print footnote no. in main text
.ev 1              \" push environment 1
.da Z              \" divert and append following into Z
.sp
\\n(Fn. \\$2, \\fI\\$1\\fR,
    \\$3, \\$4.    \" format & print footnote in Z
.br                \" flush diversion buffer
.di                \" end diversion
.ev                \" pop environment (return to ev 0)
..
```

Note that requests to change fonts are preceded by double backslashes, because they are within a macro. The change to the italic font prints the first macro argument, which should be the title of the work, in italics. Number register **Fn** contains the number of the last footnote; you should initialize it with the command

```
.nr Fn 0 1
```

As shown above, each footnote entry in your text should have four arguments. In your input text, each footnote will look like this:

```
.FN "Personal narrative of a pilgrimage to\
El-Medinah and Mecca" "Richard F. Burton"\
London 1856.
```

When you print the diversion **.Z** at the end of the chapter, each footnote will be laid out as follows:

8. Richard F. Burton,
Personal narrative of a pilgrimage to
El_Medinah and Mecca,
London, 1856.

Command Line Options

In the previous sections, you learned how to control **nroff** by including *commands* in the input along with the *text.* You can also supply information in another way: on the command line you type to call **nroff**. Unlike the commands discussed above, this information is *not* part of the script you input into **nroff**.

You already know about some simple **nroff** command lines. For example, the command

```
nroff
```

forces **nroff** to accept input from the keyboard (sometimes called the *standard input)* and print output on the terminal (the *standard output).* Type **<ctrl-D>** (that is, hold down the **ctrl** key and type **D**) to exit from **nroff** if it is reading input from your terminal.

The command line

```
nroff script1.r
```

forces **nroff** to take accept input from the file **script1.r** instead of from your terminal, while the command

```
nroff -ms script.r
```

processes **script1.r** with the **-ms** macro package. You can also redirect **nroff** output to another file **target:**

```
nroff -ms script1.r >target
```

The general form of the **nroff** command line is:

nroff *[option ...] [file ...]*

This means that the command line consists of the **nroff** command, followed by zero or more *option*s, followed by zero or more *file*s. **nroff** processes each named *file* and prints the result on the standard output (the terminal, unless redirected). If no *file* argument is given, as in the first example above, **nroff** reads from the standard input.

Each *option* on the command line must begin with a hyphen '-' to distinguish it from a *file* specification. Using **nroff** with the **-ms** macro package is one example of entering an option. In general, the **-m** option takes the form

-**m***name*

which means the option consists of the characters **-m** immediately followed by a *name.* This tells **nroff** to process the macro package found in the COHERENT file

/usr/lib/tmac.*name*

For example, the **ms** macro package discussed in chapter 2 is in the file **/usr/lib/tmac.s,** whereashe **man** macro package used for the **man** command and to process manual pages is in the file **/usr/lib/tmac.an.**

Any macro packages that you customize for your own use should be stored in the directory **/usr/lib** under such a name if you wish to use them with the **-m***name* option.

The **-i** option tells **nroff** to read input from the standard input after processing each given *file.* This allows you to supply additional input interactively from your terminal.

The **-x** option tells **nroff** not to move to the bottom of the last output page when done. This is especially useful if you want to see the output on the screen of a CRT terminal.

The -**n***N* option sets the page number of the first output page to the number *N,* rather than starting at page 1. This is useful for processing large documents with input text in several files which **nroff** processes separately.

The -**r***XN* option sets the value of number register *X* to *N.* This option lets you initialize number registers when you invoke **nroff**.

The COHERENT system provides many useful features which can be helpful while you are using **nroff.** In particular, you can use a number of special characters. The *stop-output* and *start-output* characters, usually **<ctrl-S>** and **<ctrl-Q>,** stop and restart output on your terminal. The *interrupt* character, usually **<ctrl-C>**, interrupts program execution; you can use it to stop an **nroff** command if you typed the command line incorrectly. The *kill* character, usually **<ctrl-**\>, also terminates program execution. Some COHERENT systems use different characters than those mentioned above; consult *Using the COHERENT System* for details.

For Further Information

The Lexicon entry for **nroff** summarizes its primitives, dedicated number registers, escape sequences, and command-line options.

For example, this manual was typeset by COHERENT **troff**. The program **troff** also performs text formatting. Unlike **nroff**, however, **troff** produces proportionally spaced output that can be printed on printers that support the Hewlett-Packard Page Control Language (including the LaserJet and DeskJet families of printers) or on any printer that implements the PostScript page-control language. This manual (including the positioning of the fancy capital

letters and ornaments) was typeset by **troff** under the COHERENT system. See the Lexicon entry for **troff** for details on how to use this command.

The Lexicon also has entries for two macros packages that are included with the COHERENT system: **man** which produces manual pages similar to those that appear in the Lexicon; and **ms**, which performs formatting somewhat similar to that seen in this tutorial. You will find that these two packages already perform practically all of the formatting tasks that you will commonly need to do.

The error messages generated by **nroff** are given in the appendix at the rear of this manual.

UUCP, Remote Communications Utility

UUCP is a set of programs that together let you communicate in an unattended manner with remote COHERENT and UNIX sites. The term *UUCP* is an abbreviation for "UNIX to UNIX copy"; as its name implies, UUCP was developed under the UNIX operating system.

UUCP allows your COHERENT system to talk to other computers that also run COHERENT or UNIX. It can transmit files and mail to other systems and receive material from them, without needing you to guide it by hand every step of the way. Moreover, you can instruct UUCP to telephone other computers at the same time each day; this permits regular, orderly exchange of mail, news, and files among computers, and allows you to take advantage of lower telephone rates during off-peak periods. In a similar fashion, UUCP allows other systems to log into your system, to exchange mail or other information, and otherwise perform useful tasks.

Numerous UUCP systems have linked together to create an informal network called the *Usenet*. Many megabytes of source code, news, and technical information are available across the Usenet. Anyone who is connected to the Usenet can exchange mail with anyone else who is also connected to the Usenet. All that is required to hook into the Usenet is to obtain a UUCP connection to anyone else who is connected to the Usenet.

You can use UUCP only if you have telephone access to another computer that runs UUCP, and if your system and the remote system with which you wish to communicate have been described to each other. UUCP is standard with COHERENT and UNIX, and can be purchased for MS-DOS. If you wish to copy files from another system, you must arrange with the system administrator of that system before you can begin to use UUCP. Likewise, if you want someone else to dial into your system to upload or download files, you must first describe that system to your copy of UUCP.

Contents of This Tutorial

This tutorial describes UUCP and tells you how to set up and run your UUCP system. It contains the following sections:

- An overview of UUCP.
- How to set up your modem to dial out.
- How to set up UUCP to contact **mwcbbs**, the Mark Williams bulletin board.
- How to use the COHERENT utility **uuinstall** to set up UUCP to contact **mwcbbs**.
- How to set up your UUCP system to accept calls from remote systems.
- How to use the UUCP utilities to exchange files and mail with remote systems.
- How to debug some common problems with UUCP.
- How to administer your UUCP system.
- A brief introduction to networks.

Try as we might, there is no way to present all of UUCP in a brief tutorial. If you wish to explore the heights and depths of UUCP, we urge you to acquire the following books:

- O'Reilly, T.: *!%@:: A Directory of Electronic Mail Addressing and Networks.* Sebastapol, Calif, O'Reilly & Associates Inc., 1989.
- O'Reilly, T.; Todino, G.: *Managing UUCP and Usenet.* Sebastopol, Calif, O'Reilly & Associates Inc., 1987.
- Krol, E.: *The Whole Internet: User's Guide & Catalog.* Sebastopol, Calif, O'Reilly & Associates Inc., 1992. *Highly recommended.*
- Seyer M.D.: *RS-232 Made Easy: Connecting Computers, Printers, Terminals, and Modems.* Englewood Cliffs, NJ, Prentice-Hall Inc., 1984.

An Overview of UUCP

UUCP is a set of programs that exchange files with other computers that run UUCP. You can set aside files or mail messages to be transferred to another computer; UUCP regularly checks to see if material has been set aside to be transferred, dials the remote system, and exchanges the files without requiring your assistance.

This appears to be a simple function, but it can be extremely useful to you. Suppose, for example, that you run a real-estate office that is a member of an organization with regional and national offices. You can tell UUCP to call your regional office each night, to send a file of your new listings and to accept a file of new listings in your district that had come from other local offices. Likewise, your association's regional office can telephone the national office each night to receive new listings in your region, which can then be passed on automatically to the appropriate neighborhood offices. All of this information can be transferred at night, when telephone rates are lowest, and without needing you to be at the console. When you come to work the next morning, you will have the latest listings instantly available on your terminal.

In brief, what UUCP offers is the ability to join a *network* of computers, in which every user of every computer can exchange information with every user on every other computer, automatically. What computer networks can do is limited only by your need to exchange information with other computer users, and by your imagination.

Implementations of UUCP

This version of COHERENT implements **Taylor UUCP**, a UUCP package written be Ian Taylor (**ian@airs.com**) with numerous contributions from other people in the USENET community. COHERENT does not implement the full suite of UUCP utilities provided with the Taylor UUCP package; however, it does implement the enough utilities for you to set up a robust UUCP site on your system.

The source code for the full Taylor distribution is available via **ftp** from various sources, and can be downloaded from **mwcbbs**. For a description of the full package, see Ian Taylor's documentation for his package, which is included with your COHERENT package.

This chapter presents examples to help you learn how to set up a simple UUCP configuration for a remote site. It does not discuss the more exotic features of UUCP; however, the information given in this chapter is sufficient for you to set up communications with most remote sites.

For more background information on the (sometimes arcane) subjects of serial ports and modems, see the Lexicon entries for **UUCP**, **modem**, **asy** (the serial-port driver), and **RS-232**. These discuss aspects of serial communication, and point you to other articles that you may find helpful.

Programs

The UUCP system uses the following programs to do its work:

uucico Call remote systems: log into the remote system and transfer files.

uuconv Convert configuration files into Taylor format. You will use this only if you are porting your system from another implementation of UUCP.

uucp UNIX-to-UNIX Copy: copy files from one computer to another. Be sure not to confuse the command **uucp** with the UUCP system, despite their similar names. (Note that the name of this utility is retained for the sake of convenience, and because the abbreviation of the phrase COHERENT-toCOHERENT Copy would remind users of the late Soviet Union.)

uuencode Translate binary files into printable ASCII characters for transmission to another system.

uudecode Translate files encoded by **uuencode** back into object code.

uuinstall This program displays a template on your screen, and helps you describe a system to UUCP relatively painlessly.

uulog Read the UUCP logs, which record what UUCP does.

uumkdir Create a directory for a remote site. This command is invoked by **uuinstall** and other programs.

uumvlog Copy the current UUCP log files into backup files. Throw away all log files older than a requested number of days. UUCP logs everything that it does; and since it does a lot, its log files can grow very large very quickly. **uumvlog** ensures both that you have enough information on your system to diagnose problems with uucp, and that the UUCP log files do not overwhelm your system.

uuname List the systems that your computer can reach.

uupick "Pick up" files that have been uploaded to your system from a remote site.

uusched This is a script which invoke the command **uucico** to call all systems that have jobs waiting for them.

uuto This is a script that invokes the command **uucp** to copy files from your system to another system via UUCP.

uutouch Create a file that triggers a call to a named remote system.

uutry Force a call to a remote system, for debugging purposes.

uux Execute a command on a remote system.

uuxqt Check directory **/usr/spool/uucp/***sitename* and execute all files therein that have the prefix "X."

Two other programs, while not part of UUCP *per se*, are used by it:

ttystat Check the status of your asynchronous ports. If UUCP is not receiving files from other systems or not sending files to other systems, it may be because the appropriate ports have not been enabled.

mail Send "electronic mail" to another person, either on your system or on another system via UUCP.

Files and Directories

As mentioned earlier, your system can use UUCP to contact many different remote sites, and can have many different sites contact it. Each site differs from all others in many respects: by its name; by the telephone number at which you call it; by the permissions it may grant you and you may grant it; by the day of the week and the time of day during which you may wish to call it; and by the procedure you must follow to log into it. Remote sites may also differ with regard to the port by which you contact them; the manner in which you contact it (direct connect or via modem); the protocol with which you exchange files; and the name by which your system identifies itself to the remote system.

As you can see, UUCP needs a considerable amount of information before it can communicate with a remote site. UUCP reads this information from data files. The processing of setting up communicate with a remote system means that you write the correct information about that site into each of the appropriate UUCP data files. This process will be confusing at first — in part because some of the notation is rather obscure, and in part because there's simply a lot of it, and some of the information needed may touch on aspects of your system about which you may not know very much.

Each implementation of UUCP has its own suite of data files that you must manipulate to set up a remote site. UUCP uses the following files and directories:

/etc/domain
This file lists the UUCP domain. It is read by **mail**.

/etc/uucpname
Holds the name of your system, as it is known to other UUCP sites.

/usr/bin/uucp
The **uucp** command. Copy a file to another system that runs UUCP.

/usr/bin/uulog
The **uulog** command.

/usr/bin/uuname
The **uuname** command.

/usr/bin/uudecode
The **uudecode** command.

/usr/bin/uuencode
The **uuencode** command.

/usr/bin/uupick
The **uupick** command.

/usr/bin/uuto
The script **uuto**.

/usr/lib/uucp
Contains UUCP commands and system data files.

/usr/lib/uucp/dial
This file tells **uucico** how to dial the modems on your system.

/usr/lib/uucp/port
This file describes the devices that **uucico** uses to call each remote UUCP site.

/usr/lib/uucp/sys
This file describes the remote UUCP sites that your site can call, or that can call your site. The command **uucico** (the command that actually talks with remote systems) reads the information in this file to to connect to remote systems. This file also names the directories on your system into which a each remote site may read or write files; and names the protocol or protocols that **uucico** uses to exchange files with a given remote system.

/usr/lib/uucp/uucico
The **uucico** command.

/usr/lib/uucp/uuconv
The **uuconv** command.

/usr/lib/uucp/uumkdir
The **uumkdir** command.

/usr/lib/uucp/uumvlog
The **uumvlog** command.

/usr/lib/uusched
The script **uusched**.

/usr/lib/uucp/uutouch
The **uutouch** command.

/usr/lib/uucp/uutry
The **uutry** command.

/usr/lib/uucp/uuxqt
The **uuxqt** command.

/usr/spool/logs/uucp
Log of UUCP activity.

/usr/spool/uucp/.Admin/audit.local
uucico stores logging information in this file when you invoke it with with logging specified.

/usr/spool/uucp/.Admin/xferstats
This file stores the transfer rates of files received or transmitted.

/usr/spool/uucp/.Log
Directory containing UUCP logfiles, as follows:

/usr/spool/uucp/.Log/uucico/*sitename*
/usr/spool/uucp/.Log/uux/*sitename*
/usr/spool/uucp/.Log/uucp/*sitename*
/usr/spool/uucp/.Log/uuxqt/*sitename*

/usr/spool/uucp/*sitename***/TM***
/usr/spool/uucp/.temp/*sitename***/***
These are temporary files that **uucico** generates when receiving files.

/usr/spool/uucp/*sitename***/C.***
Files that instruct the local system either to send or to receive files.

/usr/spool/uucp/*sitename***/D.***
Work files for outgoing and incoming files.

TUTORIALS

/usr/spool/uucp/LCK.*
The "lock" files UUCP uses to coordinate its resources. When a UUCP program attempts to access a remote site, it writes a "lock" file for that site. This is to prevent UUCP from accidentally attempting to access the same site more than once simultaneously. When the program that wrote the lock file exits successfully, it erases its lock files, and so makes that site accessible to other UUCP programs.

/usr/spool/uucp/.Sequence
This directory contains the sequence number of the last file handled by UUCP.

/usr/spool/uucp/*sitename***/X.***
Executed files. These files will be executed by the command **uuxqt**, and are generated by a remote system.

/usr/spool/uucppublic
Public directory accessible by all remote UUCP systems.

Attaching a Modem to Your Computer

You can use UUCP to network computers that are within the same office or the same building. It is far more common, though, to use uucp to connect computers that are far apart via modem. This tutorial assumes that you will be using uucp to exchange files via modem.

If you have not yet attached a modem to your computer, this section will give you some useful hints. It is straightforward to attach a modem to your computer, but you must be careful.

First, read the documentation that comes with your modem, and look for (1) the baud rate at which the modem operates, and (2) the command protocol that your modem uses.

Second, check the plug on the back of your modem. The modem will connect to your computer via a nine-pin or 25-pin D plug, also known as an "RS-232 interface". Such a plug can be either *male* or *female*: the male plug has nine or 25 small pins projecting from it, whereas the female does not.

Due to what can only be termed extreme stupidity, IBM AT and AT-compatible computers use RS-232D plugs for both serial and parallel ports. *Be sure to plug your modem into a serial port, not the parallel port, or you can damage your computer and your modem!*

Third, obtain a cable to connect one of the serial ports on your computer to the modem. The serial ports on an IBM AT or AT-compatible computer are almost always male. If your modem has a female plug, you will need a male-to-female cable, whereas if your modem's plug is male (which is very rare), you will need a female-to-female plug. Be sure to purchase a standard modem cable for an IBM AT; practically every computer dealer carries them. The cable you purchase should support "full modem control"; if it does not say on the package, be sure to ask your dealer before you buy it. If you are handy with a soldering iron you may be able to solder up such a cable for yourself, but unless you know precisely what you are doing it probably is not worth the trouble.

The Lexicon entry **RS-232** contains pinouts for both nine- and 25-pin connectors. When you plug in your cable, be sure to note whether you plugged it into port **com1**, **com2**, **com3** or **com4**.

Fourth, reconfigure the serial port to suit your modem. This involves the following steps:

1. Log in as the superuser **root**.
2. Edit the file **/etc/ttys**. This file normally has several lines in it, one that describes the console and one for each serial port. Each line has four fields: a one-character field that indicates whether a login prompt should be displayed (used only for devices from which people will be logging into your system); a one-character field that describes whether the device is local or remote (a local would be a modem from which you wished to dial out, a remote device would be a modem from which someone could dial in); a one-character field that describes the speed (or baud rate) at which the device operates; and a field of indefinite length that names the device being described.

   If you have plugged into serial port **com1** a 9600-baud modem that will allow remote logins, edit the line for **com1** to read as follows:

   ```
   1rPcom1r
   ```

   If you have plugged into serial port **com2** a 2400-baud modem from which you are only going to dial out, edit the line for **com2** to read as follows:

   ```
   0llcom2l
   ```

   Note that the second and last character are a lower-case L, not a one. For more information, see the Lexicon

entries for **ttys**.

3. Test if you have connected your modem. Turn on your modem; then log in as the superuser **root** and type the following command:

```
echo "foo" >/dev/com?l
```

where *?* is the number of the port. If you are addressing the correct port, the lights on your modem should blink briefly. For a more sophisticated test, try to communicate with your modem by using the command **ckermit**. If you are not familiar with **ckermit**, see its entry in the Lexicon for details.

4. When you have finished editing **/etc/ttys**, type the following command:

```
kill quit 1
```

This forces COHERENT to read **/etc/ttys** and set up its ports in the manner that you have configured them.

If you continue to have problems making connections with your modem, see the volume *RS-232 Made Easy*, referenced above. It describes in lavish detail how to connect all manner of devices via the RS-232 interface. also check the Lexicon articles **modem** and **RS-232** for helpful information.

Selecting Site and Domain Names

The first step to setting up UUCP is to select a site name for your system. You probably did this already when you installed COHERENT on your system, because the COHERENT mail system does not work unless you have named your system. If, however, you have not yet named your system, you can do so by editing the file **/etc/uucpname**. The name you select must have eight characters or fewer, and must be unique — or unique, at least, to the system into which you will log in. Avoid names taken from popular culture, such as "calvin," "hobbes," or "arnold" — these have already been used many times. See the Lexicon entry **uucpname** for more details.

Next, select a domain name for your system. Again, you probably did this when you installed COHERENT. A *domain* is a set of UUCP systems that together form one group with a common name. Even if you do not belong to a domain, you must set a domain for your system, because **mail** expects it. By convention, you can use your site name plus the suffix **.UUCP** to create a domain. The domain name is written into file **/etc/domain**. See the Lexicon entry **domain** for details.

You must edit **/etc/uucpname** and **/etc/domain** to install these names.

Set Up a UUCP Site by Hand

This chapter walks you through the setting up of a typical remote site, and explains what it's doing (and why) at each step of the way. We hope that when you have finished reading it, you will grasp at least the principles of how UUCP works.

Setting up UUCP to call a remote system can be confusing and difficult, mainly because there are many points at which this task can fail. However, with patience and with the cooperation of the administrator of the system that you will be contacting, you can accomplish this task. Fortunately, once you have succeeded in exchanging files with another system, your connection should work indefinitely without needing modification. Fortunately, too, UUCP is designed in such a way that you can reuse system descriptions; so over time this process should become easier.

To set up UUCP so it can contact another system, you must enter information into files **/usr/lib/uucp/sys**, **/usr/lib/uucp/dial**, and **/usr/lib/uucp/port**. **uucico** reads these files to learn how to communicate with a given remote system.

The rest of this section describes how to configure your system so it can communicate with **mwcbbs**, Mark Williams Company's bulletin board. A later section describes how to use UUCP's commands and utilities to work with a remote system once you have established communications.

port: Describe a Serial Port

To call a given remote system, UUCP must know about the devices that it uses to communicate with that site. File **/usr/lib/uucp/port** describes ports that **uucico** can use.

Before you proceed any further, answer the following questions:

1 What serial device will be used for communications? For example, the port **/dev/com2l**.

2 At what speed will communication take place?

3 What name will you give to this port?

4 Does communication via this port involve a modem?

5 If a modem is used, what dialer entry from **/usr/lib/uucp/dial** should **uucico** use?

With the above questions answered, let's put together our **port** entry.

A **port** entry begins with the lines that names the port. Because **mwcbbs** is the system to be called for the purposes of this example, let's call this port **MWCBBS**. The port entry should look like this:

```
port MWCBBS
type modem
device /dev/com2l
baud 2400
dialer hayes
```

Note that in the **dialer** line of this entry, an underscore was used instead of a normal space character. Do not use spaces in the fields used to name the port, the actual device nor the dialer.

Note also the **type** line of this port entry. It says **modem**. The only other valid value for this line is **direct**, which you would use should you be contacting a system via a wire that runs directly from your serial port to the other system. Because the goal here is to call **mwcbbs**, you must specify **modem**.

dial: Describe a Modem

Because the port described above has a modem attached to it, you must tell **uucico** how to talk to that modem. The following example assumes that you will call **mwcbbs** via a 2400-bps, Hayes-compatible modem. If you are not familiar with sending instructions to a modem, stop here and find the manual for your modem, or find someone familiar with modem communications before continuing.

To write the **dial** entry for your modem, you must answer the following questions. Some of the questions may be unclear at first, but don't worry — we'll get into the details and get you through this:

1 What will you name this dial entry? For example, you could call your 2400-baud Hayes modem **hayes**.

2 What command tells the modem to dial out? Most Hayes-compatible modems use the command **ATDT** (for Touch-Tone telephone lines) or **ATDP** (for pulse-dial lines).

3 What message does the modem return when it connects to the remote system's modem? Most Hayes-compatible modems return the phrase **CONNECT 2400**).

4 What messages does the modem send when it fails to connect to the remote system's modem? Example include **BUSY**, **NO ANSWER**, **NO DIALTONE**, and **NO CARRIER**.

5 What command tells the modem to hang up? Most Hayes-compatible modems use the command **ATH0**.

6 How many seconds do you want to give the modem to make the connection before **uucico** times out and gives up try to connect via this modem?

Some above information is optional, but you should find all of it to write a thorough description of your modem.

An entry in **dial** begins with the line that names the modem. In the earlier example for the **port** file, we decided to use the modem named **hayes**. Therefore, the first line in our example is:

```
dialer hayes
```

Our next step is to write a chat script for the modem. At this point, a short discussion of chat scripts is in order. A *chat script* gives the dialogue (or "chat") that UUCP has with a device or a remote system. It consists of pairs of messages: an *expect* message, which is a prompt that your system expects to receive from the device or remote system, and a *response* message, which is what your system sends in response to it. Spaces separate messages from each other; therefore, you cannot use a space character within a message. Instead, use the escape sequence **\s** to represent a space character. If you want **uucico** to send a message immediately, or to send a response message without waiting for an expect message, use an empty string **""** to represent the expect string.

The next line in our example give the chat script with which **uucico** dials your modem. This is built from your answer to question 2, above. Because a Hayes modem normally does not send a prompt to request a dial command, use an empty string for the expect string. The usual command to dial a Hayes compatible modem is **ATDT** (as described above), followed by the telephone number to dial. The chat script for this dialogue is as

follows:

```
chat "" atdt\D
```

Let's take a closer look at this chat script. It begins with **""**, which tells **uucico** to expect nothing from the modem. **atdt\D** is the message that **uucico** is to send to the modem. **atdt** is the dial command for a Touch-Tone telephone line, as noted above. The escape sequence **\D** represents the telephone number. We use an escape sequence instead of the literal telephone number because you can dial many different remote sites from the same modem. **uucico** finds the telephone number to dial from the entry for **mwcbbs** in the file **sys**, as will be described below.

Although this simple chat script will dial the desired telephone number, it can be improved. For example, **uucico** may fail to connect to the remote system for any number of reasons. The more information we can get back from the modem, the easier it will be to debug any problems we have in connecting to the remote system. If the modem is set up to return verbal result codes, **uucico** can check for these codes to determine if a connection to a remote modem succeeded. (For more information on *verbal result codes*, see the manual that came with your modem.) The following adds these features to our chat script for a Hayes-compatible modem:

```
chat "" ATQ0E1V1L2M1DT\D CONNECT\s2400
```

Look at this chat script carefully. Again, **uucico** expects nothing and sends a command to make the modem dial out. The command **ATQ0E1V1L2M1DT\D** has a lot of information packed into it. It is written mostly in Hayes-ese, so we'll break it up and show you what each portion means:

AT Attention: tell the modem that the following is a command.

Q0 Return result codes.

E1 Echo commands sent to it. (You can tell **uucico** to log what the modem returns, so you can see exactly what it is doing.)

V1 Use the long form of result codes.

L2 Set the loudness of the dial tone to medium.

M1 Turn on the speaker on the modem. Sometimes it is helpful to hear when the modem dials out.

DT Dial Touch-Tone.

\D The UUCP escape sequence that represents the telephone number, as described above.

Now, look at the end of this chat script: we have added a second expect message. This tells **uucico** that after it has dialed the telephone number for this site, it should expect the string **CONNECT 2400**. When **uucico** sees this message, it assumes that a modem connection was successfully established and that it can continue normally. If **uucico** does not see this message within a given period of time (as will be described below) or if it receives different message, it assumes that the connection attempt failed. When this occurs, **uucico** aborts the connection attempt.

Note that, as mentioned earlier, you must use the escape sequence **\s** to represent the space character in the phrase **CONNECT 2400**.

To review, the dial entry for dialer **hayes** now looks like this:

```
dialer hayes
chat "" ATQ0E1V1L2M1DT\D CONNECT\s2400
```

The next line, **chat-timeout**, gives the number of seconds that **uucico** should wait before it quits trying to connect to a remote system. The default is 40 seconds. You may or may not prefer to change this value; but for the purpose of this example, let's assume that you want **uucico** to wait 60 seconds before it times out. The **dial** entry for **hayes** now looks like this:

```
dialer hayes
chat "" ATQ0E1V1L2M1DT\D CONNECT\s2400
chat-timeout 60
```

Recall that our chat script for this modem turned on verbal result codes. Now, it is time to take advantage of this. Hayes compatible modems, will, in general, return any number of messages if a connection fails, depending upon the cause of the failure. The entry **chat-fail** defines a string that the modem returns when it has failed to connect. When **uucico** receives the string, it realizes that the attempt to connect has failed, and quits. The description for a modem can include any number of **chat-fail** entries. Let's add some messages that a typical Hayes-compatible modem might return when it fails to connect to a remote modem. The **dial** entry for the modem named **hayes** now looks like:

```
dialer hayes
chat "" ATQ0E1V1L2M1DT\D CONNECT\s2400
chat-timeout 60
chat-fail BUSY
chat-fail NO\sCARRIER
chat-fail NO\sANSWER
```

Modem-result codes offer many advantages. By looking for specific messages, **uucico** knows immediately when the modem cannot connect to a remote site, and quits immediately rather than waiting to time out. More important, **uucico**'s log files will show why the connection failed, which eases the debugging of connection problems.

We can add yet another safeguard to this dialer entry. By naming **abort-chat** and **complete-chat** scripts, we can tell **uucico** to chat again with the modem after a communication session has ended. These scripts can, for example, ensure that the modem hangs up the telephone and turns off error messages and echoing. If the port for this modem normally is enabled, then it is vital that you turn off echoing and error messages, to prevent the modem and your COHERENT getting into an infinite dialogue when the modem echoes COHERENT's login prompt. For details on this problem, see the Lexicon entry for **modem**. The following adds **abort-chat** and **complete-chat** scripts to our description of **hayes**:

```
dialer hayes
chat "" ATQ0E1V1L2M1DT\D CONNECT\s2400
chat-timeout 60
chat-fail BUSY
chat-fail NO\sCARRIER
chat-fail NO\sANSWER
complete-chat "" +++ OK ATH0E0V0Q1
abort-chat "" +++ OK ATH0E0V0Q1
```

Notice that the two chat scripts are identical. **uucico** runs this script whenever a session ends — regardless of whether it failed or completed successfully. As with the **chat** script, the first entry is an empty string, to indicate that **uucico** should expect nothing before it sends its first response message — in this case the string **+++**, which in Hayes-ese is the escape sequence that puts the modem into command mode. The modem replies with the message *OK*. When **uucico** sees **OK**, it sends the reply **ATH0E0V0Q1**, which is the Hayes command sequence to (1) hang up the telephone, (2) turn off echoing, (3) turn off verbose error messages, and (4) turn off error messages altogether.

Please note that the last six lines of this dialer entry are optional. The only required lines are the first two.

sys: Individual System Configuration

Having completed the **port** and **dial** entries for calling **mwcbbs**, the last — and most difficult — step is to describe **mwcbbs** to **uucico**. This is done by writing an entry in file **/usr/lib/uucp/sys**. The entry does the following:

- Names the remote system.
- Specifies valid times to call the system.
- Specifies telephone number to use when calling the system.
- Specifies valid protocols to use to exchange files with the system.
- Restricts read and write access for the remote system.

Before we continue, please note that the following example is typical of an entry in **sys**, but it is by no means exhaustive. Please refer to the Lexicon article **sys** for more information.

Now, be prepared to answer the following questions:

1 What is the name of the remote system?

2 When is it legal for **uucico** to telephone the remote system?

3 At what speed will communications take place?

4 On which port will the call to the remote system be made?

5 What telephone number, if any, must be dialed?

6 What is the "chat sequence" (chat script) to be used to log into the system?

7 What protocol should be used to transfer files?

8 How should your system identify itself to the remote system?

An entry in **sys** must begin with the line **system** *sysname*, where *sysname* is the name of the remote system. This both names the remote system and tells **uucico** that a new entry is beginning. Make this entry into **sys**:

```
system mwcbbs
```

The next line of the entry, **time**, restricts when the **uucico** may call the remote system. You can restrict calls to a given day of the week by using the following abbreviations:

Wk	Every weekday (Monday through Friday)
Su	Sunday
Mo	Monday
Tu	Tuesday
We	Wednesday
Th	Thursday
Fr	Friday
Sa	Saturday

You can restrict the time of contact; all time notation must be in military time. The following give some examples of day/time entries:

0100-0200	Valid to call every day between 1 and 2 AM
Mo0100-0200	Valid to call every Monday between 1 and 2 AM
Sa	Valid to call at any time on every Saturday
Any	Valid to call at any time
Never	Never call the system

To restrict the valid times for calling **mwcbbs** to Saturday and Sunday nights from 10 to 11:15 PM, add the line:

```
time SaSu2200-2315
```

Our **sys** entry now looks like this:

```
system mwcbbs
time SaSu2200-2315
```

The next line of the **sys** entry, **baud**, is self-explanatory and is answered by the third question above. For this example, we will assume a speed of 2400 bps. The **sys** entry now look like:

```
system mwcbbs
time SaSu2200-2315
baud 2400
```

The next line, **port**, names the port that **uucico** is to use to telephone the remote system. This must name a port that is defined in file **port**; in this case we'll use port **MWCBBS**, which we wrote earlier:

```
port MWCBBS
```

The next line, **phone**, is answered by question 5, above; this, too, is self-explanatory. Add the following line to our **sys** entry:

```
phone 17085590412
```

uucico uses this telephone number to expand the escape sequence **\D**, used in the **dial** entry defined above. Note, by the way, that you can access **mwcbbs** through the following three telephone numbers:

1200/2400-baud generic modem:	708-559-0412
9600-baud Trailblazer modem:	708-559-0445
9600-baud V.32 or HST modem:	708-559-0452

This example assumes that you have a generic 2400-baud modem; but you should select the number that best suits your modem.

The next line, **chat**, is the chat script that **uucico** uses to log into the remote system. Please refer to the previous section about the **dial** file for a brief description of what a chat script is and how it is laid out; or see the Lexicon entry for **sys**:

```
chat "" \n in:--in: nuucp word: public word: serialnum
```

Please note that the string *serialnum* represents the serial number of your COHERENT system. You must use this

number to log into system **mwcbbs**.

Let's take a moment to review this line. By the time **uucico** needs the information from this line, it has already successfully dialed into and connected to one of the modems on **mwcbbs**. Now it must log into the system. The chat script tells **uucico** to expect nothing and to immediately send out a newline, as represented by the escape sequence **\n**.

After **uucico** sends the newline character, it waits for **in:**, i.e., the final characters in the **mwcbbs** login prompt **mwcbbs 386 login:**. It is not necessary to expect the entire login prompt, just enough of it to let **uucico** know that it can send the next message — in this case, your system's login identifier. If **uucico** does not receive **in:**, it sends another newline character and again waits for **in:**. (**--in:** is equivilant to **-\n-in:**).

Once it receives **in:**, **uucico** sends **nuucp**, which is the login name that **mwcbbs** expects to receive from your system. **uucico** now expects the message **word:** which is the is the tail of the prompt **mwcbbs** really transmits, **password:**. When **uucico** sees **word:**, it sends **public**, which is the password that **mwcbbs** expects. Once **mwcbbs** receives the password it expects, it sends out a prompt for a remote-access password — hence, this chat script again expects to see **word:** again. When it receives this second password prompt, this chat script tells **uucico** to send your system's serial number. This completes the chat script.

The **sys** entry for **mwcbbs** now looks like this:

```
system mwcbbs
time SaSu2200-2315
baud 2400
port MWCBBS
phone 17085590412
chat "" \n in:--in: nuucp word: public word: serialnum
```

Now, we will add the **protocol** line to this entry. This tells **uucico** which protocol to use when exchanging files with the remote system. Different implementations of UUCP use various protocols for exchanging files. The **g** protocol was the first UUCP protocol to be invented, and is still the most commonly used. (By the way, this protocol is named after its designer, Greg Chesson.) Since then, other protocols have been invented. Each protocol has its strengths and weaknesses; you should weigh carefully how you are communicating with the remote system and what protocols the remote system, supports before you select a protocol. Please refer to the Lexicon article **sys** for a complete list of the protocols that the Taylor UUCP package recognizes, and for information on when each should be used.

Please note, too, that the **g** protocol is not implemented uniformly by all versions of UUCP. For instance, some **g** protocols can only send data in packets of 64 bytes, using three *sliding windows*. Other implementations can support up to 4,096 bytes per packet, using seven sliding windows. The more windows supported, the greater that transfer rate of data. The larger the packet size per window, the greater the transfer rate of data. For a detailed discussion of the internals of UUCP protocols, please refer to third-party publications or to documentation for the Taylor UUCP system in the file **/usr/src/alien/uudoc.tar.Z**. Taylor UUCP can be configured from as little as three windows and 64 bytes per packet, to seven windows and 4,096 bytes per packet. You can set these parameters in the **sys** file; for details, see the Lexicon entry for **sys**. In this example, we will use the default values of seven windows and 64-byte packets.

After all of this discussion, add the following line to the **sys** entry we are building:

```
protocol g
```

For the next step, you must select the name by which your system identifies itself to **mwcbbs**, asked in question 8, above. "But wait!" you say. "Didn't I answer this in the chat script that we labored over just earlier? We identified ourselves as 'nuucp'." Well, not exactly, as we will make clear.

The login name of **nuucp** is not the same as the site name with which we wish to identify ourselves. Once our system has logged into the remote system (in this case, **mwcbbs**), the remote system fires up **uucico** on its end. Once again our system must identify itself — but this time to **uucico**, not to the login program.

Think of the login sequence this way: as a normal user must log in giving his name and a password or two just to run a shell, so to must **uucico** log into **mwcbbs**. It logs in with the name **nuucp** and gives the necessary passwords; however, **mwcbbs**, instead of invoking a shell for user **nuucp**, invokes **uucico** instead. (If you look in the file **/etc/passwd**, you'll see that the last entry for each user is the program that the system runs for that user; usually it is a shell, but sometimes it's another program, e.g., **uucico**). Now, **uucico** on **mwcbbs** begins to talk with **uucico** on your system. The real work of transfering files can begin, as long as **mwcbbs** recognizes the sitename with which your system identifies itself.

mwcbbs does not recognize many remote systems, yet it receives calls from more than 100 different systems per day. How does **mwcbbs** handle all of these calls if it really only knows a few dozen systems? Most systems identify themselves by the same name, **bbsuser**. **mwcbbs** is set up to always recognize the remote site **bbsuser**.

Where does this name come from? In normal circumstances, this name is read from the file **/etc/uucpname**. This file holds the name you gave your system when you installed COHERENT; you were required to select a name because electronic mail will not work without it. (For more information on this file, see its entry in the Lexicon.) The odds are that you did not choose the name **bbsuser**. Let's say that when you installed COHERENT onto your system, you chose the name **foobar**. Now, when you system calls **mwcbbs**, by default it will identify itself to **mwcbbs**'s edition of **uucico** as **foobar**. However, **mwcbbs** doesn't know **foobar** from Adam, so it logs your system off and hangs up the telephone.

To get around this, you must insert the line **myname** into the **sys** entry for **mwcbbs**. Add this line to the sys entry we are building:

```
myname bbsuser
```

Now, whenever your system calls **mwcbbs**, it will identify itself to **mwcbbs**'s **uucico** as **bbsuser**.

At this point, we could fill dozens of pages with discussions of the items you can configure in **sys**. However, we now have all of the information we need to call **mwcbbs**. For a fuller discussion of **sys**, look up its entry in the Lexicon.

For now, the completed **sys** entry for **mwcbbs** is as follows:

```
system mwcbbs
time SaSu2200-2315
baud 2400
port MWCBBS
phone 17085590412
chat "" \n in:--in: nuucp word: public word: serialnum
protocol g
myname bbsuser
```

There are a few other items that you will find yourself configuring as part of a typical entry in **sys**

First, you must indicate whether the remote system and request files from your system, and transfer files into it. You may wish to deny this capacity to some remote systems, but you do want to grant it to **mwcbbs**, as the whole point of access that system is to have it download files to you. So, add the following two lines to the bottom of the entry for **mwcbbs**:

```
request yes
transfer yes
```

Next, you must name the directories on your system from which the remote system can request files, and the directories on the remote system onto which your system has permission to write files. By default, remote systems are limited to requesting files from directory **/usr/spool/uucppublic** and its subdirectories. To change this default, add the instructions **remote-send** and **remote-receive** to the entry in **sys** for the remote site. For example, suppose you decided to let **mwcbbs** read the directory **/usr/private**, but not **/usr/private/myfiles**. You also decided to let **mwcbbs** write files into **/tmp**, but not into directory **/tmp/secret**. To do this, add the following instructions to the **sys** entry for **mwcbbs**:

```
remote-send /usr/private !/usr/private/myfiles
remote-receive /tmp !/tmp/secret
```

Naming a directory in **remote-send** or **remote-receive** lets the remote system, respectively, read from that directory and all of its sub-directories. However, if you prefix a directory name with an exclamation point '!', that directory and its subdirectories are specifically excluded from being accessed by the remote system. The following gives the normal settings for these directories:

```
remote-send /usr/spool/uucppublic /tmp
remote-receive /usr/spool/uucppublic /tmp
```

Next, you must name the directories from which your system can send files to the remote system, and into which your system can write the files that you have requested from the remote system. These are named by, respectively, the instructions **local-send** and **local-receive** The following gives the normal settings for these directories:

```
local-send /usr/spool/uucppublic /tmp
local-receive /usr/spool/uucppublic /tmp
```

If you are confused about how these instructions differ from the instructions **remote-send** and **remote-receive**, just remember that the **remote** instructions name directories for send/receive requests initiated by the remote system; where as the **local** instructions name directories for send/receive requests initiated by your system.

One last entry needs to go into this file: you need to name the commands that **mwcbbs** can execute on your system. It needs to execute at least the commands **rmail** and **uucp** so that it can send you mail and upload files to your system. So, add the following line to the end of the entry for **mwcbbs**:

```
commands rmail uucp
```

With these lines, our **sys** entry for **mwcbbs** is complete. It looks like this:

```
system mwcbbs
time SaSu2200-2315
baud 2400
port MWCBBS
phone 17085590412
chat "" \n in:--in: nuucp word: public word: serialnum
protocol g
myname bbsuser
request yes
transfer yes
remote-send /usr/spool/uucppublic /tmp
remote-receive /usr/spool/uucppublic /tmp
commands rmail uucp
```

This is a typical **sys** entry for calling a remote system.

Simplifying a UUCP Configuration With uuinstall

The program **/usr/bin/uuinstall** has been included to help build, modify, or delete entries in the files **sys**, **port**, and **dial**, making configuring UUCP easier. **uuinstall** uses a system of screens and windows to gather and organize the information UUCP needs to work with a remote system. While **uuinstall** does not remove all of the pain from setting up communications with a remote site, it does make this (admittedly complex) task easier.

This section shows how **uuinstall** can simplify the process of configuring UUCP to communicate with **mwcbbs**.

If you have not read the previous section of this chapter, regarding configuring UUCP to call **mwcbbs**, *please do so now. That section discusses in detail many topics that will not be repeated here. In particular, look at the questions asked in the previous section regarding configuring the files* **port, dial**, *and* **sys**, *and be prepared to answer them in this section.*

Invoking uuinstall

To invoke **uuinstall**, just type **uuinstall** at the shell prompt.

For security reasons, only the superuser **root** and user **uucp** can invoke this program. If other users could access this program, they would have access to the information necessary to log into the remote systems listed in **sys**. Only priviledged users should have access to this information.

When you invoke **uuinstall**, the following screen appears:

```
                UUCP Configuration: Main menu



    <s>ys file      Configuration information for specific systems
    <p>ort file     Information for individual ports
    <d>ial file     Configuration information for dialing modems



      Press the letter corresponding to the file you wish to examine
                        or <q> to quit
```

The Port File

The menu says it all: choose the file to work with. For the sake of simplicity, we will modify the system files in the same order that we did in the extended example that appeared in the previous section; so, press **p** to select the file **port**.

When you make your selection, **uuinstall** displays its **action menu**:

```
                        Action menu

      You have selected to work with the port file.

                  Do you wish to:
                          <a>dd an entry
                          <d>elete an existing entry
                          <m>odify an existing entry
                          <v>iew an existing entry

 Press the letter corresponding to the action you wish to perform
             or press <RETURN> for the main menu.
```

Press **a** to add an entry. The following screen appears:

```
             Port File Entry Screen


           port   [_              ]
           type   [     ]
           device [/dev/               ]
           baud   [     ]
           dialer [              ]


 Enter the name that you want associated with
    the device that this entry will define.
           Enter nothing to cancel.
```

It is time to enter our information. To do so, type the appropriate information into each field on the screen. When you have entered all of the information required by that field, press (¢); the cursor drops to the next field on the screen.

When the cursor enters selected fields, **uuinstall** displays a help message to describe the information that you must enter. The message will, in many cases, give an example of valid data.

If you wish not to enter anything into a given field, type (¢) when the cursor enters that field. Some fields are *required*: that is, the system demands that you type something into the field; in this case, the cursor will not move until you have entered something. After you have completed the last line, **uuinstall** prompts you to ask if it should add the new entry to file **port**.

From here, configuring a **port** entry is simple. Look at the questions that we asked in the previous section about the information that a **port** entry needs. Earlier, we decided to name this port **MWCBBS**; therefore, type **MWCBBS** into the first field on the screen — the one labelled **port**. Press (¢); the cursor jumps to the next field, which is labelled **type**.

The port can be either of two types: **direct** or **modem**. Since we're using a modem to telephone **mwcbbs**, type **m** into the field labelled **type**. You would type **d** (for *direct*) should the remote system be connected to yours via a line that runs directly from your serial port to the remote system. The cursor jumps to the next field, which is labelled **device**, and positions itself just to the right of the string **/dev/**.

Now, type the name of the device associated with this port. Earlier, we chose **/dev/com2l**. Because **/dev/** is already in place, just type **com2l**. Again, press (¢); the cursor jumps to the field labelled **baud**.

Type the speed that communications will take place at: in this example, **2400**, then press (¢). The cursor jumps to the field labelled **dialer**.

Finally, type the name of the dialer script that **uucico** is to use to talk with the modem plugged into this port. As you recall, our example uses the script named **hayes**; so type that into this field and press (¢).

The completed entry, as we typed it here, looks like this:

```
        Port File Entry Screen


Do you wish to write this entry? (y/n)_

         port   [MWCBBS]
         type   [modem]
         device [/dev/com2l]
         baud   [2400]
         dialer [hayes     ]
```

uuinstall now asks if you wish to write this entry into file **/usr/lib/uucp/port**. Type **y**, and press (¢). **uuinstall** saves the information and returns to its main menu.

The Dial File

From the main menu, press **d** to select the file **dial**. This will take us to the action menu. Again, select **a** to add an entry. **uuinstall** then displays the following screen:

```
                  Dial File Entry Screen

dialer:        [_             ]
chat:          [                                          ]
chat-timeout:  [   ]
chat-fail:     [                          ]
chat-fail:     [                          ]
chat-fail:     [                          ]
complete-chat: [                                          ]
abort-chat:    [                                          ]


Enter the name of the dialer that
this entry describes.
Leaving this field blank aborts entry.
```

Begin by typing the name of the dialer script. In the previous example, we named it **hayes**. Type this in the first line; then press (¢). The cursor jumps to the next field, which is labelled **chat**.

Now, type the chat script that **uucico** will use to dial the modem. Please refer to the previous section for a discussion of what a chat script is, and of the elements that the dialer chat should contain. In the previous section, we decided to use the following script:

```
"" ATE0Q1V1L1M0DT\D CONNECT\s2400
```

Type this, then press (¢). The cursor jumps to the next field, which is labelled **chat-timeout**.

Type the number of seconds that **uucico** should wait before it aborts its attempt to dial out on the modem. Press (¢); the cursor jumps to first field labelled **chat-fail**.

Into the next three fields, enter messages that the modem might return when an attempt to connect to a remote system fails. In the previous section, we selected the messages **NO\sCARRIER**, **BUSY**, and **NO\sDIALTONE**. Type the first message, then press (¢). The cursor jumps to the second **chat-fail** field; type the second message and press (¢) again. The cursor jumps to the third **chat-fail** field; type the third message and press (¢) once again. The cursor jumps to the field labelled **complete-chat**.

Note that **uuinstall** gives you space to enter three **chat-fail** messages. You can, however, enter an indefinite number of these messages into a dialer script. If you wish to enter more than three **chat-fail** messages, you must edit the file **/usr/lib/uucp/dial** by hand.

The next two fields, respectively labelled **complete-chat** and **abort-chat**, let you enter the chat scripts that **uucico** should execute when, respectively, a call completes successfully or fails for some reason. As we noted in the previous section, these scripts are optional; however, you are well advised to enter them, to ensure that the modem is returned to its correct state after a call is completed. In the previous section, we devised the following script for our Hayes-compatible modem:

```
"" +++ OK ATH0E0V0Q1
```

Type that script into the field labelled **complete-chat**, then press (¢). Type again type it into the field labelled **abort-chat**, and press (¢) again.

The completed dial entry screen should now look like this:

```
            Dial File Entry Screen

dialer:        [hayes     ]
chat:          ["" ATE0Q1V1L1M0DT\D CONNECT\s2400]
chat-timeout:  [60]
chat-fail:     [BUSY]
chat-fail:     [NO\sDIALTONE]
chat-fail:     [NO\sCARRIER]
complete-chat: ["" +++ OK ATH0E0V0Q1]
abort-chat:    ["" +++ OK ATH0E0V0Q1]


     Do you wish to save this entry? (y/n)
```

If you are comfortable with your entry as it is, then press **y** to write it into **/usr/lib/uucp/dial**. If not, press **n**. In either case, **uuinstall** returns to its main menu.

The sys File

It is now time to describe a remote system, in this case **mwcbbs**, to this system. From the main menu, press **s** to select the **sys** file, then select **a** from the action menu to add an entry to the sys file. The following screen will appear:

```
                        Sys File Entry Screen


      system:     [_       ]
      time:       [                                          ]
      speed:      [     ]
      port:       [             ]
      phone:      [                                          ]
      chat:       [                                                    ]
      myname:     [        ]
      protocol:   [   ]
      commands:   [                                                    ]
      read-path:  [                                                    ]
      write-path: [                                                    ]


      Enter the name of a remote uucp system. You should
      limit the name to 8 characters.

      Leaving this field blank aborts entry.
```

Because we are using the remote system **mwcbbs** as our example, type **mwcbbs** into the field labelled **system**, then press (¢); the cursor jumps to the field labelled **time**.

For this field, let's get a little creative. Let's limit calling **mwcbbs** to after 6 PM and before 6 AM on weekdays, but permit any time on weekends. Type the following line in the **time** field:

```
Sa,Su,Wk1800-0600
```

Press (¢), which moves the cursor to the field labelled **speed**.

Now enter the speed or ("baud rate") at which communications will occur. Earlier, we selected 2400 bps; so type **2400** and then (¢).

The cursor is now in the field labelled **port**. Type the name of the port via which we will call **mwcbbs**. As noted above, we will be using the port that we named **MWCBBS**; so type that into this field and then press (¢).

The cursor is now in the field labelled **phone**, which holds the telephone number for the remote system. To find the telephone number for **mwcbbs**, check the release notes that came with your copy of COHERENT; then type it into this field and press (¢). The cursor jumps into the field labelled **chat**, for the chat script.

We discussed the chat script in some detail in the previous section. Enter the following chat script in this field:

```
"" \n in:--in: nuucp word: public word: serialno
```

Remember to replace *serialno* with the serial number provided with your COHERENT package. When you have finished typing the chat script, press (¢). this moves the cursor to the field labelled **myname**.

Because **mwcbbs** does not grant access to just any system, your system must identify itself as a system that **mwcbbs** already knows about. **mwcbbs** grants access to every system that identifies itself as **bbsuser**; so type **bbsuser** into this field, and press (¢). The cursor moves into the field labelled **protocol**.

This field holds the protocol with which your system will exchange files with **mwcbbs**. The Taylor UUCP package supports several protocols. **mwcbbs** in turn recognizes protocols **a**, **g**, and **i**. Please refer to the Lexicon article **sys** for more information on available protocols, and the strengths and weaknesses of each. For the purposes of this example, type **g** (for the **g** protocol), then press (¢).

The cursor is now in the field labelled **commands**, which lists the commands that the remote system may execute on your system. Because **mwcbbs** will not be calling you, just press (¢). **uuinstall** will write a default list of commands into this field, then move the cursor to the next field.

The last two fields, which respectively are labelled **read-path** and **write-path**, limit the directories that the remote system can, respectively, write into or read from. They point to the **remote-send** and **remote-receive** instructions within a **sys** entry. Press (¢) for each field; **uuinstall** will write into each field the default directory, which is **/usr/spool/uucppublic**.

With the template completed, your entry should look like this:

```
                        Sys File Entry Screen

      system:     [mwcbbs]
      time:       [Sa,Su,Wk1800-0600]
      speed:      [2400]
      port:       [MWCBBS]
      phone:      [17085590412]
      chat:       ["" \n in:--in: nuucp word: public word: serialno]
      myname:     [bbsuser]
      protocol:   [g]
      commands:   [rmail rnews uucp uux]
      read-path:  [/usr/spool/uucppublic]
      write-path: [/usr/spool/uucppublic]

                 Do you wish to save this entry? (y/n)
```

If you are comfortable with the information you entered, press **y** to have it written to the file **/usr/lib/uucp/sys**.

This concludes our examples of configuring UUCP to call remote systems. Due to the extreme flexibility of the Taylor UUCP package, it is not feasible to review all possible configurations available to you. Each of the configuration files, **/usr/lib/uucp/sys**, **/usr/lib/uucp/port**, and **/usr/lib/uucp/dial**, can include more commands than are reviewed here. Many of these are reviewed in the Lexicon entries for each of the files. A complete set of ASCII text Taylor UUCP documentation, as provided with the distribution available from several internet sites, is included in the file **/usr/src/alien/uudoc104.tar.Z**. The complete source code for the Taylor UUCP package as distributed with COHERENT is available from **mwcbbs**.

Modifying an Existing Entry

The above examples show how to use **uuinstall** to enter a new entry into files **port**, **dial**, and **sys**. You can also use **uuinstall** to delete, view, or modify existing entries in these files. Of these, the most useful is the feature for modifying an existing entry.

Let's say that we want to modify our entry for dialer **MWCBBS** to include a fourth **chat-fail** instruction, for the modem message **ERROR**. To do so, do the following:

- Invoke **uuinstall** from the shell, as described above.
- When **uuinstall** displays its main menu, type **d**, to edit the file **/usr/lib/uucp/dialer**.
- When **uuinstall** displays its action screen, type **m**, to modify this file.
- **uuinstall** then displays the names of all of the entries in this file. Use the arrow keys on your terminal to move the cursor to the entry that you wish to enter, in this case **MWCBBS**; then press (¢).
- **uuinstall** extracts the entry for **MWCBBS** from **/usr/lib/uucp/dial**, writes it into a temporary file, then invokes the MicroEMACS editor for that temporary file. You can use the usual MicroEMACS commands to edit this entry. In this case, add the line

  ```
  chat-fail ERROR
  ```

  into the entry.
- When you have finished editing the entry, type **<ctrl-Z>**.
- **uuinstall** will prompt you and ask if you wish to save your changes. Type **y** if you do, **n** if you do not.

Note that if you do save your changes, you can always use **uuinstall** to remodify the entry.

Configuring UUCP for Dial-in Access

The above examples show how to configure your UUCP system so that it can dial out to another remote system. Configuring UUCP so that other systems can dial into your system is, for the most part, like configuring it to dial out; but this task does present some special problems that you must consider. The following example shows how to set up UUCP so that remote system **dalek** can dial into your system.

Giving a Remote UUCP Site a Login

At this point, you are now the systems administrator of your COHERENT system who must tell someone else how to set up her UUCP to log into your system. We've shown you the flip side of this by showing you how to access **mwcbbs**: now the job is yours.

When a UUCP site calls your system, it must log in as would any ordinary user would. Once it has logged in, however, it runs the command **/usr/lib/uucp/uucico** rather than a shell, which a normal user would run. This portion of what you must set up is configured in the file **/etc/passwd**.

You can create a UUCP login by running the command **newusr**; then edit the last field of the **/etc/passwd** entry for the login you just established to run the command **/usr/lib/uucp/uucico** instead of a **/bin/sh** or **/usr/bin/ksh**.

You could also create a UUCP login by manually editing **/etc/passwd** and copy the entry for user **uucp**, but change the user name of **uucp** to something else.

You must also define the home directory if using **newusr**. Because this is a UUCP account, the home directory appears under the directory **/usr/spool/uucp**. For example, if you wanted site **dalek** to call you, you might establish an **/etc/passwd** entry that looks like:

```
dalek:password:6:6:Coherent-Coherent \
        copy:/usr/spool/uucp:/usr/lib/uucp/uucico
```

Please note that *password* in the entry for **dalek** represents the encrypted password you assigned to site **dalek**. Give the password to the system administrator of site **dalek** so that she may properly configure her chat script to log into your system.

If we were to stop right here, **dalek** could call your system, log in, and begin a UUCP session. Unfortunately, since we've yet to configure the UUCP files themselves to know about **dalek**, your site would quickly terminate the call when **dalek** identified itself to your system after completing its chat script.

Configuring a Spooling Directory for Remote UUCP Access

Each UUCP site that calls your system must have a *spooling directory* in **/usr/spool/uucp**. While logged in as **root**, go to the directory **/usr/spool/uucp** and run the command:

```
/usr/lib/uucp/uumkdir dalek
```

Configuring UUCP Files

To control what **dalek** does on your system, you should describe it in file **/usr/lib/uucp/sys**. COHERENT includes a dummy entry in this file that you can easily modify for site **dalek**. You should make an entry that looks like this:

```
system dalek
time Never
commands rmail rnews uucp uux
remote-send /usr/spool/uucppublic !/usr/spool/uucppublic/bobfiles
remote-receive /usr/spool/uucppublic !/usr/spool/uucppublic/private
```

This entry names **dalek** as a system UUCP recognizes. Your system will never call **dalek**, because the **time** command (which gives the legal dates and times during which the remote system can be contacted) states **Never**.

The **commands** instruction names the commands that you permit **dalek** to execute on your system. This line overrides any permissions that may be available to **dalek** on the system level: that is, this line can stop **dalek** from executing commands that it otherwise would have permission to execute (e.g., **ls** or **cat**), but it cannot enable **dalek** to execute commands that it normally would not be permitted to run (e.g., **su**, **shutdown**, or **reboot**).

The lines **remote-send** and **remote-receive** name, respectively, the directories whose contents **dalek** can read, and the directories into which it can write files. Once again, these lines can stop **dalek** from accessing directories that it otherwise would be allowed to access (e.g., **tmp**), but they cannot give **dalek** permission to manipulate directories that it normally would not be able to access (e.g., **bin**).

In brief, all you have done is identified **dalek** to your system, named the commands it can execute on your system, and limited the directories it can access. It's that easy!

One Last, Loose Thread

With the spooling directory created, we are almost done. Run this command:

```
/usr/lib/uucp/uutouch dalek
```

It will place a dummy command in **dalek**'s spooling directory. More important, it returns an error if it finds some errors in the UUCP configuration for **dalek**.

Unfortunately, we cannot give you a test system that will call your system to test your UUCP configuration. You will have to use this section as a guide to configure for another UUCP site to call yours.

Requesting Files From a Remote UUCP System

To request a file from a remote UUCP system, you must know where that file is on the remote system. The file **howto.start** can be found in the directory **/usr/spool/uucppublic/mwcnews** on **mwcbbs**. This file introduces **mwcbbs**, its features and intended uses, and how to request files from it.

With this bit of knowledge, we can now request the file with the command **uucp**.

uucp is very simple. Invoked it with a specific site to call, and file to upload or download. For example, the command:

```
uucp mwcbbs!/usr/spool/uucppublic/mwcnews/howto.start /tmp
```

tells your machine to call **mwcbbs**, download the file

```
/usr/spool/uucppublic/mwcnews/howto.start
```

and put it into directory **/tmp** on your system. The call will take place seconds after you enter the command, unless you tell **uucp** to spool the request. For more information on this and other arguments, see the Lexicon entry for **uucp**.

Please note that the entry for **mwcbbs** in **/usr/lib/uucp/sys** must specify that **mwcbbs** can write to **/tmp** as part of the **remote-receive** instruction.

To send a file to **mwcbbs**, use the command:

```
uucp filename mwcbbs!/usr/spool/uucppublic/uploads/
```

This command uploads a copy of *filename* to the directory **/usr/spool/uucppublic/uploads** on **mwcbbs**. Again, the call takes place within seconds, unless you tell **uucp** to spool the request.

At this point, we have completed our **uucp** configuration to "talk" to **mwcbbs**, and we have requested our first file. You can tell **uucp** to download other files from **mwcbbs**; only the file names and path names will change.

Sending Files to a Remote UUCP System

Suppose, for example that site **santa** has been described to your UUCP system, and everyone has permission to read from your current directory. Suppose, too, that you have permission to write into directory **/usr/spool/reports/parents**. To send the files **good.kids** and **bad.kids** to **santa**, type the following command:

```
uucp good.kids bad.kids santa!/usr/spool/reports/parents
```

The **uucp** command compels UUCP to copy one or more files from your site to a remote site. UUCP queues both files automatically and sends them at the next scheduled time.

Note, too, the use of the '!' in the above command. The '!' separates a site name from another site name, from a directory name, or from a user ID. In the above example, the '!' indicates that directory **/usr/spool/reports/parents** can be found at site **santa**. One feature of a UUCP network is that any member can send files to any other member. That does not mean that every member must have full permissions with every other member; rather, for the sake of efficiency it is possible to route files through one or more intermediate computers, to allow batch transmissions of files. For example, to send the file **visibility** to user **blitzen** via machines **santa** and **reindeer**, use the following command:

```
uucp visibility santa!reindeer!blitzen!/usr/spool/weather/usa
```

In this example, the string **santa!reindeer!blitzen!/usr/spool/weather** indicates that directory **/usr/spool/weather** can be contacted at site **blitzen**, which in turn can be contacted via site **reindeer**, which in

turn can be contacted via site **santa**. This scenario assumes that site **reindeer** has permission to write into directory **/usr/spool/weather** on site **blitzen**, and that site **santa** has permission to upload files to site **reindeer**. (And, of course, that you have permission to upload files to site **santa**.) If any of these are not true, the transaction will fail.

UUCP Administration

Once you have written and debugged the descriptions of your ports, dialers, and systems, administering UUCP consists mainly of reviewing the log files periodically to ensure that all connections are being made, and all programs executed correctly. The command **uulog** will assist you in this. When you type the command

```
uulog widget
```

uulog will open all of the log files associated with site **widget**, and display them for you. Given that the log files for given site are kept in four different directories, this can be a great convenience.

Logfiles are organized as follows:

```
/usr/spool/uucp/.Log/uucico/sitename
/usr/spool/uucp/.Log/uucp/sitename
/usr/spool/uucp/.Log/uux/sitename
/usr/spool/uucp/.Log/uuxqt/sitename
```

As you can see, one logfile for each site is kept in a directory named after a given UUCP command. UUCP records every transaction; so by reading these files, you can see whether your UUCP commands are succeeding.

If you are having trouble with your UUCP connections, send files through UUCP and observe how they fail. You may need to use **uuinstall** a few times to tweak your description of the remote site. If all else fails, contact Mark Williams Company.

If all is going well, you should run **/usr/lib/uucp/uumvlog** every day. This keeps the log files from getting out of hand. The previous section on setting the polling time describes how to do this.

The main task of the UUCP administrator is to monitor the UUCP log files to see that hardware is functioning correctly, and that files are transferred correctly. For example, failure to connect with a remote site after several attempts may mean that the remote site is having problems with its modem, or that it is scheduling outgoing calls for when you were scheduled to call in. Likewise, failure to receive scheduled calls from several sites may indicate equipment failure on your end.

Finally, the UUCP administrator must monitor the use of disk space on the system. Old mail and messages, multiple copies of files, and files automatically input by various subscription and network services can eat up disk space rapidly; you must prune extraneous material ruthlessly.

Networks

UUCP becomes truly useful when you are hooked into a network of machines that exchange information. Through UUCP, you can gain access to the Internet, through which you can exchange news and mail with others users around the world.

This section briefly describes the services you can obtain from a network, and the networks now available.

Services

Many different services are available from networks: domain-name service (DNS), mail exchanger service, and connectivity.

DNS is the registration of an Internet-style domain, e.g., **mwc.com** or **baqaqi.chi.il.us**. This usually divides into two subservices — registration in an existing domain, and registration of your own domain. For this service you need two sites on the Internet willing to either register you in their nameserver or run a nameserver for your domain. A *nameserver* gives other people's machines information about your registered machine or registered domain. In particular, the nameserver publishes an MX record, which tells machines how to get mail to you. (There are other kinds of records, but MX records are the important ones for UUCP sites.)

MX service is mail forwarding. The MX record published by the nameserver must point at a machine directly on the Internet. This machine will the be responsible for figuring out how to actually get the mail to you.

Connectivity usually means a UUCP connection. Because almost everybody in a major city is connected to everyone else, a UUCP connection to anybody effectively translates to an indirect connection to the Internet.

Available Networks

UUNET is a company in Falls Church, Virginia, that provides a large number of networking services, including all of those mentioned above.

The **UUCP network** is an extremely informal group of machines defined only by the fact that they connect to each other via the UUCP protocol. It is one of the largest networks in the world and has no central control.

UseNet is a network of machines defined by the fact that they exchange UseNet news with each other. UseNet is also an anarchy — no central organization runs it. It includes machines in the UUCP network and the Internet, as well as hundreds of other networks. It is the largest network in the world.

The **Internet** is a group of high-speed networks which all communicate with *Internet Protocol*, i.e., TCP/IP. The networks that comprise the Internet are mostly academic and research networks run by large central organizations, such as the National Science Foundation or the Australian Academic Research Project. The center of the Internet is the NIC (Network Information Center), whose address is **nic.ddn.mil**.

The **internet** (lower-case 'i') is defined by connectivity under mail. It is technically larger than UseNet, though less is said about it because it is so weakly defined.

For information on networks, what is available on them, and how to connect to them, we strongly recommend the book *The Whole Internet: User's Guide and Catalog*, cited above. For a copy, check your local bookstore, or telephone the publisher, O'Reilly & Company, at 1-800-998-9938.

Debugging UUCP Problems

When you have a problem with UUCP — and in particular, a problem with telephoning another UUCP system — you must have a clear picture of what is occurring, and what is not occurring. For instance, if you try to call **mwcbbs** and UUCP fails, you must determine what is working properly before the failure takes place.

The following subsections describe common problems with UUCP, and gives some hints on how to solve them. Please review them *carefully* before you telephone Mark Williams Company to ask for help. If you do not, we will ask that you do review them and call back only if you still cannot solve the problem.

Define the Problem Exactly

UUCP problems can take many forms. Define the problem *exactly*. This process may actually help you solve the problem. Statements like "I'm having a problem using UUCP" or "UUCP doesn't work" do not describe problems relating to UUCP. You need to know *exactly* what does or does not happen when you try to connect with another site. Please review it before you call Mark Williams Technical Support.

Before you do anything else, trying running **uucp** and **uucico** with the option **-x**. This option tells these programs to log what they do; the logs are written into the subdirectories of directory **/usr/spool/uucp/.Admin**. Often, these log messages will point directly to the problem.

A subtle error within a UUCP configuration file can cause no end of grief when you try to debug a UUCP problem. To help spot these potential problems, run the command **/usr/lib/uucp/uuchk**. This command generates a full report on your configuration files. It will spot and report on syntax errors in your configuration files. You will, of course, have to fix by hand whatever **uuchk** finds to be in error.

The following sections discuss commonly encountered problems.

Enabling and Disabling Ports

On some COHERENT systems, the permissions for the programs **/etc/enable** and **/etc/disable** are set to:

```
-r-x------   root     root
```

That is, these commands can be executed only by user **root**. This is to close a security hole; otherwise, a person who breaks into your system can disable any port she wants — including the console.

If you have only one modem, and you both initiate and receive UUCP sessions on that modem, you may run into problems with enabling and disabling that port. For the communications program **uucico** to be able to enable and disable the port on its own, **/etc/enable** and **/etc/disable** must have the permissions:

```
-r-s--s--x   root     root
```

These permissions permit **uucico** to dial out on an enabled port on its own; but it reopens the security hole. *Caveat utilitor.*

TUTORIALS

For information on how to change permissions, see the Lexicon entry for the command **chmod**.

Stale Requests and Multiple Requests

From time to time, you may accidentally issue the same **uucp** request more than once.

Note that if **uucp** fails because you failed to connect with the remote site, one action you should *not* take is to repeat the **uucp** command. If you do this, **uucp** will simply queue another request for the same file or files that you requested with the previous **uucp** command. When connection is finally made, multiple **uucp** requests will be executed, thus downloading multiple copies of the same file or files. Depending upon the size of the file or files, this could be an expensive mistake.

To remove stale requests, log in as user **uucp** or user **root**; then **cd** to directory **/usr/spool/uucp/***sitename* and remove the extraneous requests. (You can also do this to remove mail files about which you have had second thoughts.)

Note that a **uucp** generates one file, which has the prefix **C**. A mail message generates two files: one with the prefix **C**, which tells the remote system what to do with the mail message; the other has the prefix **D**, which holds the text of the message. Read the existing files to make sure that you are removing the correct files.

Problems With Lock Files

If **uucico** wishes to dial on a modem but somebody else is already using it, you will see the message

```
ERROR: All matching ports in use
```

in the log file for the site **uucico** is attempting to call. The solution simply is to wait until the port clears.

If **uucico** is already communicating with a given remote system, or if a lock file exists for a given remote system, you will see the message

```
ERROR: System already locked
```

in the log file for the site **uucico** is attempting to call.

Sometimes lock files are not cleared properly, and so tie up the system long after they should have been removed. Such files are called *stale* lock files. The solution is to use the command **uurmlock** to remove stale files.

Note that in some instances, permission problems may stop **uurmlock** from removing lock files. In this case, log in as the superuser **root** and execute the command:

```
rm /usr/spool/uucp/LCK*
```

Enabling Ports, /etc/ttys Problems

uucico reads a port's status in file **/etc/ttys**, then restores that status after it finishes its work. If you had disabled a port by hand, it remains disabled after **uucico** has worked with it — which means, of course, that no remote system can dial into your system via that port. To re-enable a port, use the command **/etc/enable**.

Note, too, that file **/etc/ttys** is sensitive to the order in which devices are named within it. The port into which you have plugged your modem must have an entry for both the remote device (e.g., **/dev/com1r**) and the local device (e.g., **/dev/com1l**). Note that the entry for the raw device *must* precede the entry for the local device. If it does not, **uucico** will not be able to dial out properly.

Note, too, that device **/dev/console** *must* be the last entry in **/etc/ttys**, or your system will not be able to dial out via a serial port.

Permission Problems

Incorrect permissions on files and directories will harm UUCP's behavior.

uucico runs as a user named **uucp** and therefore does not have any special permission or privileges that give it free access to every file or directory on your system. For example, consider what UUCP does when it attempts to contact a remote system:

1. The daemon **uucico** checks the directory **/usr/spool/uucp** to see if any lock files are present; these lock files indicate whether someone has already logged into the port from which **uucico** wishes to dial out.
2. If **uucico** finds a lock file for the port it wants to use, it checks the file **/usr/lib/uucp/sys** for information on an alternate way to contact the remote system. If **uucico** finds an alternate way to contact the remote system, it tries that way. If it does not, then it quits.

3. When **uucico** finds a port that is available, it then check file **/etc/ttys** to see if the port is already enabled for remote logins.

4. If the port is enabled, **uucico** disables the port and makes its call.

So far, so good. If, however, **uucico** does not have read and write permission on the port device from which it is attempting to make its call, its attempt to make the call will fail.

For another example of how directory-level permission affect the behavior of UUCP, consider how UUCP transfers files. When UUCP transfers files, it stores those files as temporary files in the directory **/usr/spool/uucp/***sitename*, where *sitename* is the name of the system with which UUCP is communicating. All files and directories under **/usr/spool/uucp** must be owned by user **uucp** and group **uucp**, or UUCP cannot transfer files correctly.

UUCP can also write temporary files into directory **/usr/spool/uucp/.Temp/***sitename*. This directory must also be owned by user and group **uucp**.

The permissions on the serial port from which you will dial out can affect the behavior of UUCP. UUCP must have permission to read and write to that port. The device specified by the **line** entry in **/usr/lib/uucp/port** should have permissions of **666** (see the Lexicon entry for the command **chmod**).

uucp should own all of its spooling directories. The *spooling directory* is the directory into which UUCP writes stores data and command files for the site being contacted. The spool directory for a given remote site resides under **/usr/spool/uucp** and is named after the remote site. For example, your system will use directory **/usr/spool/uucp/mwcbbs** to store files being exchanged with **mwcbbs**. Likewise, **mwcbbs** has the directory */usr/spool/uucp/yoursystem*, where UUCP stores files to be exchanged with *yoursystem*.

UUCP Cannot Find Its Own Files

If the command **/usr/bin/uucp** says it can not get its own name when you invoke it, then you give yourself a UUCP site name of no more than seven characters in the file **/etc/uucpname**.

The command **/bin/mail** command may also return a similar message. The cure is the same.

As noted above, a UUCP command may also fail to execute because permissions are set up incorrectly on the UUCP executables. UUCP commands frequently invoke one another. For instance, **uucico** invokes **uuxqt** after it has communicated with a remote system. **uuxqt** processes all files uploaded from the remote system. **uuxqt** may, in turn, invoke other commands — for example, it can invoke **smail** to deliver mail to a user on your system or forward mail to a user on another remote system. If the permissions on a UUCP executable are incorrect, it may find that, when it tries to complete a task, it does not have permission to write to a given directory or a log. A list of UUCP permissions appears in the Lexicon entry for **UUCP**. Make sure that the permission on your file conform to what is given there.

Modem Configuration

The commonest source of error lies in modem configuration. Each modem has its idiosyncrasies, and command languages differ from device to device; however, in general your modem should be configured as follows:

- Echo off.
- No result codes.
- Carrier detect (DCD) is true.
- Terminal ready (DTR) is true.
- DCD follows DTR.

If you are working with a high-speed modem (9600 baud or faster), you should also configure it to do the following:

- Lock modem to computer baud rate at 19,200 if your modem supports it;, if not, lock it to 9600 baud.
- Set modem for RTS/CTS handshaking.
- Use a flow-control device for the port's description in file **/usr/lib/uucp/port**. See the Lexicon entry **asy** for details on how to do this.

Telebit modems present special problems for configuration. They are designed to be used with UUCP, and in general they are fast and robust; but because they do more than ordinary modems, they require more extensive setup to work correctly. The Lexicon entry for **modem** gives detailed information on how to set up a Trailblazer modem so that it works properly with the COHERENT implementation of UUCP.

If your modem supports data compression, it may not be ideal to use this feature with every remote site. For instance, attempting to compress files that already are compressed (as are the files on the Mark Williams bulletin board), only adds to the data stream and *reduces* overall throughput. Before you turn on data compression, make sure that the files you are downloading are *not* compressed.

The Modem Does Not Respond

When you try to call a system via the commands **/usr/bin/uucp** or **/usr/lib/uucp/uucico** and the modem does not respond (i.e., the lights on the modem do not flicker), look at file **/usr/lib/uucp/port**. Check the permissions on the serial port used to dial out on, as specified therein.

In some cases, you will see the error message

```
Retry time not reached
```

Taylor UUCP puts a horizon on callouts: if a call to a given site fails, UUCP will wait a predetermined amount of time before it tries again. If you are repeatedly invoking **uucp** or **uucico** from the command line, you may see this error. To get around this limitation, use the command-line option **-f**; for example:

```
uucico -s systemname -f
```

This problem can also arise if a previous connection to a site failed. In this case, UUCP writes a bad-connection report into file

```
/usr/spool/uucp/.Status/systemname
```

where *systemname* names the system you are trying to contact. UUCP does this to keep your system from wasting time contacting a system whose connection is defective, even if you use the **-S** option with **uucico**. Remove this file and UUCP will resume dialing out. Note that this will not clear up the problem that triggered the original bad-connection report, and the connection may fail for other reasons.

The Modem Responds But Does Not Dial

In some cases, the modem responds (i.e., its lights flicker) but it does not dial out. This can have any of several causes.

First, the modem's register settings may be incorrect. Review them. Check the above example for some simple examples of how to set modem registers, and check the documentation that came with your modem.

You may be trying to access the modem through the remote COM port, e.g., **/dev/com1r**. In this instance, the system awaits a carrier signal, as you cannot open a remote COM port without; therefore, no communication ever begins. The solution is to use the local device, e.g., **/dev/com1l**. This way, the system will not wait for the carrier to come up on the modem, and dialing will begin.

The Modem Dials But No Connection Made

Sometimes a modem will dial out but no connection is made. This is typically caused by plugging the telephone line into the wrong port on the modem.

Check the log file for the site you are calling. It will usually give a message that indicates what the problem really is. If calling **mwcbbs**, use the command:

```
uulog mwcbbs
```

The Modem Dials, Carrier Is Established, Nothing Else Happens

The first suspect is the modem's register settings. The modem register settings that we discussed above generally work well for **uucp** to dial out to another system, *if* your modem is Hayes-compatible. If it is not, or if it is an off-brand that only *claims* to be Hayes-compatible, check your modem's documentation and make sure that the register settings are correct.

To get a picture of what is happening, run the command **/usr/lib/uucp/uucico** with the option **-xchat**. If calling **mwcbbs**, use the command:

```
/usr/lib/uucp/uucico -Smwcbbs -xchat
```

This tells your system to call **mwcbbs** and to write debugging information into its log file **/usr/spool/uucp/.Admin/audit.local**, which you can review later. This is very useful in determining if there is a problem in a chat script.

uulog Shows Lost Packets

If your UUCP communication sessions terminate prematurely, your system may be losing characters on the serial ports. An indication of this problem is the appearance of the message **Lost Packets** in your UUCP logs. If your system exhibits these symptoms during transfers on 4800-bps or higher-speed lines, *we strongly urge you to replace your existing 8250- or 16450-based UARTs with those based upon the 16550A design, such as the National Semiconductor NS16550AFN.* These newer UARTs are pin-compatible with the older UARTs. COHERENT automatically senses and enables them when it boots.

uulog Shows Incorrect Response

This points to one of four problems:

1. Your site is sending an improper site name to the remote system (in other words, the remote site doesn't know about your system).
2. The remote site does not have a spooling directory for your site.
3. Your site does not have a spooling directory for the remote site
4. **/usr/lib/uucp/sys** contains an error or incorrect chat script.

Files Refuse To Be Sent or Cannot Be Received

Check the instructions **local-send**, **local-receive**, **remote-send**, and **remote-receive** in the remote system's entry in file **/usr/lib/uucp/sys**. This can result in the remote system being denied permission to load files onto your system, or read files from it. Make sure these instructions are set correctly, and point to directories in which user **uucp** has read and write permission.

File Transfers Fail With imsg Statements

One problem is frequently encountered when COHERENT systems attempt to UUCP with Sun workstations: connections are made correctly, but when file transfers fail with numerous iterations of the debug message "imsg" until the script times out. This is caused by differences in parity: the COHERENT chat scripts by default use no parity, while those on the other system do. The solution is to change the chat script on the Sun system to set no parity when it talks with COHERENT system.

Files are Being Lost

If a configuration problem exists on your side of a UUCP connection, files could be lost. This may be true if **uucico** or **uuxqt** are running with incorrect permissions. Taylor UUCP notes problems of this nature in its log files. If it can preserve a file that would otherwise be lost, **uucico** saves the file in directory **/usr/spool/uucp/.Preserve**, and logs the fact that it has saved it.

Non-COHERENT UUCP Site Problems

It is important to understand that COHERENT's UUCP is designed to be compatible with other implementations of UUCP, but may not use the same configuration files. This can complicate the debugging of problems when you attempt to establish communication with a system that uses a different implementation of UUCP.

We will supply whatever assistance we can, but if it is determined that the non-COHERENT site is part of the problem, it is up to you to find how that non-COHERENT site has configured itself. You may even need to set up a conference call among yourself, the remote site's administrator, and MWC Technical Support.

Where to Go From Here

As we mentioned earlier, COHERENT does not now implement the entire Taylor UUCP package. Full sources for Taylor UUCP are available from **mwcbbs**, and from various sites on the Internet.

This tutorial only touches upon the esentials of configuring UUCP for communicating with other systems. Taylor UUCP is much more conformable than this tutorial may have led you to believe.

For a fuller description of the Taylor UUCP configuration files, see the Lexicon entries for **dial**, **port**, and **sys**. For further information, check the Lexicon entry for each UUCP command, as well as the overview article **UUCP**. This article will also point you to related articles in the Lexicon, such as the ones for **modem** and **RS-232**.

The Lexicon

The rest of this manual consists of the Lexicon. The Lexicon consists of more than 1,000 articles, each of which describes a function or command, defines a term, or otherwise gives you useful information. The articles appear in alphabetical order.

Internally, the Lexicon has a *tree structure*. The "root" entry is the one for **Lexicon**. It, in turn, introduces (or "branches to") three "overview": **Running COHERENT**, **Administering COHERENT**, and **Programming COHERENT**. Each overview article points to a group of related entries. For example, the article **Programming COHERENT** points to the articles on the COHERENT C compiler and to articles that introduce the library functions, macros, and header files included with COHERENT.

Each Lexicon entry cross-references other entries. These cross-references point up the documentation tree, to its overview article and, ultimately, to the entry for **COHERENT**; down the tree to subordinate entries; and across to entries on related subjects. For example, the entry for **getchar()** cross-references **STDIO**, which is its overview article, plus **putchar()** and **getc()**, which are related entries of interest to the user. The Lexicon is designed so that you can trace from any one entry to any other, simply by following the chain of cross-references up and down the documentation tree.

For more information on how to use the Lexicon and how it is organized, see the Lexicon entry for **COHERENT**.

! to ~

— Preprocessing Operator

String-ize operator

The preprocessing operator **#** can be used within the replacement list of a function-like macro. It and its operand are replaced by a string literal, which names the sequence of preprocessing tokens that replaces the operand throughout the macro.

For example, the consider the macro:

```
#define display(x) show((long)(x), #x)
```

When the preprocessor reads the following line

```
display(abs(-5));
```

it replaces it with the following:

```
show((long)(abs(-5)), "abs(-5)");
```

Here, the preprocessor replaced **#x** with a string literal that gives the sequence of token that replaces **x**.

The following rules apply to interpreting the **#** operator:

1. If a sequence of white-space characters occurs within the preprocessing tokens that replace the argument, it is replaced with one space character.
2. All white-space characters that occur before the first preprocessing token and after the last preprocessing token are deleted.
3. The original spelling of the preprocessing tokens is preserved. This means that you must take care to preserve certain characters: a backslash '\' should be inserted before every quotation mark '"' that marks a string literal, and before every backslash that introduces a character constant.

Example

The following uses the operator **#** to display the result of several mathematics routines.

```
#include <errno.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

void show(value, name)
double value, char *name;
{
      if (errno)
            perror(name);
      else
            printf("%10g %s\n", value, name);
      errno = 0;
}

#define display(x) show((double)(x), #x)

main()
{
      extern char *gets();
      double x;
      char string[64];

      for(;;) {
            printf("Enter a number: ");
            fflush(stdout);
            if(gets(string) == NULL)
                  break;
```

```
            x = atof(string);
            display(x);
            display(cos(x));
            display(sin(x));
            display(tan(x));
            display(acos(cos(x)));
      }
}
```

See Also

#define, C preprocessor
ANSI Standard, §6.8.3.2

— Preprocessing Operator

Token-pasting operator

The preprocessing operator **##** can be used in both object-like and function-like macros. When used immediately before or immediately after an element in the macro's replacement list, **##** joins the corresponding preprocessor token with its neighbor. This is sometimes called "token pasting".

As an example of token pasting, consider the macro:

```
#define printvar(number) printf("%s\n", variable ## number)
```

When the preprocessor reads the following line

```
printvar(5);
```

it substitutes the following code for it:

```
printf("%s\n", variable5);
```

The preprocessor throws away all white space both before and after the **##** operator. This gives you an easy way to print any one of a set of strings.

must not be used as the first or last entry in a replacement list. All instances of the **##** operator are resolved before further macro replacement is performed.

For more information on object-like and function-like macros, see **#define**.

See Also

 # #define, C preprocessor
ANSI Standard, §6.8.3.3

Notes

Some C implementations allow token pasting by using an empty comment. For example:

```
variable/**/number
```

The COHERENT C compiler does not recognize this "trick" because it is not consistent with the Kernighan & Ritchie standard for C, which states that a comment is white space and therefore is a token separator. In any event, token pasting should always be performed with **##**.

The **##** operator may be used only within the replacement text of a preprocessor macro definition.

The order of evaluation of multiple **##** operators is unspecified.

#define — Preprocessing Directive

Define an identifier as a macro

The preprocessing directive **#define** tells the C preprocessor to regard *identifier* as a macro.

#define can define two kinds of macros: *object-like*, and *function-like*.

An object-like macro has the syntax

```
#define identifier replacement-list
```

This type of macro is also called a *manifest constant*. The preprocessor searches for *identifier* throughout the text of the translation unit, and replaces it with the elements of *replacement-list*, which is then rescanned for further macro substitutions.

For example, consider the directive:

```
#define BUFFERSIZE 75
```

When the preprocessor reads the line

```
malloc(BUFFERSIZE);
```

it replaces it with:

```
malloc(75);
```

A given *identifier* is replaced only once by a given *replacement-list*. This is to prevent such code as

```
#define FOO FOO
```

or

```
#define FOO BAR
#define BAR FOO
```

from generating an infinite loop.

A function-like macro is more complex. It has the syntax:

`#define` *identifier lparen identifier-list*$_{opt}$) *replacement-list*

The preprocessor looks for *identifier*, which is a macro that resembles a function in that it is followed by a pair of parentheses that may enclose an *identifier-list*. It replaces *identifier* with the contents of *replacement-list*, up to the first lparen '(' within *replacement-list*.

The preprocessor then examines *identifier-list* for further macros, which it expands. The modified *identifier-list* is then replaced with the rest of *replacement-list*. Pairs of parentheses that are nested between the lparen that begins *replacement-list* and the ')' that ends it are copied into *identifier-list* as literal characters. The identifiers within *identifier-list* are preserved after it has been modified by *replacement-list*. The only exceptions are identifiers that are prefixed by the preprocessing operators **#** or **##**; these are handled appropriately.

For example, the consider the macro:

```
#define display(x) show((long)(x), #x)
```

When the preprocessor reads the following line

```
display(abs(-5));
```

it replaces it with the following:

```
show((long)(abs(-5)), "abs(-5)");
```

When an argument to a function-like macro contains no preprocessing tokens, or when an argument to a function-like macro contains a preprocessing token that is identical to a preprocessing directive, the behavior is undefined.

Example

For an example of using a function-like macro in a program, see **#**.

See Also

#, ##, #undef, C preprocessor
ANSI Standard, §6.8.3

Notes

A macro expansion always occupies exactly one line, no matter how many lines are spanned by the definition or the actual parameters. If you have defined macros that span more than one line, you must either redefine them to occupy one line, or somehow embed the newline character within the macro itself; otherwise, the macro will not expand correctly.

A macro definition can extend over more than one line, provided that a backslash '\' appears before the newline character that breaks the lines. The size of a **#define** directive is therefore limited by the maximum size of a logical

source line, which can be up to at least 509 characters long.

Some implementations allowed a user to re-define a macro with a new **#define** directive. The Standard, however, allows only a "benign" redefinition; that is, the body of the new definition must exactly match the old definition, including parameter names and white space.

#elif — Preprocessing Directive

Include code conditionally

The preprocessing directive **#elif** conditionally includes code within a program. It can be used after any of the instructions **#if**, **#ifdef**, or **#ifndef**.

If the conditional expression of the preceding **#if**, **#ifdef**, or **#ifndef** directive is false (i.e., evalutates to zero) and if the current condition is true (i.e., evaluates to a value other than zero), then *group* is included within the program, up to the next **#elif**, **#else**, or **#endif** directive. An **#if**, **#ifdef**, or **#ifndef** directive may be followed by any number of **#elif** directives.

The *constant-expression* must be an integral expression, and it cannot include a **sizeof** operator, a cast, or an enumeration constant. All macro substitutions are performed upon the *constant-expression* before it is evaluated. All integer constants are treated as long objects, and are then evaluated. If *constant-expression* includes character constants, all escape sequences are converted into characters before evaluation.

See Also

#else, #endif, #if, #ifdef, #ifndef, C preprocessor, defined
ANSI Standard, §6.8.1

#else — Preprocessing Directive

Include code conditionally

The preprocessing directive **#else** conditionally includes code within a program. It is preceded by one of the directives **#if**, **#ifdef**, or **#ifndef**, and may also be preceded by any number of **#elif** directives. If the conditional expressions of all preceding directives evaluate to false (i.e., to zero), then the code introduced by **#else** is included within the program, up to the **#endif** directive.

A **#if**, **#ifdef**, or **#ifndef** directive can be followed by only one **#else** directive.

See Also

#elif, #endif, #if, #ifdef, #ifndef, C preprocessor
ANSI Standard, §6.8.1

#endif — Preprocessing Directive

End conditional inclusion of code

The preprocessing directive **#endif** must follow any **#if**, **#ifdef**, or **#ifndef** directive. It may also be preceded by any number of **#elif** directives and an **#else** directive. It marks the end of a sequence of source-file statements that are included conditionally by the preprocessor.

Example

For an example of using this directive in a program, see **assert**.

See Also

#elif, #else, #if, #ifdef, #ifndef, C preprocessor
ANSI Standard, §6.8.1

#if — Preprocessing Directive

Include code conditionally

The preprocessing directive **#if** tells the preprocessor that if *constant-expression* is true (i.e., that it evalutes to a value other than zero), then include the following lines of code within the program until it reads the next **#elif**, **#else**, or **#endif** directive.

The *constant-expression* must be an integral expression, and it cannot include a **sizeof** operator, a cast, or an enumeration constant. All macro substitutions are performed upon the *constant-expression* before it is evaluated. All integer constants are treated as long objects, and are then evaluated. If *constant-expression* includes character constants, all escape sequences are converted into characters before evaluation.

If *constant-expression* is an undefined symbol, the preprocessor treats it the same as it would a false statement.

See Also

#elif, #else, #endif, #ifdef, #ifndef, C preprocessor, defined
ANSI Standard, §6.8.1

#ifdef — Preprocessing Directive

Include code conditionally

The preprocessing directive **#ifdef** checks whether *identifier* has been defined as a macro name. If *identifier* has been defined as a macro, then the preprocessor includes *group* within the program, up to the next **#elif**, **#else**, or **#endif** directive. If *identifier* has not been defined, however, then *group* is skipped.

An **#ifdef** directive can be followed by any number of **#elif** directives, by one **#else** directive, and must be followed by an **#endif** directive.

Example

For an example of using this directive in a program, see **assert**.

See Also

#elif, #else, #endif, #if, #ifndef, C preprocessor, defined
ANSI Standard, §6.8.1

#ifndef — Preprocessing Directive

Include code conditionally

The preprocessing directive **#ifndef** checks whether *identifier* has been defined as a macro name. If *identifier* has *not* been defined as a macro, then the preprocessor includes *group* within the program, up to the next **#elif**, **#else**, or **#endif** directive. If *identifier* has been defined, however, then *group* is skipped.

An **#ifndef** directive can be followed by any number of **#elif** directives, by one **#else** directive, and by one **#elif** directive.

See Also

#elif, #else, #endif, #if, #ifndef, C preprocessor, defined
ANSI Standard, §6.8.1

#include — Preprocessing Directive

Read another file and include it
#include <*file***>**
#include "*file***"**

The preprocessing directive **#include** tells the preprocessor to replace the directive with the contents of *file*.

The directive can take one of two forms: either the name of the file is enclosed within angle brackets (**<***header***.h>**), or it is enclosed within quotation marks (**"***header***.h"**). Angle brackets tell **cpp** to look for *file***.h** in the directories named with the **-I** options to the **cc** command line, and then in the standard directory. Quotation marks tell **cpp** to look for *file***.h** in the source file's directory, then in directories named with the **-I** options, and then in the standard directory.

Most often, the file being included is a *header*, which is a file that contains function prototypes, macro definitions, and other useful material; as its name implies, it most often appears at the head of a program. The header name must be a string of characters, possibly followed by a period '.' and a single letter, usually (but not always) 'h'. A header name may have up to 12 characters to the left of the period, and names may be case sensitive.

#include directives may be nested up to at least eight deep. That is to say, a file included by an **#include** directive may use an **#include** directive to include a third file; that third file may also use a **#include** directive to include a fourth file; and so on, up to at least eight files.

Note, too, that a subordinate header file is sought relative to the original source file, rather than relative to the header that calls it directly. For example, suppose that a file **example.c** resides in directory **/v/fred/src**. If **example.c** contains the directive **#include <header1.h>**. The operating system will look for **header1.h** in the standard directory, **/usr/include**. If **header1.h** includes the directive **#include <../header2.h>** then COHERENT looks for **header2.h** not in directory **/usr**, but in directory **/v/fred**.

A **#include** directive may also take the form **#include** *string*, where *string* is a macro that expands into either of the two forms described above.

See Also

header files, C preprocessor
ANSI Standard §6.8.2

Notes

If the header's name is enclosed within quotation marks note that the name is *not* a string literal, although it looks exactly like one. Thus, a backslash '\' does not introduce an escape character.

Trigraphs that occur within a **#include** directive are substituted, because they are processed by an earlier phase of translation than are **#include** directives.

The mapping provided for included files may map a given name either to an actual file, or to a member in a partitioned data set.

#line — Preprocessing Directive

Reset line number
#line *number newline*
#line *number filename newline*
#line *macros newline*

#line is a preprocessing directive that resets the line number within a file. The ANSI Standard defines the line number as being the number of newline characters read, plus one.

#line can take any of three forms. The first, **#line** *number*, resets the current line number in the source file to *number*. The second, **#line** *number filename*, resets the line number to *number* and changes the name of the file to *filename*. The third, **#line** *macros*, contains macros that have been defined by earlier preprocessing directives. When the macros have been expanded by the preprocessor, the **#line** instruction will then resemble one of the first two forms and be interpreted appropriately.

See Also

C preprocessor
ANSI Standard, §6.8.4

Notes

Most often, **#line** is used to ensure that error messages point to the correct line in the program's source code. A program generator may use this directive to associate errors in generated C code with the original sources. For example, the program generator **yacc** uses **#line** instructions to link the C code it generates with the **yacc** code written by the programmer.

#pragma — Preprocessing Directive

Perform implementation-specific preprocessing

#pragma is the C preprocessing directive that triggers implementation-specific behavior. The ANSI Standard demands that every conforming implementation of C document what **#pragma** does.

COHERENT recognizes one use of **#pragma**:

```
#pragma align [n]
```

This directive permits COHERENT to conform to the Intel Binary Compatability Standard (BCS), which specifies alignment requirements for **struct**s.

The BCS requires that a **struct** be aligned consistently with the alignment of its most strictly aligned member. For example, the structure

```
struct s {
      short s_s1;
      int   s_i;
      short s_s2;
};
```

must put member **s_i** at offset 4, not 2 (because **int** is dword-aligned). If you have an array of **struct s** objects, the second will be at offset 12, not 10 (or 8), because **struct s** itself must also be dword-aligned.

This, unfortunately, creates problems with existing compiled code, and with some standards, e.g., COFF. For example, a **struct filsys** (a COHERENT file system, e.g., on a floppy or hard disk) is defined in **<sys/filsys.h>** as starting out just like the above:

```
struct filsys {
      unsigned short    s_isize;
      daddr_t           s_fsize;
      short       s_nfree;
      ...
};
```

Because **daddr_t** is **long**, COHERENT would compile this and expect to find **s_fsize** at offset 4 (not 2) and **s_nfree** at offset 8 (not 6); but this is not where the bits actually fall on an existing file system. So we circumvent the BCS with **#pragma align**. The directive **#pragma align** *n* means "align objects on *n*-byte boundaries, at most," and **#pragma align** means "restore default alignment." Thus, **<sys/filsys.h>** is edited to read:

```
struct filsys {
      unsigned short    s_isize;
#pragma align 2
      daddr_t           s_fsize;
#pragma align
      short       s_nfree;
      ...
};
```

and the compiler thinks the struct members fall at offsets 0, 2 and 6, which preserves compatibility with existing binary objects.

See Also

cpp, C preprocessor
ANSI Standard, §6.8.6

#undef — Preprocessing Directive

Undefine a macro
#undef *identifier*

The preprocessing directive **#undef** tells the C preprocessor to disregard *identifier* as a macro. It undoes the effect of the **#define** directive.

See Also

#define, C preprocessor
ANSI Standard, §6.8.3

__DATE__ — Manifest Constant

Date of translation

__DATE__ is a preprocessor constant that is defined by the C preprocessor. It represents the date that the source file was translated. It is a string literal of the form

```
"Mmm dd yyyy"
```

where **Mmm** is the same three-letter abbreviation for the month as is used by **asctime**; **dd** is the day of the month, with the first **d** being a space if translation occurs on the first through the ninth day of the month; and **yyyy** is the current year.

The value of **__DATE__** remains constant throughout the processing of the a module of source code. It may not be the subject of a **#define** or **#undef** preprocessing directive.

Example

The following prints the preprocessor constants set by the ANSI standard.

```
#include <stddef.h>
#include <stdio.h>
```

```
main(void)
{
      printf("Date: %s\n", __DATE__);
      printf("Time: %s\n", __TIME__);
      printf("File: %s\n", __FILE__);
      printf("Line No.: %d\n", __LINE__);

      printf("ANSI C? ");
#ifndef __STDC__
      printf("no0);
#else
      printf("ANSI C? %s(%d)0, __STDC__ ? "Yes" : "No", __STDC__);
#endif /* _defined(__STDC__) */

      exit(EXIT_SUCCESS);

}
```

See Also

__FILE__, __LINE__, __STDC__, __TIME__, manifest constant
ANSI Standard, §6.8.8

__FILE__ — Manifest Constant

Source file name

__FILE__ is a preprocessor constant that is defined by the C preprocessor. It represents, as a string constant, the name of the current source file being translated.

__FILE__ may not be the subject of a **#define** or **#undef** preprocessing directive, but it may be altered with the **#line** preprocessing directive.

Example

For an example of how to use **__FILE__** in a program, see **__DATE__**.

See Also

__DATE__, __LINE__, __STDC__, __TIME__, manifest constant
ANSI Standard, §6.8.8

__LINE__ — Manifest Constant

Current line within a source file

__LINE__ is a preprocessor constant that is defined by the C preprocessor. It represents the current line within the source file. The ANSI standard defines the current line as being the number of newline characters read, plus one.

__LINE__ may not be the subject of a **#define** or **#undef** preprocessing directive.

Example

For an example of how to use **__LINE__** in a program, see **__DATE__**.

See Also

__DATE__, __FILE__, __STDC__, __TIME__, manifest constant
ANSI Standard, §6.8.8

__STDC__ — Manifest Constant

Mark a conforming translator

__STDC__ is a preprocessor constant that is defined by the C preprocessor. If it is defined to be equal to one, then it indicates that the translator conforms to the ANSI standard.

The value of **__STDC__** remains constant throughout the entire program, no matter how many source files it comprises. It may not be the subject of a **#define** or **#undef** preprocessing directive.

Example

For an example of using **__STDC__** in a program, see **__DATE__**.

See Also

__DATE__, __FILE__, __LINE__, __TIME__, manifest constant
ANSI Standard, §6.8.8

Notes

Many users incorrectly attempt to use the construction

```
#ifdef __STDC__
```

instead of the correct form:

```
#if __STDC__
```

These constructions give different results because **__STDC__** is defined, but it is defined to a value of zero, in keeping with the fact that COHERENT C does not yet conform to the ANSI standard.

To help users avoid this error, COHERENT does not define **__STDC__** at all.

__TIME__ — Manifest Constant

Time source file is translated

__TIME__ is a preprocessor constant that is defined by the C preprocessor. It represents the time that a source file is translated. It is a string literal of the form:

```
"hh:mm:ss"
```

This is the same format used by the function **asctime**.

The value of this preprocessor constant remains constant throughout the processing of the translation unit. It may not be the subject of a **#define** or **#undef** preprocessing directive.

Example

For an example of how to use **__TIME__** in a program, see **__DATE__**.

See Also

__DATE__, __FILE__, __LINE__, __STDC__, manifest constant
ANSI Standard §6.8.8

_exit() — System Call (libc)

Terminate a program
#include <unistd.h>
void _exit(*status***) int** *status***;**

The system call **_exit()** terminates a program directly. It returns *status* to the calling program, and exits. Unlike the library function **exit()**, **_exit()** does not perform extra termination cleanup, such as flushing buffered files and closing open files.

_exit() should be used only in situations where you do *not* want buffers flushed or files closed. For example, you may wish to call **_exit()** if your program detects an irreparable error condition and you want to "bail out" to keep your data files from being corrupted.

_exit() should also be used with programs that do not use STDIO. Unlike **exit()**, **_exit()** does not use STDIO. This will help you create programs that are extremely small when compiled.

See Also

close(), exit(), EXIT_FAILURE, EXIT_SUCCESS, libc, unistd.h, wait()
POSIX Standard, §3.2.2

Notes

If you do not explicitly set *status* to a value, the program returns whatever value happens to have been in the register EAX. You can set *status* to either **EXIT_SUCCESS** or **EXIT_FAILURE**.

_getwd() — General Function (libc)

Get current working directory name
char *__getwd(__pathname__)**
char **pathname__;**

The *current working directory* is the directory from which file name searches commence when a path name does not begin with '/'. **_getwd()** returns the name of the current working directory. It is useful for processes like spoolers and daemons, which must generate full path names for files.

If you do not have permission to search all levels of the directory hierarchy above the current directory, **_getwd()** cannot obtain the directory name for you.

See Also

chdir(), getcwd(), libc, pwd

Diagnostics

_getwd() returns NULL and writes an error message into *pathname* if an error occurs, e.g., if the current directory cannot be found or if any other error occurs.

Notes

_getwd() is obsolete, and is included for reasons of compatibility. Programmers should use the function **getcwd()** instead.

_getwd() fails if the current directory name is longer than MAXPATH characters (128 characters as defined in header file **<path.h>**). The chunk of memory pointed to by *pathname* must be big enough to hold **MAXPATHLEN** characters plus a trailing NUL.

If **_getwd()** fails, the working directory cannot be restored to its initial value.

The name of this function has been change to **_getwd()** to avoid confusion with the Berkeley UNIX function **getwd()**, which has a different calling sequence.

_tolower() — ctype Function (libc)

Convert characters to lower case
#include <ctype.h>
int _tolower(c**) int** c**;**

The function **_tolower()** converts the character *c* to lower case, and returns the converted character. Unlike the related function **tolower()**, **_tolower()** is not guaranteed to work correctly if handed anything other than an upper-case character, that is, a character for which **isupper()** returns true.

See Also

_toupper(), libc, tolower()

Notes

_tolower() is not part of the ANSI standard; COHERENT includes it only to support old code. You should use **tolower()** instead.

_toupper() — ctype Function (libc)

Convert characters to upper case
#include <ctype.h>
int _toupper(c**) int** c**;**

The function **_toupper()** converts the character *c* to upper case and returns the converted character. Unlike the related function **toupper()**, **_toupper()** is not guaranteed to work correctly if it is passed something other than a lower-case character, that is, any character for which **islower()** returns true.

See Also

_tolower(), libc, toupper()

Notes

_toupper() is not part of the ANSI standard; COHERENT includes it only to support old code. You should use **toupper()** instead.

a.out.h — Header File

Include all COFF header files
#include <coff/a.out.h>

a.out.h includes all header files needed to generate COFF output.

See Also

arcoff.h, file formats, header files
Gircyc, G.R.: *Understanding and Using COFF*. Sebastopol, Calif, O'Reilly & Associates, Inc., 1990.

abort() — General Function (libc)

End program immediately
#include <stdlib.h>
void abort()

abort() terminates a process with a core dump, creating a file called **core**, and prints a message on the screen. It is normally invoked in situations that "should not happen". For example, **malloc()** invokes **abort()** if it discovers a corrupt storage arena.

Where possible, **abort()** executes a machine instruction that causes the processor to trap. If the signal associated with the trap is caught or ignored, the dump will not be produced.

See Also

_exit(), core, exit(), libc, stdlib.h
ANSI Standard, §7.10.4.1
POSIX Standard, §8.1

abs() — General Function (libc)

Return the absolute value of an integer
#include <stdlib.h>
int abs(*n***) int** *n***;**

abs() returns the absolute value of integer *n*. The *absolute value* of a number is its distance from zero. This is *n* if *n***>=0**, and -*n* otherwise.

Example

This example prompts for a number, and returns its absolute value.

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
      extern char *gets();
      extern int atoi();
      char string[64];
      int counter;
      int input;

      printf("Enter an integer: ");
      fflush(stdout);
      gets(string);
```

```
      for (counter=0; counter < strlen(string); counter++) {
            input = string[counter];

            if (!isascii(input)) {
                  fprintf(stderr,
                        "%s is not ASCII\n", string);
                  exit(EXIT_FAILURE);
            }

            if (!isdigit(input))
                  if (input != '-' || counter != 0) {
                        fprintf(stderr,
                              "%s is not a number\n", string);
                        exit(1);
                  }
      }

      input = atoi(string);
      printf("abs(%d) is %d\n", input, abs(input));
      exit(EXIT_SUCCESS);
}
```

See Also

fabs(), floor(), int, libc, stdlib.h
ANSI Standard, §7.10.6.1
POSIX Standard, §8.1

Notes

On two's complement machines, the **abs()** of the most negative integer is itself.

ac — Command

Summarize login accounting information
ac [**-dp**] [**-w** *wfile*][*username ...*]

One of the accounting mechanisms available on the COHERENT system is login accounting, which keeps track of the time each user spends logged into the system. Login accounting is enabled by creating the file **/usr/adm/wtmp**. Thereafter, the routines **date**, **login**, and **init** write raw accounting data to **/usr/adm/wtmp** to record the time, the name of the terminal, and the name of the user for each date change, login, logout, or system reboot.

The command **ac** summarizes the accounting data that have accumulated for your system. By default, it prints the total connect time found in **/usr/adm/wtmp**. If its command line includes a user's login identifier, **ac** prints a summary only of that user's activity.

ac recognizes the following command-line options:

-d Itemize the output into daily periods. **ac** defines a day as beginning at midnight.

-p Print a summary for every user on your system.

-w Read data from *wfile*. By default, **ac** reads its data from **/usr/adm/wtmp**.

See Also

commands, date, init, login, sa, utmp.h

Notes

File **/usr/adm/wtmp** can become very large; therefore, you should truncated it periodically. Special care should be taken if you have enabled login accounting and your system has limited amounts of free disk space.

accept() — Sockets Function (libsocket)

Accept a connection on a socket
#include <sys/types.h>
#include <sys/socket.h>
int accept(*socket***,** *address***,** *addrlen***)**
int *socket***,** ******addrlen***; struct sockaddr** ******address***;**

accept() accepts a connection on a socket. It extracts the first connection request on the queue of pending connections, creates a new socket with the same properties as *socket*, and allocates a file descriptor for the newly created socket. It is used with connection-based types of sockets, currently with **SOCK_STREAM**.

socket gives a file descriptor that identifies a socket. It must have been returned by a call to **socket()**, have been bound to an address by a call to **bind()**, and be listening for connections after a call to **listen()**.

If no connections are pending on the queue and *socket* is not marked as non-blocking, **accept()** blocks the calling process until it can establish a connection. If *socket* is marked non-blocking and no connections are pending on the queue, **accept()** returns an error, as described below. The accepted socket may not be used to accept more connections; however, the original *socket* remains open.

address gives the address of the connecting entity, as known to the "communications layer". Its exact format is dictated by the domain in which communication occurs.

addrlen points to an integer that gives the number of bytes available at *address*. Upon return, that integer contains the number of bytes to which *address* actually points.

The function **select()** can perform the same action as **accept()**: simply select the socket for reading.

If all goes well, **accept()** returns the file descriptor for the accepted socket, which is a non-negative integer. If something goes wrong, **accept()** returns -1 and set **errno** to an appropriate value. The following lists the errors that can occur, by the value to which **accept()** sets **errno**:

EBADF *socket* is somehow invalid.

ENOTSOCK
socket references a file, not a socket.

EOPNOTSUPP
socket references a socket that is not of type **SOCK_STREAM**.

EFAULT
addr contains an illegal address.

EWOULDBLOCK
The socket is marked non-blocking, and no connections are present to be accepted.

Example

For an example of this function, see the Lexicon entry for **libsocket**.

See Also

bind(), connect(), libsocket, listen(), select()

access() — System Call (libc)

Check if a file can be accessed in a given mode
#include <unistd.h>
int access(*filename, mode***) char** **filename***; **int** *mode*;**

access() checks whether a file or directory can be accessed in the mode you wish. *filename* is the full path name of the file or directory you wish to check. *mode* is the mode in which you wish to access *filename*, as follows:

F_OK	File exists
R_OK	Read a file
W_OK	Write into a file
X_OK	Execute a file

The header file **unistd.h** defines these values, which may be logically combined to produce the *mode* argument.

If *mode* is **F_OK**, **access()** tests only whether *filename* exists, and whether you have permission to search all directories that lead to it.

access() returns zero if *filename* can be accessed in the requested mode, and a nonzero value if it cannot. Note that the return value is the opposite of the intuitive value, i.e., zero means success rather than failure.

access() uses the *real* user id and *real* group id (rather than the *effective* user id and *effective* group id), so set user id programs can use it.

Example

The following example checks if a file can be accessed in a particular manner.

```
#include <unistd.h>
#include <stdio.h>

main(argc, argv)
int argc; char *argv[];
{
      int mode;
      extern int access();

      if (argc != 3) {
            fprintf(stderr, "usage: acc dir_name/file_name mode\n");
            exit(EXIT_FAILURE);
      }

      switch (*argv[2]) {
      case 'x':
            mode = X_OK;
            break;

      case 'w':
            mode = W_OK;
            break;

      case 'r':
            mode = R_OK;
            break;

      case 'f':
            mode = F_OK;
            break;

      default:
            fprintf(stderr, "Bad mode. Modes: f, x, r, w\n");
            exit(EXIT_FAILURE);
            break;
      }

      if (access(argv[1], mode))
            printf("file %s cannot be found in mode %d\n", argv[1], mode);
      else
            printf("file %s is accessible in mode %d\n", argv[1], mode);
      exit(EXIT_SUCCESS);
}
```

See Also

libc, path(), unistd.h
POSIX Standard, §5.6.3

Notes

When the superuser **root** executes **access()**, it always returns readable/writable/executable for any file that exists, regardless of permissions.

Note that **access()** used to be declared in header file **<access.h>**. It is now prototyped in header file **<unistd.h>**, to comply with the POSIX standard. **<access.h>** is obsolete and has been dropped from COHERENT beginning with release 4.2.

acct() — System Call (libc)

Enable/disable process accounting
#include <acct.h>
acct(*file***)**
char **file***file***;**

Process accounting records who initiates each system process and how long each process takes to execute. These data can be analyzed, to administer the system most efficiently.

The system call **acct()** enables or disables process accounting. If *file* is not NULL, then accounting is turned on; if

file is NULL, however, then process accounting is turned off.

It is usual, but not necessary, that *file* be **/usr/adm/acct**. *file* must exist. When enabled, the system appends a raw accounting data record in the format described by **acct.h** to *file* as each process terminates.

acct() is restricted to the superuser.

See Also

ac, acct.h, accton, exit(), libc, sa, times(),

Diagnostics

Successful calls return zero. **acct()** returns -1 for errors, such as nonexistent *file* or invocation by a user other than the superuser.

Notes

The system writes accounting records for a process only when the process exits. Processes that never terminate and processes running at the time of a system crash do not produce accounting information.

acct.h — Header File

Format for process-accounting file
#include <acct.h>

Process accounting is a feature of the COHERENT system that allows it record what processes each user executes and how long each process takes. These data can be used to track how much each user uses the system.

The function **acct()** turns process accounting off or on. When process accounting has been turned on, the COHERENT system writes raw process-accounting information into an accounting file as each process terminates. Each entry in the accounting file, normally **/usr/adm/acct**, has the following form, as defined in the header file **acct.h**:

```
struct acct {
        char        ac_comm[10];
        comp_t      ac_utime;
        comp_t      ac_stime;
        comp_t      ac_etime;
        time_t      ac_btime;
        short       ac_uid;
        short       ac_gid;
        short       ac_mem;
        comp_t      ac_io;
        dev_t       ac_tty;
        char        ac_flag;
};

/* Bits from ac_flag */
#define     AFORK       01      /* has done fork, but not exec */
#define     ASU         02      /* has used superuser privileges */
```

Every time a process calls **exec()**, the contents of **ac_comm** are replaced with the first ten characters of the file name. The fields **ac_utime** and **ac_stime** represent the CPU time used in the user program and in the system, respectively. **ac_etime** represents the elapsed time since the process started running, whereas **ac_btime** is the time the process started. The effective user id and effective group id are **ac_uid** and **ac_gid. ac_mem** gives the average memory usage of the process. **ac_io** gives the number of blocks of input-output. **ac_tty** gives the controlling typewriter device major and minor numbers.

For some of the above times, the **acct** structure uses the special representation **comp_t,** defined in the header file **types.h**. It is a floating point representation with three bits of base-8 exponent and 13 bits of fraction, so it fits in a **short** integer.

See Also

acct(), accton, header files, sa

accton — Command

Enable/disable process accounting
/etc/accton [*file*]

One of the accounting mechanisms available on the COHERENT system is *process accounting*, Process accounting records each process, who initiates it, and how long it takes to execute.

The command **accton** turns process account on or off. To turn on process accounting, issue the command **accton** followed by a *file* argument; COHERENT then begin to write accounting data into *file*, By convention, *file* should be **/usr/adm/acct**. To turn off process accounting, issue the command **accton** without any arguments.

The command **sa** summarizes the data that will have been written into *file*.

See Also

ac, acct, acct.h, commands, init, sa

Notes

As the accounting file can become very large, you should truncate that file from time to time. You should take extra care to monitor the growth of that file should you enable process accounting on a system with a limited amount of free disk space.

acos() — Mathematics Function (libm)

Calculate inverse cosine
#include <math.h>
double acos(*arg***) double** *arg***;**

acos() calculates the inverse cosine. *arg* should be in the range of -1.0, 1.0. It returns the result, which is in the range of from zero to π radians.

Example

This example demonstrates the mathematics functions **acos()**, **cabs()**, and **tan()**.

```
#include <errno.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define display(x) dodisplay((double)(x), #x)

dodisplay(value, name)
double value; char *name;

{
      if (errno)
            perror(name);
      else
            printf("%10g %s\n", value, name);
      errno = 0;
}

main()
{
      extern char *gets();
      double x;
      char string[64];

      for(;;) {
            printf("Enter number: ");
            if(gets(string) == NULL)
                  break;

            x = atof(string);
            display(x);
            display(acos(cos(x)));
            display(cabs(sin(x),cos(x)));
      }
}
```

See Also

cos(), errno, errno.h, libm, perror()
ANSI Standard, §7.5.2.1
POSIX Standard, §8.1

add_history() — Editing Function (libedit)

Add a line to history buffer
void add_history(*line***)**
char **line*****;**

The function **add_history()** adds *line* to a "history" buffer, from which it can be retrieved by the function **readline()**. *line* must have been returned by **readline()**.

See Also

libedit, readline()

address — Definition

An **address** is the location where an item of data is stored in memory.

On the i8086, a physical address is a 20-bit number. The i8086 builds an address by left-shifting a 16-bit segment address by four bits, and then adding it to a 16-bit offset address. The segment address points to a particular chunk of memory. The i8086 uses four segment registers, each of which governs a different portion of a program, as follows:

CS	Address of code segment
DS	Address of data segment
ES	Address of "extra" segment
SS	Address of stack segment

SMALL-model programs use only the offset address; hence, their pointers are only 16 bits long, equivalent to an **int**. LARGE-model programs use both segment and offset addresses. Their addresses are 20 bits long, which must be stored in a 32-bit pointer, equivalent to a **long**. COHERENT 286 supports SMALL model.

On the i80386, addresses start as 32 bits. Segment registers are used to look up a segment descriptor. The descriptor's base then defines the address within a four-gigabyte virtual address space. The page tables are then used to translate this to a physical address. For details, see the *Intel 386 Programmers Manual*.

On the M68000, an address is simply a 24-bit integer that is stored as a 32-bit integer. The upper eight bits are ignored; this is not true with the more advanced microprocessors in this family, such as the M68020. The M68000 uses no segmentation; memory is organized as a "flat address space," with no restrictions set on the size of code or data.

On machines with memory-mapped I/O, such as the 68000, some addresses may be used to control or communicate with peripheral devices.

Example

The following printes the address and contents of a given byte of memory.

```
#include <stdio.h>

main()
{
      char byte = 'a';
      printf("Address == %x\tContents == \"%c\"\n",
            &byte, byte);
}
```

See Also

data formats, pointer, Programming COHERENT

Administering COHERENT — Overview

To administer a COHERENT system, you must know how to do the following:

- Perform backups, manage archives and purge old files.
- Set up and manage complex system, such as mail, UUCP, and the print spooler.
- Attach peripheral devices, such as terminals, modems, and printers.
- Install third-party software.
- Configure the kernel, and add or configure device drivers.
- Act as a resource person for other users.

Overview Lexicon Articles

Many users who have purchased COHERENT for their personal use will find some of these tasks to be confusing or daunting. This is especially true if they have had no previous exposure to UNIX or similar operating systems. Such a person will find the following Lexicon articles to be helpful:

backups
When and how to back up your system, using tape or floppy disks.

booting
How booting works. In particular, it shows how to boot a kernel other than the default kernel.

CD-ROM
Introduce how to use CD-ROM drives under COHERENT.

console
This introduces the device **/dev/console**. It also lists the many escape sequences with which you can change the appearance and behavior of the console.

device drivers
The suite of device drivers available under COHERENT. This article also gives a

floppy disks
Information about floppy disks. This describes the floppy-disk devices available under COHERENT, how to format floppy disks, and how to record data on a floppy disk using a COHERENT file systems, a **tar** archive, or an file systems.

hard disk
This gives basic information about hard disks. In particular, it discusses the devices by which hard disks are accessed, and how to partition a hard disk.

IRQ This article lists the IRQs available on the IBM PC.

kernel This introduces the *kernel*, which is the master program of COHERENT. It also gives examples of how to configure and patch the kernel.

keyboard
This introduces the suite of keyboard drivers available for the COHERENT keyboard.

lpsched
This command is the daemon for the **lp** print spooler. For an overview of **lp** and the other print spoolers, see the Lexicon entry for **printer**.

mail This gives an overview of the COHERENT mail system — both commands and configuration files.

modem
This describes how to add a modem to your COHERENT system. It also introduces the communications programs available under COHERENT.

printer This describes how to add a printer to your system. It also gives an overview of the various print spoolers available with COHERENT, and how to configure each to work with a variety of printers.

RS-232
This presents the design and pin-out of the RS-232 plug, which is the standard plug for serial and parallel ports on the IBM PC and its clones.

security
This article discusses the problem of system security — that is, how to let your users but keep the "crackers" out.

tape This introduces tape devices. It describes how to access tape, and goes into some detail on how to manage tape archives.

terminal
This describes how to plug a terminal into your system, and configure it correctly.

tboot The tertiary boot is the program that loads the COHERENT kernel into memory and launches it. This article describes it. You probably will never need to work with **tboot**— but you never know.

virtual console
COHERENT supports virtual consoles, whereby several console sessions can be run on the same physical device. This describes how to set up and manage virtual consoles on your system.

System Files

The COHERENT system is controlled by *system files* and *daemons*. System files contain the information that controls the minute-to-minute operation of the COHERENT system. A daemon is a program that the system runs to manage a peripheral device or perform some other task that does not require the intervention of a human. COHERENT's system files and daemons are described in the following Lexicon articles:

/usr/lib/mail/aliases
This file holds the aliases by which your system is known to other systems.

atrun This daemon executes other commands at a preset time. A user can use the command **at** to spool another command for execution at a later time.

/etc/boottime
This file records the date and time your system was last booted.

/etc/brc
COHERENT executes this script when your system enters single-user mode. It performs maintenance chores.

/etc/checklist
This file lists the devices to check with **fsck** when you boot COHERENT.

/usr/lib/mail/config
This file performs overall configuration of **smail**.

/usr/lib/uucp/config
This file performs overall configuration of UUCP.

/usr/spool/mlp/controls
This file holds the data base for the MLP print spooler.

core This Lexicon entry describes the format of a core file — that, the file that a program dumps when it fails catastrophically.

/etc/cron
This daemon reads a data base of commands to execute periodically, and executes each when its time comes round at last.

/etc/d_passwd
This file holds the passwords that control access to your system via peripheral devices. For example, you can set an extra password in this file for all users who may attempt to log in via modem.

/usr/lib/uucp/dial
This file holds the information by which UUCP dials a modem.

/etc/dialups
This file names every peripheral device that requires an additional password.

/usr/lib/mail/directors
Name the director routines that **smail** uses, and configure them.

LEXICON

/etc/domain
This file names the mail domain to which your system belongs.

/etc/drvld.all
This file names the loadable drivers to load when you boot your system.

$HOME/.forward
This File lets you set a forwarding address for mail.

/etc/getty
This daemon initializes a serial port, watches the port, and assists any user who attempts to log into your system.

/etc/group
This file define groups of users on your system.

/etc/hosts
This file gives the name and address of every host on your local network.

/etc/hosts.equiv
This file names "equivalent hosts" on your local network — that is, the hosts that have identical (or nearly identical) sets of users.

/etc/hosts.lpd
This file holds the name and domain of your local host.

/usr/lib/hpd
This daemon is a spooler daemon for a laser printer.

/etc/inetd.conf
This file configures the Internet daemons.

/etc/init
Command helps to bring COHERENT into multi-user mode. It also helps users to log in.

$HOME/.kshrc
This script configures the Korn shell to suit your tastes.

$HOME/.lastlogin
This file records the date and time you last logged in to your COHERENT system.

login This command logs a user in to your COHERENT system. Its Lexicon article also describes the entire convoluted process of managing an enabled port and logging a user in.

/etc/default/login
This file sets default values for logging in.

/usr/adm/loginlog
This file logs failed attempts to log in.

/etc/logmsg
This file holds the COHERENT login prompt. If you do not like the prompt

```
Coherent 386 login:
```

and a beep, you can change it by editing this file.

/usr/lib/lpd
This daemon manages the MLP print spooler.

/etc/conf/mdevice
This file describes the device drivers currently available on your system.

/etc/mnttab
This file holds the mount table — that is, the table that describes which file systems are mounted, and what directories they are mounted on.

/etc/motd
This file holds the message of the day — a message that is printed on each user's terminal when she logs in.

/etc/mount.all
This file names the disk devices to mount when your system enters multi-user mode.

/etc/conf/mtune
This file names the set of variables in the kernel and its device drivers that you can "tune," to modify the kernel's behavior.

/etc/networks
This file describes remote networks that your system can contact.

/etc/nologin
This file, if it exists, prevents users from logging in. It is used during special periods of time, such as when you wish to shut the system down.

/etc/passwd
This file describes every user who has permission to log into your system.

/usr/lib/mail/paths
This file holds the information by which your system routes mail to other systems.

/usr/lib/uucp/port
This file describes the serial ports through which UUCP can dial out from your system.

/etc/profile
This script sets up the default environment for each user on your system.

$HOME/.profile
This script holds commands that are executed when a given user logs in to your COHERENT system.

/etc/protocols
This file names the Internet protocols that your system supports.

/usr/bin/ramdisk
This script lets you build a RAM disk on your system.

/etc/rc
This script is executed when your system enters multi-user mode. It normally performs standard housekeeping chores.

/usr/lib/mail/routers
This file names the routing programs that **smail** uses, and configures them.

/etc/conf/sdevice
This file holds the information by which device drivers are configured when you build a kernel.

/etc/serialno
This file holds your system's serial number, which you entered when you first installed COHERENT.

/etc/services
This file lists the Internet services that your system supports.

/etc/shadow
This file holds each user's password.

/etc/conf/stune
This file sets the values of tunable kernel variables.

/usr/lib/uucp/sys
This file describes the remote systems that you can contact via UUCP, and how to contact them.

term This Lexicon article describes the format of a compiled **terminfo** file.

/etc/termcap
This file holds **termcap** terminal-description data base.

terminfo
This article describes the **terminfo** terminal-description language. Its data base is kept in directory **/usr/lib/terminfo**.

/usr/lib/mail/transports
This file names the transport routines that **smail** can use, and configures them.

/etc/trustme
This file names of trusted users — that is, users who can log in even if file **/etc/nologin** exists.

/etc/ttys
This file describes terminal ports — that is, the ports via which a user can log in. This includes both serial ports and pseudo-ttys.

/etc/update
This daemon periodically flushes all buffered information to disk.

/etc/usrtime
This file holds the time, day of the week, and terminal line by which each user can log into your COHERENT system.

/etc/utmp
This file notes every login event that has not yet concluded — that is, a user has logged in but not logged out again. You can examine this file to see who is using your system at this moment.

/etc/uucpname
This file sets your system's UUCP name — that is, the name by which it is known to all other systems.

/etc/default/welcome
This script is executed whenever a user logs in for the first time. It gives the new user some basic information and advice.

/usr/adm/wtmp
This file notes every login event that has concluded — that is, a user has logged in and logged out again. You can examine this file to see who has logged into your system in the past, and for how long.

Finally, the following header files also hold information on file formats:

acct.h. . . . . Format for process-accounting file
ar.h . . . . . Format for archive files
canon.h . . . . . Portable layout of binary data
coff.h . . . . . Define format of COHERENT 386 objects
l.out.h . . . . . Define format of COHERENT 286 objects
mtab.h . . . . . Currently mounted file systems
utmp.h . . . . . Login accounting information

For a fuller description of each file and its contents, see its entry in the Lexicon.

See Also

COHERENT, Programming COHERENT, Using COHERENT

alarm() — System Call (libc)

Set a timer
#include <unistd.h>
alarm(*seconds***)**
unsigned *seconds***;**

alarm() sets a timer. After *seconds*, the COHERENT kernel sends signal **SIGALRM** to the process that invoked **alarm()**. Setting *seconds* to zero turns off the alarm timer.

By default, signal **SIGALRM** terminates the process. However, a program can invoke the system call **signal()** to catch this signal, or ignore it. Because of scheduling variation and the one-second granularity, the action of **alarm()** is predictable only to within one second.

alarm() is useful for such things as timeouts. For example, a process on a dial-in port might hang up the line after a sufficient time has elapsed with no user response.

alarm() returns the previous alarm value, which represents the time remaining from the previous call. Time remaining is superseded by the new alarm value.

See Also

libc, signal(), sleep(), unistd.h
POSIX Standard, §3.4.1

Notes

A process can set only one alarm at a time.

alias — Command

Set an alias
alias [*name***[=***value ...***]]**

The command **alias** is used by the Korn shell **ksh** to set or display an alias.

When called without an argument, **alias** lists all aliases that have been set so far. When called with a *name* argument alone, it lists alias of *name*, assuming one has been set.

When called with one or more arguments of the form *name*=*value*, it established *name* as an alias for the command *value*. For example, the command

```
alias FOO="echo bar"
```

establishes the string **FOO** as an alias for the command **echo bar**. Thereafter, when you type **FOO** on the shell's command line, it will execute the command **echo bar** and so echo the string **bar** on your terminal.

The Korn shell sets a number of aliases by default. See the Lexicon entry for **ksh** for a list of these aliases and their settings.

See Also

commands, ksh, unalias

aliases — System Administration

File of users' aliases
/usr/lib/mail/aliases

File **/usr/lib/mail/aliases** holds aliases for users' addresses — either on your system, or on other systems. The command **smail** reads this file when it figures out how to deliver a mail message.

An *alias* is a "nickname" for a user. Once you have established an alias for a user, you can use that alias to send mail to her; this spares you the trouble of typing that person's convoluted e-mail address. An alias can also name an entire group of users; when you use the alias to send a mail message, every person in the group receives a copy.

The format of each alias is

```
alias_name:           target
```

where *alias_name* gives the alias to which you mail your message, and *target* is name to which where **smail** actually directs the message. *target* can be a login identifier on your local system; a mail address of a user on another system; or a cluster of users either on your system or on remote systems.

For example, consider the user whose e-mail address is **ivan@lepanto.mwc.com**. If you add the entry

```
ivan: ivan@lepanto.mwc.com
```

to file **/usr/lib/mail/aliases**, then whenever you send mail to **ivan**, the routing program **smail** will automatically "expand" the address from **ivan** to **ivan@lepanto.mwc.com**, and dispatch the message properly. This spares you needless work, and eliminates the errors that would occur if you typed long addresses by hand.

Please note that **smail** ignores differences in case when it compares a name with an alias. If a line begins with a white-space character, **smail** assumes that that line is a continuation of the previous line. **smail** ignores strings within parentheses, as well as any text that appears after the pound sign '#'. Thus, you can use '#' to embed a comment within **aliases**.

Examples

The following gives an example form of **aliases**:

```
# this whole line is a comment
```

```
# "mail programmers" sends mail to local users joe, jack, and bill
programmers:          joe jack bill

# same as above
programmers:          joe jack
                      bill

# same as above
programmers           joe jack
                      bill

# same as above
programmers           joe   # Joe Smith
                      jack  # Jack Thomas
                      bill  # Bill Williams

# and yet another way; note use of parentheses to comment text
programmers           joe (Joe Smith) jack (Jack Thomas)
                      bill (Bill Williams)

# send a message to someone on another system.
# this uses ''bang-path'' addressing
joe:                  boston!widget!js

# send a message to users on both your and another system
programmers:          boston!widget!js   # Joe Smith
                      chicago!gadget!jt  # Jack Thomas
                      bill               # Bill Williams

# all members of "programmers" group work at site "widget"
programmers!widget            joe jack bill
```

To tell **smail** to use the contents of another file to expand an alias, use the following form:

```
fredlist              :include:/usr/lib/mail/fredlist
```

smail adds each entry in **/usr/lib/mail/fredlist** to the alias for **fredlist**.

You can also tell **smail** to read another alias file, and include its contents in the list of aliases to be expanded. For example, the following instruction

```
:include:/usr/lib/mail/morealiases
```

when embedded within **/usr/lib/mail/aliases**, tells **smail** to add the contents of **/usr/lib/mail/morealiases** to those of **/usr/lib/mail/aliases** as a regular alias file.

All aliases are recursive, so you must be careful when you define them. For example, the entries

```
bill:          joe
joe:           bill
```

causes an infinite loop. **smail** attempts to detect infinite loops, and to guess what you intended to do. The following example illustrates how you can use an alias to deliver mail to a remote user as well as to a local user who has the same name as the alias being expanded. **smail** expands the alias

```
mylogin:              mypc!mylogin  mylogin
```

to

```
mypc!mylogin mylogin
```

even though the second occurrence of **mylogin** matches the alias name.

Both forms of file inclusion are recursive, too, and may lead to infinite loops if handled carelessly.

See Also

Administering COHERENT, mail [overview], smail

Notes

Beginning with release 4.2.14 of COHERENT, **smail**'s aliases are kept in the form of a DBM data base. This is a simple data base that uses a hash table to speed the retrieval of information. If you change your file of aliases, you must invoke either the command **newaliases** or the command **smail -bi** to rebuild the binary data base of aliases.

For details on what a DBM data base is, see the Lexicon entry for **libgdbm**. For details on how to use **newaliases** or **smail**, see their respective entries in the Lexicon.

alignment — Definition

Alignment or packing of fields within a structure

Alignment refers to the fact that some microprocessors require the address of a data entity to be *aligned* to a numeric boundary in memory so that *address* modulo *number* equals zero. For example, the M68000 and the PDP-11 require that an integer be aligned along an even address, i.e., *address***%2==0**. In the MS-DOS world, this is called "packing".

Generally speaking, alignment is a problem only if you write programs in assembly language. For C programs, COHERENT ensures that data types are aligned properly under foreseeable conditions. You should, however, beware of copying structures and of casting a pointer to **char** to a pointer to a **struct**, for these could trigger alignment problems.

Processors react differently to an alignment problem. On the VAX or the i8086, it causes a program to run more slowly, whereas on the M68000 it causes a bus error.

See Also

#pragma, data types, ld, Programming COHERENT

Notes

The COHERENT preprocessor instruction **#pragma** lets you set alignment to conform to Intel's Binary Compatibility Standard (BCS). For details, see the Lexicon entry for **#pragma**.

alloc.h — Header File

Define the allocator
#include <sys/alloc.h>

alloc.h defines manifest constants and structures that are used internally with memory allocation.

See Also

header files

Notes

This header file is obsolete, and will be dropped from a future release of COHERENT. Its use is strongly discouraged.

alloca() — General Function (libc)

Dynamically allocate space on the stack
alloca(*memory***)**
int *memory***;**

The function **alloca()** allocates *memory* number of bytes dynamically on the stack. The allocated memory disappears automatically as soon as the program exits from the function within which the memory was allocated.

For example, consider the function:

```
foo(some_string)
char *some_string;
{
      char *cp;
      . . .
      cp = alloca(strlen(some_string) + 1);
      strcpy(cp, some_string);
      . . .
}
```

Here, the call to **alloca()** allocates enough space upon the stack for **some_string** plus the terminating NUL character. When function **foo()** returns, the allocated memory vanishes.

This routine is popular in Berkeley and GNU circles because it is much faster than **malloc()**, and the programmer does not need to call **free()** to de-allocate the memory.

See Also

calloc(), libc, malloc(), realloc()

almanac — Command

Print an almanac entry for this date
almanac [*month day***]**

The command **almanac** prints on the standard output an almanac entry of noteworthy births, deaths, and events that occurred on this date. *month* and *day* give the date whose listing you wish to see. *month* must be the name of the month. For example, the command

```
almanac November 23
```

prints something like the following on your screen:

```
BIRTHS:
1221: Alfonso X (el Sabio), monarch and music collector, Toledo.
1876: Manuel de Falla, composer, Cadiz.
1887: William Henry Pratt (Boris Karloff), actor.
1888: Adolph "Harpo" Marx, comedian & musician, New York City.
DEATHS:
1585: Thomas Tallis, composer, Greenwich.
EVENTS:
1923: Height of German inflation: 4.2 trillion marks to the dollar.
1935: First "Porky Pig" cartoon premieres.
```

If you do not supply any arguments on the command, **almanac** prints an almanac listing for today.

almanac reads its information from the files **almanac.birth**, **almanac.death**, and **almanac.event**, which are kept in directory **/usr/games/lib**. Each has the same format: the date encoded by the first three letters of the name of the month (with an initial capital letter), followed by the day of the month, followed by the body of the entry. For example:

```
Nov 23 1221: Alfonso X (el Sabio), monarch and music collector, Toledo.
```

You are encouraged to modify these files to suit your tastes and interests.

See Also

commands

Notes

almanac does not check for bogus dates before it reads its data files. It also is quite rigid in how it expects its data base to be laid out.

The data files reflect the tastes of the person who compiled them, and can be rather idiosyncratic.

ANSI — Definition

Standards for information

The American National Standards Institute (ANSI) includes a standards committee called X3, which writes and publishes standards for information-processing systems. Relevant ANSI standards include the following:

X3.4-1977
Code for Information Interchange

X3.64-1979
Additional Controls for Use with ANS Code for Information Interchange

X3.159-1989
Programming Language C

Published ANSI standards are available from:

American National Standards Institute, Inc.
1430 Broadway
New York, NY 10018

See Also

C language, POSIX Standard, Programming COHERENT,
The C Language, tutorial
Mark Williams Company: *ANSI C: A Lexical Guide*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1988.

apropos — Command

Find manual pages on a given topic
apropos *topic* **[** *topic* **...]**

This command implements a simplified version of the Berkeley command **apropos**. It prints every line in the file **man.index** that contains a *topic*. In this way, you can find what manual pages are available on a given topic. For example, the command

```
apropos daemon
```

prints something like the following:

```
daemon          Definition
hpd             Spooler daemon for laser printer
lpd             Spooler daemon for line printer
lpshut          Turn off the printer daemon despooler
```

You can also use **apropos** to nudge your memory when you cannot recall the name of a given command or library function.

apropos normally reads its information from the index files kept in directory **/usr/man**. **apropos** assumes that every file in that directory that ends in the string **.index** is an index file. For details on index files and their format, see the Lexicon entry for **man**.

If the environmental variable **MANPATH** is set, **apropos** searches the index files in each directory that it names. **MANPATH** must name one or more directories, with directories separated by a colon ':'.

Files

/usr/man/*.index

See Also

commands, help, man, Using COHERENT

ar — Command

The librarian/archiver
ar *option* **[***modifier***][***position***]** *archive* **[***member ...***]**

The librarian **ar** edits and examines libraries. It combines several files into a file called an *archive* or *library*. Archives reduce the size of directories and allow many files to be handled as a single unit. The principal use of archives is for libraries of object files. The linker **ld** understands the archive format, and can search libraries of object files to resolve undefined references in a program.

Options and Modifiers

The mandatory *option* argument consists of one of the following command keys:

d Delete each given *member* from *archive*. The **ranlib** header is updated if present.

m Move each given *member* within *archive*. If no *modifier* is given, move each *member* to the end. The ranlib header is modified if present.

p Print each *member*. This is useful only with archives of text files.

q Quick append: append each *member* to the end of *archive* unconditionally. The ranlib header is not updated.

r Replace each *member* of *archive*. If *archive* does not exist, create it. The optional *modifier* specifies how to perform the replacement, as described below. The ranlib header is modified if present.

t Print a table of contents that lists each *member* specified. If none is given, list all in *archive*. The modifier **v** tells **ar** to give you additional information.

x Extract each given *member* and place it into the current directory. If none is specified, extract all members. *archive* is not changed.

The *modifier* may be one of the following. The modifiers **a**, **b**, **i**, and **u** may be used only with the **m** and **r** options.

a If *member* does not exist in *archive*, insert it after the member named by the given *position*.

b If *member* does not exist in *archive*, insert it before the member named by the given *position*.

c Suppress the message normally printed when **ar** creates an archive.

i If *member* does not exist in *archive*, insert it before the member named by the given *position*. This is the same as the **b** modifier, described above.

k Preserve the modify time of a file. This modifier is useful only with the **r**, **q**, and **x** options.

s Modify an archive's ranlib header, or create it if it does not exist. This must be used for archives read by the linker **ld**.

u Update *archive* only if *member* is newer than the version in the *archive*.

v Generate verbose messages.

Note that because **ar** was created before UNIX established the standard of introducing an option with a hyphen. Therefore, the syntax of options to **ar** differs from most other COHERENT commands: **ar** expects all options and modifiers to be clumped together as its first argument, without an introductory hyphen. For example, to use the option **r** with the modifiers **c** and **s** on library **libname.a** and objects **file1.o** through **file3.o**, type the following command:

```
# RIGHT!
ar rcs libname.a file1.o file2.o file3.o
```

The syntax

```
# WRONG!
ar r -s libname.a file1.o file2.o file3.o
```

creates an archive named **-s**, which you may have some trouble removing.

ar reads the environmental variables **ARHEAD** and **ARTAIL** and appends them to, respectively, the beginning and end of its command line. For example, to ensure that **ar** is always executed with the **c** modifier, insert the following into your **.profile**:

```
export ARHEAD=c
```

Library Structure

All archives are written into a specialized file format. Each archive starts with a "magic string" called **ARMAG**, which identifies the file as an archive. The members of the archive follow the magic number; each is preceded by an **ar_hdr** structure. For information on this structure, see **ar.h**. The structure is followed the data of the file, which occupy the number of bytes specified by the variable **ar_size**.

See Also

ar.h, commands, ld, nm, ranlib

Notes

Each library that you create should have a name that begins with "lib" and ends with ".a". The prefix "lib" lets you call that library with the **-l** option to the command **cc**; and the linker **ld** ignores archives whose names do not end in **.a**.

ar.h — Header File

Format for archive files
#include <ar.h>

An *archive* is a file that has been built from a number of files. Archives are maintained by the command **ar**. Usually, an archive is a library of object files used by the linker **ld**.

The header **ar.h** describes the format of an archive. All archives start with a magic number **ARMAG**, which identifies the file as an archive. The members of the archive follow the magic number, each preceded by the structure **ar_hdr**:

```
#define DIRSIZ  14
#define ARMAG   0177535          /* magic number */

struct ar_hdr {
        char    ar_name[DIRSIZ]; /* member name */
        time_t  ar_date;         /* time inserted */
        short   ar_gid;          /* group owner */
        short   ar_uid;          /* user owner */
        short   ar_mode;         /* file mode */
        size_t  ar_size;         /* file size */
};
```

The structure at the head of each member is immediately followed by **ar_size** bytes, which are the data of the file.

To enhance the performance of **ld**, the command **ranlib** provides a random library facility. **ranlib** produces archives that contain a special entry named **__.SYMDEF** at the beginning.

All integer members of the structure (everything but **ar_name**) are in canonical form to ease portability. See **canon.h** for more information.

See Also

ar, canon.h, header files, ld, ranlib

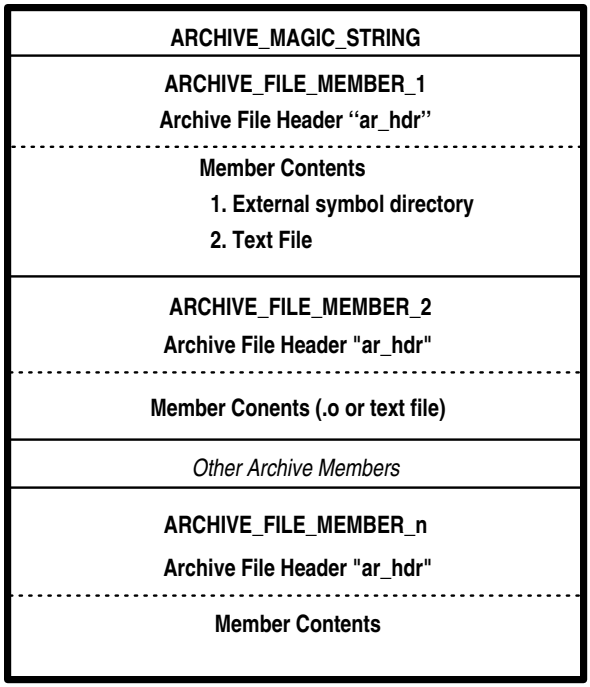
arcoff.h — Header File

COFF archive-file header
#include <coff/arcoff.h>

arcoff.h declares the structure **ar_hdr**, which is the header to a member of an archive. **ar_hdr** is structured as follows:

```
struct ar_hdr
{
        char    ar_name[16];                /* file member name - '/' terminated */
        char    ar_date[12];                /* file member date - decimal */
        char    ar_uid[6];                  /* file member user id - decimal */
        char    ar_gid[6];                  /* file member group id - decimal */
        char    ar_mode[8];                 /* file member mode - octal */
        char    ar_size[10];                /* file member size - decimal */
        char    ar_fmag[2];                 /* ARFMAG - string to end header */
};
```

The COFF common-archive format has the following structure:



See Also

a_out.h, file formats, header files
Gircyc, G.R.: *Understanding and Using COFF*. Sebastopol, Calif, O'Reilly & Associates, Inc., 1990.

arena — Definition

An *arena* is the area of memory that is available for a program to allocate dynamically at run time. It is divided into *allocated* and *unallocated* blocks. The unallocated blocks together form the "free-memory pool".

To allocate a portion of the arena, use any of the functions **malloc()**, **calloc()**, or **realloc()**. To return an allocated portion of memory to the free-memory pool, use the function **free()**. To check whether a given portion of the arena is already allocated, use the function **notmem()**. To check whether the arena has been corrupted, use the function **memok()**.

See Also

calloc(), free(), malloc(), memok(), notmem(), Programming COHERENT, realloc()

argc — C Language

Argument passed to main()
int argc;

argc is an abbreviation for "argument count". It is the traditional name for the first argument to a C program's **main** routine. By convention, it holds the number of arguments that are passed to **main** in the argument vector **argv**. Because **argv[0]** is always the name of the command, the value of *argc* is always one greater than the number of command-line arguments that the user enters.

Example

For an example of how to use **argc**, see the entry for **argv**.

See Also

argv, C language, envp, main()
ANSI Standard, §5.1.2.2.1

argv — C Language

Argument passed to main()
char *argv[];

argv is an abbreviation for "argument vector". It is the traditional name for a pointer to an array of string pointers passed to a C program's **main** function; by convention, it is the second argument passed to **main**. By convention, **argv[0]** always points to the name of the command itself.

Example

This example demonstrates both **argc** and **argv[]**, to recreate the command **echo**.

```
main(argc, argv)
int argc; char *argv[];
{
      int i;

      for (i = 1; i < argc; ) {
            printf("%s", argv[i]);
            if (++i < argc)
                  putchar(' ');
      }

      putchar('\n');
      exit(0);
}
```

See Also

argc, C language, envp, main()
ANSI Standard, §5.1.2.2.1

ARHEAD — Environmental Variable

Append options to beginning of ar command line
export ARHEAD=*options*

The COHERENT archiver **ar** reads the environmental variables **ARHEAD** and **ARTAIL** before it begins its work. You can set these variables to hold the default options that you want the archiver always to use. **ar** appends the options in **ARHEAD** to the beginning of its command line.

See Also

ar, ARTAIL, environmental variables

Notes

This environmental variable is included only to support existing code. Its use is deprecated, and it may not be supported in future releases of COHERENT.

array — Definition

An **array** is a concatenation of data elements, all of which are of the same type. All the elements of an array are stored consecutively in memory, and each element within the array can be addressed by the array name plus a subscript.

For example, the array **int foo[3]** has three elements, each of which is an **int**. The three **int**s are stored consecutively in memory, and each can be addressed by the array name **foo** plus a subscript that indicates its place within the array, as follows: **foo[0]**, **foo[1]**, and **foo[2]**. Note that the numbering of elements within an array always begins with '0'.

Arrays, like other data elements, may be automatic (**auto**), **static**, or external (**extern**).

Arrays can be multi-dimensional; that is to say, each element in an array can itself be an array. To declare a multi-dimensional array, use more than one set of square brackets. For example, the multi-dimensional array **foo[3][10]** is a two-dimensional array that has three elements, each of which is an array of ten elements. The second sub-script is always necessary in a multi-dimensional array, whereas the first is not. For example, the form **foo[][10]** is acceptable, whereas **foo[10][]** is not. The first form is an indefinite number of ten-element arrays, which is correct C, whereas the second form is ten copies of an indefinite number of elements, which is illegal.

You can initialize automatic arrays and structures, provided that you know the size of the array, or of any array

contained within a structure. An automatic array is initialized in the same manner as aggregate, but initialization is performed on entry to the routine at run time, instead of at compile time.

Flexible Arrays

A **flexible array** is one whose length is not declared explicitly. Each has exactly one empty '[ ]' array-bound declaration. If the array is multidimensional, the flexible dimension of the array must be the *first* array bound in the declaration; for example:

```
int example1[][20];     /* RIGHT */
int example2[20][];     /* WRONG */
```

The C language allows you to declare an indefinite number of array elements of a set length, but not a set number of array elements of an indefinite length.

Flexible arrays occur in only a few contexts; for example, as parameters:

```
char *argv[];
char p[][8];
```

as **extern** declarations:

```
extern int end[];
```

or as a member of a structure — usually, though not necessarily, the last:

```
struct nlist {
      struct nlist *next;
      char name[];
};
```

Example

The following program initializes an automatic array, and prints its contents.

```
main()
{
      int foo[3] = { 1, 2, 3 };

      printf("Here's foo's contents: %d %d %d\n",
             foo[0], foo[1], foo[2]);
}
```

See Also

initialization, Programming COHERENT, struct

ARTAIL — Environmental Variable

Append options to end of ar command line
export ARTAIL=*options*

The COHERENT archiver **ar** reads the environmental variables **ARHEAD** and **ARTAIL** before it begins its work. You can set these variables to hold the default options that you want the archiver always to use.

ar appends the options in **ARTAIL** to the end of its command line.

See Also

ar, ARHEAD, environmental variables,

Notes

This environmental variable is included only to support existing code. Its use is deprecated, and it may not be supported in future releases of COHERENT.

as — Command

i80386 assembler
as [-o *outfile***] [-bfglnpwxX]** *infile*

The 80386 version of **as**, the COHERENT assembler, assembles programs written in any of several different dialects of assembly language into object modules in COFF format, which can be linked with objects written by the COHERENT C compiler. This version of **as** contains numerous features not available with the COHERENT 286 assembler:

- It serves as a flexible base for writing programs in native 80386 assembly language.
- It assembles programs written in older flavors of COHERENT assembly language.
- It assembles programs written in UNIX assembly language.
- Unlike the old COHERENT assembler and the UNIX assembler, 80386 **as** comes with full macro faculities.
- It is also designed to detect many of the common errors made by assembly-language programmers.

The COHERENT system also includes the command **asfix**, which updates files written in the COHERENT 286 assembler. **asfix** changes local and character symbols to the new format.

Invoking the Assembler

as permits file names and options to be interspersed upon the command line. It recognizes the following command-line options:

-D*name***=***string*
Initialize string variable *name* to *string*. For example, the option

```
-Dname=some_string
```

is equivalent to:

```
name  .define     some_string
```

-E*name***=***value*
Initialize variable *name* to *value*. For example, the option

```
-Ename=17
```

is equivalent to:

```
name  .equ  17
```

-a Set alignment for data objects. For example, when this option is used the express

```
.long 5
```

is automatically aligned to a four-byte boundary, but is left unaligned without it.

-b Reverse bracket sense; that is, use () for expressions and [] for code. For example:

```
movl   $[2 * 5], (%eax)      / without -b
movl   $(2 * 5), [%eax]      / with -b
```

-f Reverse the order of the operands, from UNIX-assembler form to that of the Intel documentation or the 80286 version of **as**.

-g Make undefined symbols **.globl**.

-l Generate an output listing.

-n This option turns off the **as** mechanism for handling bugs in the 80386 chip. **as** tries to cope with known 80386 bugs by changing code at appropriate points in its output. If these changes create problems with your code, you can turn off the **as** bug-handler mechanism by using the **-n** option to **as**.

-o *outfile***.o**
Write the output into *outfile***.o**. Note that the suffix **.o** must appear in the output file's name, or the assembler will exit with an error message. The default output file is *infile***.o**.

-p Don't use '%' on register names; e.g., use **ax**, not **%ax**.

-Q Quiet: Suppress all error messages, no matter how awful an error they indicate.

-w Disable warning messages.

-x Remove all non-global symbols from the common symbol output.

-X Remove all non-global symbols starting with **.L** from the common symbol output.

as reads the environmental variables **ASHEAD** and **ASTAIL** and appends them to, respectively, the beginning and the end of its command line. By setting these variables, you can ensure that **as** always executes with the switches

that you want. For example, to ensure that **as** always executes with the **-g** switch set, insert the following into your **.profile**:

```
export ASHEAD=-g
```

Lexography

A symbol consists of from one to 256 characters. The assembler defines a *character* as being an alphabetic character, question mark, period, percent sign, or underscore. **Xyz**, **.20**, and **hi_there** are legal symbols; whereas **85i** is not.

Like C, the **as** assembly language is case sensitive.

Local symbols begin with a question mark. These are recognizable (or *visible*) only between nonlocal symbols. For example:

```
/       ?loop invisible here
abc     mov     $10, %cx
?loop   add     $1, %bx                 / ?loop visible here
        jcxz    xyz
        jmp     ?loop
xyz:
/       ?loop invisible here
```

An octal number is defined just as in the C language: it consists of an initial **0** plus two other numerals between 0 and 7. For example, **077** is a legal octal number.

A hexadecimal number consists of an initial **0x** or **0X** plus two other numerals: 0 through 9, a through f, or A through F. For example, **0x0F** and **0Xa3** are legal hexadecimal numbers.

A binary number consists of an intial **0b** or **0B** followed by an indefinite number 0's and 1's. For example, **0b01001010** is a legal binary number.

A decimal number begins with a numeral other than 0, followed by an indefinite number of numerals between 0 and 9. For example, **109** is a legal decimal number.

A floating-point number begins is a string of numerals, 0 through 9, with a period or **e** within or at the end of it. It is like a C floating-point number, except that it cannot begin with a period because a symbol may begin with a period. For example, **123.456**, **123456.**, and **17e26** are legal floating-point numbers, but **.123456** is not.

A character constant is enclosed between apostrophes, as in C. **as** recognizes the same escape sequences as C. See the Lexicon article **C language** for a table of these constants.

String constants are enclosed between quotation marks, as in the C language, and use the same escape sequences as C. See the Lexicon article **C language** for a table of these sequences.

Pseudo-Opcodes

as recognizes a rich set of pseudo-opcodes. These are not true assembly-language opcodes, but are interpreted by the assembler; they are designed to help make your life easier. The following briefly summarizes the pseudo-opcodes.

.16 . . . . . 16-bit mode
.2byte . . . . . Make unaligned short variables
.32 . . . . . 32-bit mode
.4byte . . . . . Make unaligned long variables
.align . . . . . Increment location counter to two- or four-byte aligned spot
.alignoff . . . . . Turn alignment off
.alignon . . . . . Turn alignment on
.blkb . . . . . Set tag in **.data**
.bracketnorm . . . . . Normal bracket sense — see **-b** option
.brcketrev . . . . . Reverse bracket sense — see **-b** option
.bss . . . . . Set tag in **.bss**
.bssd . . . . . Set tag in **.bss**
.byte . . . . . Make byte variables
.comm . . . . . Set label as common
.data . . . . . Change segment to **.data**
.def . . . . . Reserved to set auxiliary symbol entries in a later release
.define . . . . . Define string constant

.dim . . . . . Reserved to set auxiliary symbol entries in a later release
.double . . . . . Make double variables
.eject . . . . . Force a page break
.else . . . . . Connected to **.if**
.endef . . . . . Reserved to set auxiliary symbol entries in a later release
.endi . . . . . End **.if**
.endm . . . . . End **.macro** definition
.endw . . . . . End **.while**
.equ . . . . . Define numeric constant
.erratoff . . . . . Turn off chip errata fixes
.erratoon . . . . . Turn on chip errata fixes
.even . . . . . Increment location counter to byte-aligned spot
.fail . . . . . Print error message
.file . . . . . Reserved to set auxiliary symbol entries in a later release
.float . . . . . Make **float** variables
.globl . . . . . Declare names as visible to linker
.ident . . . . . **.ident** string
.if . . . . . Compile-time conditional
.include . . . . . Include a file
.intelorder . . . . . Intel operand order — see **-f** option
.lcomm . . . . . Set name up as common
.line . . . . . Reserved to set auxiliary symbol entries in a later release
.list . . . . . Turn on listing (assumes **-l** option)
.llen . . . . . Set print line length
.ln . . . . . Reserved to set auxiliary symbol entries in a later release
.long . . . . . Make long variables
.macro . . . . . Define a macro name
.mexit . . . . . Exit current macro expansion
.mlist . . . . . Toggle listing of macro expansion
.nolist . . . . . Turn off listing (assumes **-l** option)
.nopage . . . . . Turn off page breaks and titles
.number . . . . . Convert a string to a number.
.org . . . . . Change location counter
.page . . . . . Turn on page breaks and titles
.plen . . . . . Set page length
.prvd . . . . . Change segment to **.data**
.prvi . . . . . Change segment to **.text**
.scl . . . . . Reserved to set auxiliary symbol entries in a later release
.set . . . . . Makes name equal to expr
.shift . . . . . Shift macro parameters
.shrd . . . . . Change segment to **.data**
.shri . . . . . Change segment to **.text**
.size . . . . . Reserved to set auxiliary symbol entries in a later release
.string . . . . . Convert a floating-point expression to a string
.strn . . . . . Change segment to **.data**
.tag . . . . . Reserved to set auxiliary symbol entries in a later release
.text . . . . . Change segment to **.text**
.ttl . . . . . Set page titles
.type . . . . . Reserved to set auxiliary symbol entries in a later release
.undef . . . . . Free string, numeric constant, or opcode
.unixorder . . . . . Return normal order of operands; undoes **.intelorder**
.val . . . . . Reserved to set auxiliary symbol entries in a later release
.value . . . . . Make short variables
.version . . . . . Comment string
.warn . . . . . Print a warning message
.warnoff . . . . . Turn off warning messages
.warnon . . . . . Turn on warning messages
.while . . . . . Compile-time loop control
.word . . . . . Make short variables
.zero . . . . . Create zero-filled memory

Each pseudo-opcode is described in the following sections.

Input Format

An assembly-language program consists of a series of lines with the following format:

```
[#][label]  [opcode]    [operands]  [/ comment]
```

The optional '#' at the beginning of the line tells **as** not to replace any **.define** symbols within the line. (These are described below.) Normally, the assembler replaces all **.define** symbols in a line before it parses that line. Without this option, a series of **.define**s could lead to awkward results.

For example, the code

```
#%ecx .define     xx
#xx   .define     (%ecx)
      mov   $3, %ecx
```

results in:

```
      mov   $3, (%ecx)
```

Like the C compiler, **as** will not go into an infinite loop if two **.define** statements mirror each other.

A comment begins with a slash '/' and may include the entire line. Blank lines are also legal.

Extra operands are not assumed to be comments. This is to tighten up error checking for the convenience of new and part-time assembly-language programmers.

Expression Format

The **as** macro assembler has mostly the same operators and precedence as the C preprocessor. The exceptions are **?:**, **&&**, **||**, **:**, and ',' (which are missing), '/' (which is spelled **.div**), and '%' (which is spelled **.rem**).

In addition, the macro assembler includes the following directives: **.defined**, **.sizeof**, **.segment**, **.parmct**, **.location**, **.string**, **.number**, and **.float**.

Expression bracketing is normally done by **[]**, because **()** is used by the operand format. This may be reversed by the **-b** option, described above.

The unary operators have the following priority:

.float .number .string	Conversion
.defined .sizeof	
.location .segment	Inquiry
-	Negation
!	Logical negation

The binary operators have the following priority:

[]	
* **.div .rem**	Multiply, divide, remainder
+ -	Add, subtract.
>> <<	Left shift , right shift
< > <= >= == !=	Comparison
&	AND
^	Exclusive OR
|	OR
#	Repeat

Most binary operators should be familiar to C programmers; the exception is the **#**, which repeats an instruction *N* times. For example, the expression

```
      .byte 5 # 3
```

produces five copies of byte 3, whereas the expression

```
      .long 7 # 4
```

produces seven copies of the long '4'. Note that this operator has the lowest precedence of all binary operators.

You can use an expression wherever you can use a number. This includes address displacements, constants, and

.if and **.while** statements. Integers are internally 32 bits, floats are internally C doubles.

Like C, comparison operators return one for true and zero for false.

In addition, **as** provides string operators. Like C, the first element of a string is indexed as zero. Unlike C, however, attempts to access past the end of a string gives all zeroes. The following summarizes the **as** suite of string operators:

string **+** *string*
: Concatenate two strings. For example, **"12" + "34"** yields **"1234"**.

string **[** *expr1*, *expr2* **]**
: Address a substring from *expr1* to *expr2*. For example, **"1234567"[1,3]** yields **"234"**; and **"123"[1,10]** yields **"23"**.

string **[** *expr* **]**
: Address a substring from *expr* to the end of the string. For example **"1234567"[5]** yields **"67"**.

.string *expr*
: Convert a numeric expression to a string. For example, **.string 123** gives **"123"**.

.string *float*
: Convert a floating-point expression to a string. For example, **.string 0.5 * 3** gives **"1.5"**

.float *string*
: Convert a string to a floating-point number.

.float *expr*
: Convert a numeric expression to a floating-point number.

.number *string*
: Convert a string to a number.

.number *float*
: Convert a floating-point number to a number.

.string (*expr* **)**
: Return character at position *expr* as a number. For example, **"123"(1)** gives two.

string1 **@** *string2*
: Return the position at which *string2* begins within *string1*. For example, **"12345" @ "23"** returns one; and **"123" @ "jj"** gives -1 (because "jj" does not appear within "123").

The unary operator **:** creates a label equal to the current location. It is generally not needed. For example, the expression

```
connected   .long 5
```

builds an aligned **long**, initializes it to five, and gives it the label **connected**. However, the expression

```
unconnected:      .long 5
```

builds the label **unconnected** at the current location, then builds an aligned **long** with a value of five. Note that the label **connected** will be on the five, whereas the label **unconnected** may be somewhere else if there was alignment. For example, the expression

```
      .align      4
lab1: lab2: lab3: .long 5
```

puts **lab1**, **lab2**, and **lab3** on the **long** because it is already aligned.

Macros and Conditional Compilation

The **as** directive **.macro** lets you declare a macro that you can use through a program. The directive **.endm** marks the end of a macro declaration.

A macro has the following form:

name **.macro** *params*
body of macro
.endm

LEXICON

The following example creates and uses the macro **store**:

```
store   .macro  xy,xz                   / declare "store" with two parms: xy and xz
        movl    xy,%ecx
        movl    %ecx,(%eax)
        movl    xz,%ecx
        movl    %ecx,4(%eax)
        .endm                           / end of macro

        store   5,10                    / moves 5 and 10 to where %eax points.
```

Macros can contain **.if** statements, and can even define other macros. For example:

```
def     .macro  .name, to               / macro for defining other macros
name    .macro
        movl    from, to
        .endm
        .endm

        def     frog, %eax, %ebx        / define the macro frog
        frog                            / movel%eax, %ebx
```

as increments a count every time you expand any macro, and associates that number with the macro. When the keyword **.macno** is used within a macro, **as** translates it into that number. Thus, **.macno** is a unique number within each macro expansion. This allows the generation of unique labels internal to macros. For example:

```
stradd  .macro  str
        .data
L\.macno        .byte str, 0            / create a data item
        .text
        movl    L\.macno, %eax          / put its address into %eax
        .endm
```

L\.macno becomes something like L51. Note that a '\' before any defined symbol or macro name vanishes in the expansion pass.

To permit macros with indefinite parameter counts, the assembler offers the reserved word **.parmct** and the command **.shift**. The former holds the number of parameters passed to a macro, and the latter shifts the parameters one position to the left. For example:

```
kall    .macro  fun, parm
        .while .parmct > 1              / while more than one parm remains
        push parm
        .shift                          / parm 3 becomes parm 2, parm 4 parm 3 etc
        call fun
        .endm
```

The operators **.if**, **.else**, and **.endi** allow a program to implement compile-time decisions. These may be inside or outside of macros. When a macro exits, the assembler automatically closes all **.if** statments that had been started within it. For example:

```
defy    .macro
        .if .defined y                  / if y has been defined true
        .mexit                          / exits closing any if statements
        .else
y       .equ 1  / define y as 1
/ For UNIX compatibility
/       .set    y, 1
/ produces the same result
        .endm
```

When used with a label, the operator **.defined** is true if that label had been defined in this pass. If the label is defined later, **.defined** can still be used with it, but causes a phase error, as occurs in some assemblers.

The operator **.fail** permits the flagging of errors. For example:

```
      .if ! .defined y
      .fail y is not defined
      .endi
```

The operator **.include** permits the inclusion of files. For example:

```
      .include    somefile.h
```

Undefining Symbols or Opcodes

as Some software (e.g., the GNU C compiler) requires that opcodes be recognized on column one and that opcodes be replacable by macros. The command **.undef** un-defines all macros and opcodes. Once you have un-defined an identifier, you can re-use it to name a macro or other data item. For example, to use **mov** (which names an opcode) to name a macro, do the following:

```
      .undef      mov
mov   .macro      foo, bar
      movl  foo, bar
      .endm
```

Data-Definition Operators

The following describes the data-definition operators that **as** supports.

.byte *expr*
Define *expr* as an array of single bytes. *expr* can take any number of forms, as shown by the following examples:

```
       .byte   5, 2                  / defines 2 bytes 0x05 and 0x02
       .byte   "Hello World", 0      / a zero-terminated Hello World
       .byte   10 # 1                / 10 repetitions of 0x01
```

.word *expr*
Define *expr* as a word, that is, as a two-byte integer. For example:

```
       .word   .sizeof xx            / a short the size of xx
       .word   50 * 50               / a short of 100
/ For UNIX compatability
/      .value  50 * 10
/ produces the same result.
```

.long *expr*
Define *expr* as a long (four-byte) integer. For example:

```
       .long   10                    / a long of 10
```

.comm *name*, *length*
Define a common variable named *name*, that is *length* bytes long. (See the entry for **.lcomm**, below, for a discussion of what segment the variable is stored.) If *name* is linked with another module that also declares *name* but sets it to another length, the linker creates one such variable and gives it the greater length of the two.

The linker deduces the alignment of a common variable from its length: if the length of a common is divisible by four, it is aligned on a four-byte boundary; if it is divisible by two, it is aligned on a two-byte boundary. Otherwise, it is assumed to be unaligned. The linker supports only three classes of alignment: four-byte, two-byte, and unaligned.

A common variable is aligned according to its most strongly aligned contributor. For example, if one module contributes a common variable named **xyz** whose length is four bytes, and another contributes an **xyz** whose length is five bytes, the resulting **xyz** is given a length of eight bytes to satisfy the length requirement (at least five) and the alignment requirement (four-byte boundary).

After the first linker pass, all common variables are placed at the end of the **.bss** segment: first the four-byte-aligned variables, then the two-byte-aligned, then the unaligned.

By default, **as** does not align its data objects. The command-line option **-a** instructs **as** to align all data objects automatically.

.lcomm *label*, *length*
Same as **comm**, described above.

Please note that on a COFF-based system, it is not possible to put common data into the **.data** section, even though the UNIX assembler documentation claims that **.comm** does this. Both **.comm** and **.lcomm** place data into the **.bss**.

LEXICON

The problem is that COFF format for common variables leaves no place for information about alignment or segment. This creates two problems. First, the lack of information about alignment forces COFF to adopt the complex strategy of deducing alignment from length. Second, the lack of information about segment compels COFF to store all common variables in one segment, **.bss** being chosen.

.float *expr*
Define *expr* as a single-precision floating-point number. For example:

```
.float  1.5                    / a float of 1.5
```

.double *expr*
Define *expr* as a double-precision floating-point number. For example:

```
.double 3.0 * 0.5              / a double of 1.5
```

Resetting the Location Counter

The instructions **.org** and **.align** reset the location counter. For example:

```
.org    .+5                    / Location counter to here plus 5
.org                           / Location counter to top of current section
.align  2                      / Up to nearest two-byte boundary
```

The pseudo-opcodes **.alignon** and **.alignoff** respectively turn aligning on and off.

As noted above, the command-line option **-a** instructs **as** to align all data objects automatically.

The instructions **.text**, **.data**, and **.bss** reset the location counter to the corresponding sections. Instructions are placed in the **.text** section, initialized data in the **.data** section, and the **.bss** is reserved for unitialized data. Placing information into the **.bss** results in an error.

Dynamic Linking

The Intel Binary Compatibility Standard dictates the way that **as** computes addresses, to permit dynamic linking of objects.

In object files, all **.data** addresses must follow all **.text** addresses, and all **.bss** address must follow all **.data** addresses. This allows dynamic linking of object files, in which the object file is mapped, not read in pieces.

In the **as** assembly language, **.data** and **.text** addresses are started from 0 for each module. At the end of assembly, during the output phase, **as** fixes these addresses to make **.data** follow **.text**, and so on.

For example, if you have a conditional like

```
.if   some_data_address > 0x300
```

as calculates the address for the **.if** statement from the beginning of its segment; and the address is only corrected in the final output. Such statements may appear to be working incorrectly.

Listing Commands

as prints a listing if you use its **-l** option. The following commands modify the form of this listing.

.ttl *string*
Print *string* as the title to the command page. For example:

```
.ttl  This is a page title
```

If you do not use this command, the assembler uses the file name for the title. The first **.ttl** encountered in the assembly pass 0 is used to set the first title. Subsequent **.ttl** commands reset the title before printing.

.nopage
Turn off page breaks and titles.

.page Turn on page breaks and titles.

.eject Force a page break.

.nolist Turn off the listing.

.list Turn the listing back on.

.mlist off
Turn off the listing of macro expansions.

.mlist on
Turn on the listing of macro expansions.

Addressing Modes

as recognizes two modes of addressing: *16-bit mode* and *32-bit mode*. In 16-bit mode, the address type and operand mode default to 16 bits; in 32-bit mode they default to 32 bits. For example:

```
.16
movw   %ax, (%si)     # Is generated without escapes.
movl   %eax, (%esi)   # Has two escapes, address and operand
.32
movw   %ax, (%si)     # Has two escapes, address and operand
movl   %eax, (%esi)   # Is generated without escapes.
```

In 16-bit mode, the 16-bit addressing forms in table 17-2 of the *Intel 386 Programmer's Manual* are generated where they fit; otherwise, an address escape is built and the 32-bit forms in tables 17-3 and 17-4 are used. In 32 bit mode, this is reversed.

as uses the following grammar in its addressing modes:

Eight-bit registers

```
r8  : %al | %cl | %dl | %bl | %ah | %ch | %dh | %bh;
```

16-bit registers

```
r16 : %ax | %cx | %dx | %bx | %sp | %bp | %si | %di;
```

32-bit registers

```
r32 : %eax | %ecx | %edx | %ebx | %esp | %ebp | %esi | %edi;
```

Segment registers

```
sreg : %es | %cs | %ss | %ds | %fs | %gs;
```

Control registers

```
ctlreg : %cr0 | %cr2 | %cr3;
```

Debug registers

```
dbreg : %dr0 | %dr1 | %dr2 | %dr3 | %dr6 | %dr7;
```

Test registers

```
testreg : %tr6 | %tr7;
```

m16 These addresses can have a segment prefix:

```
m16 : m16b | sreg ':' m16b;
```

m32 These addresses can have a segment prefix:

```
m32 : m32b | sreg ':' m32b;
```

rm16 These addresses can have a segment prefix or may be **r16**:

```
rm16 : rm16b | sreg ':' rm16b;
```

rm32 These addresses can have a segment prefix or may be **r32**:

```
rm32 : r32 | rm32b | sreg ':' rm32b;
```

rm8 These addresses can be **rm32**, **rm16**, or **r8**:

```
rm8  : r8 | rm16b | sreg ':' rm16b | rm32b | sreg ':' rm32b;
```

rm16b

```
displacement | (vx, vy) | displacement(vx, vy) |
      displacement(vw) | (vz);
vx : %bx | %si;
vy : %si | %di;
vz : %si | %di | %bx;
```

rm32b

```
(va) | displacement(vb) | (, vb, scale) | (vb, scale)
     | displacement(vb, scale) | (vb, vb, scale)
     | displacement(vb, vb, scale);
va : %eax | %ecx | %edx | %ebx | %esi | %edi;
vb : %eax | %ecx | %edx | %ebx | %ebp | %esi | %edi;
vb : %eax | %ecx | %edx | %ebx | %ebp | %esp | %esi | %edi;
scale : 0 | 1 | 2 | 4 | 8;
```

mem32
A 32-bit memory address.

mem16
A 16-bit memory address.

reli Expand to eight-, 16-, or 32-bit relative addresses.

rel8 Eight-bit relative addresses.

rel16 Sixteen- or 32-bit relative addresses.

Using as To Create Debug Information

Some UNIX languages, such as **gcc** and **gc++**, produce assembly language rather than object code. The following documents how to use **as** with such compilers. Note that error checking is minimal, and that bad debug information can corrupt the generated COFF output. This section must be read with a listing of the header file **coff.h** for reference; or see the Lexicon article **coff.h**.

The compiler starts with type and line information in a format much like that of the desired COFF output files. It must break this down into lines to ship through the assembler, and the assembler then must rebuild the information into COFF format for output.

.file *filename*
This connects the object file to the original source file. If used, this should be the first statement in the file. It produces a **SYMENT** of *n_sclass* = *C_FILE* and an **AUXENT** with *ae_fname* = *filename*.

.def *symbolName*
This instruction initializes **SYMENT** with *n_name* = *symbolName*. If there is a symbol by that name on the assembler's internal symbol table, it is marked to prevent output to the symbol table. Any **RELOC** references point to this table entry, so its *n_value* must be correct. Because we assume that code of this kind is result of a compiler, we assume it is correct. The following commands up to and including **.endef** refer to this **SYMENT**.

.type *number*
This sets this **SYMENT**'s *n_type number*. If *number* indicates a function, **DT_FCN**, a **LINENO** record is built pointing at this **SYMENT**.

.val [*symbol***] [***number***]**
This sets this **SYMENT**'s *n_value*. If it is a *symbol*, it sets *n_scnum* to the symbol's section number.

.scl *number*
This sets this **SYMENT**'s *n_sclass* to *number*.

.dim *number* **[,** *number* **[,** *number* **[,** *number***]]]**
This sets up to four entries in an **AUXENT**'s *ae_dimen*. It describes multidimensioned arrays to COFF. This command supports only four dimensions because the COFF specifications are reliable only though four dimensions.

.size *n* This sets this **AUXENT** *ae_size* to *n*.

.tag *name*
This scans backwards on the **SYMENT**s for a matching *n_name*. It points this *ae_tagndx* to that name's symbol number and that *ae_endndx* to the next symbol *number*.

A good example is a **struct**: It would start with a **SYMENT** of type **T_STRUCT**, then then have **SYMENT**s for its members. At the end, there would be a **C_EOS** (end of structure) with a tag that gets us back to the symbol's name. **.tag** connects the forward and backward pointers.

.line *n* This sets the **AUXENT**'s *ae_lnno* to *n*.

.endef This marks the end of a **SYMENT** started by **.def**. If the *n_sclass* **== C_EFCN** (end of function), it builds the functions *ae_fsize* and *ae_endndx* and does not output this **SYMENT**. If any **AUXENT** fields were set, an **AUXENT** record follows this **SYMENT**.

.ln *number*
This builds a **LINENO** record with **l_lnno =** *n* and **l_addr.l_paddr =** the current location.

Instructions

In matching instructions, **as** first looks up the name of the instruction. A number of actual instructions will match that name. For example, **btsw** matches 0xab and 0x0fab /5, and **bts** matches anything that matches **btsw** and **btsl**.

as attempts to match operands to the instruction until a form is found that will accept all the operands. If no form matches all the operands, **as** prints the error message

```
Illegal combination of opcode and operands
```

The assembler at that point cannot say which operand is wrong because of the nature of the 80386 instruction set.

If you see a great number of these messages, **as**'s command-line option **-f** may be in the wrong sense: although the opcode is valid and the operands are valid, there is no form of this opcode that takes these operands in this order.

as first attempts to match opcodes that do not require an operand-mode escape: that is, in 80386 mode it attempts to match long-mode instructions first, then short-mode instructions.

Register Usage

The COHERENT C compiler uses the following save/restore sequence for a function, to set the frame pointer when the function contains no automatic variables:

```
push  %ebp
movl  %ebp, %esp
```

If *n* bytes of autos are required, then it uses the following sequence:

```
enter $n, $0
```

It then executes the code

```
push  %esi
push  %edi
push  %ebx
```

to preserve register variables *as required:* they are not saved/restored if the function does not touch them. (This is why they are saved after the frame adjust, not before). To restore register variables, it executes

```
pop   %ebx
pop   %edi
pop   %esi
```

as required, followed by

```
leave
ret
```

Routines written in assembly language must preserve registers **ebp**, **esi**, **edi**, and **ebx**; they may overwrite **eax**, **ecx**, and **edx**.

Absolute Symbols

as can create what COFF calls "absolute symbols." For example

LEXICON

```
      .globl      x
x     .equ  10
x     .equ  x * x / The last value of x in the module
```

leaves on the symbol table an absolute symbol for **x** of 100. For internal reason, the **.globl** must preceed any **.equ**.

Opcodes

The following gives a table of the opcodes recognized by **as**. Note that the opcode is sometimes followed by a slash and a number, or a letter. For example,

```
      D0 /4 salb  con1, rm8
```

means opcode is 0xD0 place 4 in the register/opcode field of the **modr/m** byte.

```
      58 +r popl  r32
```

means add the register number to 0x58.

Opcode	*Instruction*	*Operands*	*Description*
37	**aaa**		Adjust after addition
D5 0A	**aad**		Adjust AX before division
D4 0A	**aam**		Adjust AX after multiply
3F	**aas**		Adjust AL after subtraction
	adc		Add with carry
83 /2	**adcl**	*imm8s,rm32*	
83 /2	**adcw**	*imm8s,rm16*	
14	**adcb**	*imm8,al*	
15	**adcw**	*imm16,ax*	
15	**adcl**	*imm32,eax*	
15	**adcl**	*imm32*	
80 /2	**adcb**	*imm8,rm8*	
81 /2	**adcw**	*imm16,rm16*	
81 /2	**adcl**	*imm32,rm32*	
12 /r	**adcb**	*rm8,r8*	
13 /r	**adcw**	*rm16,r16*	
13 /r	**adcl**	*rm32,r32*	
10 /r	**adcb**	*r8,rm8*	
11 /r	**adcw**	*r16,rm16*	
11 /r	**adcl**	*r32,rm32*	
	add		Add
83 /0	**addl**	*imm8s,rm32*	
83 /0	**addw**	*imm8s,rm16*	
04	**addb**	*imm8,al*	
05	**addw**	*imm16,ax*	
05	**addl**	*imm32,eax*	
05	**addl**	*imm32*	
80 /0	**addb**	*imm8,rm8*	
81 /0	**addw**	*imm16,rm16*	
81 /0	**addl**	*imm32,rm32*	
02 /r	**addb**	*rm8,r8*	
03 /r	**addw**	*rm16,r16*	
03 /r	**addl**	*rm32,r32*	
00 /r	**addb**	*r8,rm8*	
01 /r	**addw**	*r16,rm16*	
01 /r	**addl**	*r32,rm32*	
	and		Logical AND
83 /4	**andl**	*imm8s,rm32*	
83 /4	**andw**	*imm8s,rm16*	
24	**andb**	*imm8,al*	
25	**andw**	*imm16,ax*	
25	**andl**	*imm32,eax*	
25	**andl**	*imm32*	

80 /4	**andb**	*imm8,rm8*	
81 /4	**andw**	*imm16,rm16*	
81 /4	**andl**	*imm32,rm32*	
22 /r	**andb**	*rm8,r8*	
23 /r	**andw**	*rm16,r16*	
23 /r	**andl**	*rm32,r32*	
20 /r	**andb**	*r8,rm8*	
21 /r	**andw**	*r16,rm16*	
21 /r	**andl**	*r32,rm32*	
63 /r	**arpl**	*r16,rm16*	Adjust RPL field of selector
	bound		Check if register is within bounds
62 /r	**boundw**	*m16,r16*	
62 /r	**boundl**	*m32,r32*	
	bsf		Bit scan forward
0F BC	**bsfw**	*rm16,r16*	
0F BC	**bsfl**	*rm32,r32*	
	bsr		Bit scan reverse
0F BD	**bsrw**	*rm16,r16*	
0F BD	**bsrl**	*rm32,r32*	
	bt		Save bit in carry flag
0F A3	**btw**	*r16,rm16*	
0F A3	**btl**	*r32,rm32*	
0F BA /4	**btw**	*imm8,rm16*	
0F BA /4	**btl**	*imm8,rm32*	
	btc		Bit test and complement
0F BB	**btcw**	*r16,rm16*	
0F BB	**btcl**	*r32,rm32*	
0F BA /7	**btcw**	*imm8,rm16*	
0F BA /7	**btcl**	*imm8,rm32*	
	btr		Bit test and reset
0F B3	**btrw**	*r16,rm16*	
0F B3	**btrl**	*r32,rm32*	
0F BA /6	**btrw**	*imm8,rm16*	
0F BA /6	**btrl**	*imm8,rm32*	
	bts		Bit test and set
0F AB	**btsw**	*r16,rm16*	
0F AB	**btsl**	*r32,rm32*	
0F BA /5	**btsw**	*imm8,rm16*	
0F BA /5	**btsl**	*imm8,rm32*	
E8	**call**	*reli*	Call Procedure
98	**cbtw**		Sign extend AL
98	**cbw**		Sign extend AL
99	**cdq**		Double word to quad word
F8	**clc**		Clear carry
FC	**cld**		Clear direction Flag
FA	**cli**		Clear interrupt Flag
99	**cltd**		Double word to quad word
0F 06	**clts**		Clear task-switched flag in CR0
F5	**cmc**		Complement carry flag
	cmp		Compare
83 /7	**cmpl**	*imm8s,rm32*	
83 /7	**cmpw**	*imm8s,rm16*	
3C	**cmpb**	*imm8,al*	
3D	**cmpw**	*imm16,ax*	
3D	**cmpl**	*imm32,eax*	
3D	**cmpl**	*imm32*	

80 /7	**cmpb**	*imm8,rm8*	
81 /7	**cmpw**	*imm16,rm16*	
81 /7	**cmpl**	*imm32,rm32*	
3A /r	**cmpb**	*rm8,r8*	
3B /r	**cmpw**	*rm16,r16*	
3B /r	**cmpl**	*rm32,r32*	
38 /r	**cmpb**	*r8,rm8*	
39 /r	**cmpw**	*r16,rm16*	
39 /r	**cmpl**	*r32,rm32*	
A6	**cmpsb**		Compare bytes
A7	**cmpsl**		Compare long
A7	**cmpsw**		Compare words
99	**cwd**		Word to double word
98	**cwde**		Sign extend AX
99	**cwtd**		Word to double word
98	**cwtl**		Sign extend AX
27	**daa**		Decimal adjust after addition
2F	**das**		Decimal adjust after subtraction
	dec		Decrement by 1
48 +r	**decw**	*r16*	
48 +r	**decl**	*r32*	
FE /1	**decb**	*rm8*	
FF /1	**decw**	*rm16*	
FF /1	**decl**	*rm32*	
	div		Unsigned divide
F6 /6	**divb**	*rm8,al*	
F6 /6	**divb**	*rm8*	
F7 /6	**divw**	*rm16,ax*	
F7 /6	**divw**	*rm16*	
F7 /6	**divl**	*rm32,eax*	
F7 /6	**divl**	*rm32*	
C8	**enter**	*imm8,imm16*	Make stack frame for procedure
D9 F0	**f2xm1**		ST = 2 ** ST - 1
D9 E1	**fabs**		ST = abs(ST)
	fadd		Floating add
D8 /0	**fadds**	*m32*	
DC /0	**faddl**	*m64*	
D8 C0 +r	**fadd**	*fpreg,st0*	
D8 C0 +r	**fadd**	*fpreg*	
DE C1	**fadd**		
DC C0 +r	**fadd**	*st0,fpreg*	
	faddp		Floating add and pop
DE C0 +r	**faddp**	*st0,fpreg*	
DE C0 +r	**faddp**	*fpreg*	
DE C1	**faddp**		
DF /4	**fbld**	*m80*	Load Binary Coded Decimal
DF /6	**fbstp**	*m80*	Store Binary Coded Decimal and Pop
D9 E0	**fchs**		Change Floating Sign
9B DB E2	**fclex**		Clear floating point exception flags
	fcom		Floating Compare
D8 /2	**fcoms**	*m32*	
DC /2	**fcoml**	*m64*	
D8 D0 +r	**fcom**	*fpreg,st0*	
D8 D0 +r	**fcom**	*fpreg*	
D8 D1	**fcom**		
	fcomp		Floating Compare and Pop

D8 /3	**fcomps**	*m32*	
DC /3	**fcompl**	*m64*	
D8 D8 +r	**fcomp**	*fpreg*	
D8 D9	**fcomp**		
DE D9	**fcompp**		Floating Compare and pop twice
D9 FF	**fcos**		Cosine
D9 F6	**fdecstp**		Decrement Stack Top Pointer
	fdiv		Floating divide
D8 /6	**fdivs**	*m32*	
DC /6	**fdivl**	*m64*	
D8 F0 +r	**fdiv**	*fpreg,st0*	
D8 F0 +r	**fdiv**	*fpreg*	
DE F1	**fdiv**		
DC F0 +r	**fdiv**	*st0,fpreg*	
	fdivp		Floating divide and pop
DE F0 +r	**fdivp**	*st0,fpreg*	
DE F0 +r	**fdivp**	*fpreg*	
DE F1	**fdivp**		
	fdivr		Reverse floating divide
D8 /7	**fdivrs**	*m32*	
DC /7	**fdivrl**	*m64*	
D8 F8 +r	**fdivr**	*fpreg,st0*	
D8 F8 +r	**fdivr**	*fpreg*	
DE F9	**fdivr**		
DC F8 +r	**fdivr**	*st0,fpreg*	
	fdivrp		Reverse floating divide and pop
DE F8 +r	**fdivrp**	*st0,fpreg*	
DE F8 +r	**fdivrp**	*fpreg*	
DE F9	**fdivrp**		
DD C0 +r	**ffree**	*fpreg*	Free Floating Point Register
	fiadd		Add integer to float
DA /0	**fiaddl**	*m32*	
DE /0	**fiadds**	*m16*	
	ficom		Compare float to integer
DA /2	**ficoml**	*m32*	
DE /2	**ficoms**	*m16*	
	ficomp		Compare float to integer and pop
DA /3	**ficompl**	*m32*	
DE /3	**ficomps**	*m16*	
	fidiv		Divide float by integer
DA /6	**fidivl**	*m32*	
DE /6	**fidivs**	*m16*	
	fidivr		Reverse divide integer by float
DA /7	**fidivrl**	*m32*	
DE /7	**fidivrs**	*m16*	
	fild		Load integer
DB /0	**fildl**	*m32*	
DF /0	**filds**	*m16*	
DF /5	**fildll**	*m64*	
	fimul		Multiply integer to float
DA /1	**fimull**	*m32*	
DE /1	**fimuls**	*m16*	
D9 F7	**fincstp**		Increment Stack Top Pointer

9B DB E3	**finit**		Initialize Floating Point Unit
	fist		Store integer
DB /2	**fistl**	*m32*	
DF /2	**fists**	*m16*	
	fistp		Store integer and pop
DB /3	**fistpl**	*m32*	
DF /3	**fistps**	*m16*	
DF /7	**fistpll**	*m32*	
	fisub		Subtract integer from float
DA /4	**fisubl**	*m32*	
DE /4	**fisubs**	*m16*	
	fisubr		Reverse subtract integer from float
DA /5	**fisubrl**	*m32*	
DE /5	**fisubrs**	*m16*	
	fld		Load Real
D9 C0 +r	**fld**	*fpreg*	
D9 /0	**flds**	*m32*	
DD /0	**fldl**	*m32*	
DB /5	**fldt**	*m64*	
D9 E8	**fld1**		Load Constant 1
D9 /5	**fldcw**	*m32*	Load Floating Point Control Word
D9 /4	**fldenv**	*m32*	Load FPU Environment
D9 EA	**fldl2e**		Load Constant log(e) base 2
D9 E9	**fldl2t**		Load Constant log(10) base 2
D9 EC	**fldlg2**		Load Constant log(2) base 10
D9 ED	**fldln2**		Load Constant log(2) base e
D9 EB	**fldpi**		Load Constant pi
D9 EE	**fldz**		Load Constant 0.0
	fmul		Floating multiply
D8 /1	**fmuls**	*m32*	
DC /1	**fmull**	*m64*	
D8 C8 +r	**fmul**	*fpreg,st0*	
D8 C8 +r	**fmul**	*fpreg*	
DE C9	**fmul**		
DC C8 +r	**fmul**	*st0,fpreg*	
	fmulp		Floating multiply and pop
DE C8 +r	**fmulp**	*st0,fpreg*	
DE C8 +r	**fmulp**	*fpreg*	
DE C9	**fmulp**		
DB E2	**fnclex**		Clear floating point exception flags no wait
DB E3	**fninit**		Initialize Floating Point Unit no wait
D9 D0	**fnop**		No Operation
DD /6	**fnsave**	*m32*	Store FPU State no wait
D9 /7	**fnstcw**	*m32*	Store Control Word no wait
D9 /6	**fnstenv**	*m32*	Store FPU Environment no wait
	fnstsw		Store Status Word no wait
DD /7	**fnstsw**	*m16*	
DF E0	**fnstsw**	*ax*	
D9 F3	**fpatan**		Partial Arctangent
D9 F8	**fprem**		Partial Remainder toward 0
D9 F5	**fprem1**		Partial Remainder < 1/2 modulus
D9 F2	**fptan**		Partial Tangent
D9 FC	**frndint**		Round To Integer
DD /4	**frstor**	*m32*	Resore FPU State
DB F4	**frstpm**		set 287XL real mode (nop for 387/486)

9B DD /6	**fsave**	*m32*	Store FPU State
D9 FD	**fscale**		Scale
DB E4	**fsetpm**		set 287 protected mode (nop for 387/486)
D9 FE	**fsin**		Sine
D9 FB	**fsincos**		Sine and Cosine
D9 FA	**fsqrt**		Square Root
	fst		Store Real
DD D0 +r	**fst**	*fpreg*	
D9 /2	**fsts**	*m32*	
DD /2	**fstl**	*m64*	
9B D9 /7	**fstcw**	*m32*	Store Control Word
9B D9 /6	**fstenv**	*m32*	Store FPU Environment
	fstp		Store Real and pop
DD D8 +r	**fstp**	*fpreg*	
D9 /3	**fstps**	*m32*	
DD /3	**fstpl**	*m64*	
DB /7	**fstpt**	*m80*	
	fstsw		Store Status Word
9B DD /7	**fstsw**	*m16*	
9B DF E0	**fstsw**	*ax*	
	fsub		Floating subtract
D8 /4	**fsubs**	*m32*	
DC /4	**fsubl**	*m64*	
D8 E0 +r	**fsub**	*fpreg,st0*	
D8 E0 +r	**fsub**	*fpreg*	
DE E1	**fsub**		
DC E0 +r	**fsub**	*st0,fpreg*	
	fsubp		Floating subtract and pop
DE E0 +r	**fsubp**	*st0,fpreg*	
DE E0 +r	**fsubp**	*fpreg*	
DE E1	**fsubp**		
	fsubr		Reverse floating subtract
D8 /5	**fsubrs**	*m32*	
DC /5	**fsubrl**	*m64*	
D8 E8 +r	**fsubr**	*fpreg,st0*	
D8 E8 +r	**fsubr**	*fpreg*	
DE E9	**fsubr**		
DC E8 +r	**fsubr**	*st0,fpreg*	
	fsubrp		Reverse floating subtract and pop
DE E8 +r	**fsubrp**	*st0,fpreg*	
DE E8 +r	**fsubrp**	*fpreg*	
DE E9	**fsubrp**		
D9 E4	**ftst**		Test
	fucom		Unordered compare real
DD E0 +r	**fucom**	*st0,fpreg*	
DD E0 +r	**fucom**	*fpreg*	
DD E1	**fucom**		
	fucomp		Unordered compare real and pop
DD E8 +r	**fucomp**	*st0,fpreg*	
DD E8 +r	**fucomp**	*fpreg*	
DD E9	**fucomp**		
DA E9	**fucompp**		Unordered compare %st %st1 and pop twice
9B	**fwait**		Wait for coprocessor
D9 E5	**fxam**		Examine

	fxch		Floating exchange
D9 C8 +r	**fxch**	*st0,fpreg*	
D9 C8 +r	**fxch**	*fpreg,st0*	
D9 C8 +r	**fxch**	*fpreg*	
D9 C9	**fxch**		
D9 F4	**fxtract**		Extract Exponent and Significand
D9 F1	**fyl2x**		%st1 * log(%st) base 2
D9 F9	**fyl2xp1**		%st1 * log(%st + 1.0) base 2
F4	**hlt**		Halt
FF /2	**icall**	*rm32*	Call indirect
	idiv		Signed divide
F6 /7	**idivb**	*rm8,al*	
F6 /7	**idivb**	*rm8*	
F7 /7	**idivw**	*rm16,ax*	
F7 /7	**idivw**	*rm16*	
F7 /7	**idivl**	*rm32,eax*	
F7 /7	**idivl**	*rm32*	
FF /4	**ijmp**	*rm32*	Jump indirect
FF /3	**ilcall**	*m32*	Long call indirect
FF /5	**iljmp**	*m32*	Long jump indirect
	imul		Signed multiply
F6 /5	**imulb**	*rm8,al*	
F6 /5	**imulb**	*rm8*	
F7 /5	**imulw**	*rm16,ax*	
F7 /5	**imulw**	*rm16*	
F7 /5	**imull**	*rm32,eax*	
F7 /5	**imull**	*rm32*	
0F AF /r	**imulw**	*rm16,r16*	
0F AF /r	**imull**	*rm32,r32*	
6B	**imulw**	*imm8s,rm16,r16*	
6B	**imull**	*imm8s,rm32,r32*	
6B /r	**imulw**	*imm8s,r16*	
6B /r	**imull**	*imm8s,r32*	
69	**imulw**	*imm16,rm16,r16*	
69	**imull**	*imm32,rm32,r32*	
69 /r	**imulw**	*imm16,r16*	
69 /r	**imull**	*imm32,r32*	
	in		Input from port
E4	**inb**	*imm8*	
E5	**inw**	*imm8*	
E5	**inl**	*imm8*	
EC	**inb**	*(dx)*	
ED	**inw**	*(dx)*	
ED	**inl**	*(dx)*	
	inc		Increment by one
40 +r	**incw**	*r16*	
40 +r	**incl**	*r32*	
FE /0	**incb**	*rm8*	
FF /0	**incw**	*rm16*	
FF /0	**incl**	*rm32*	
6C	**insb**		Input byte from port into ES:(E)DI
6C	**insb**	*(dx)*	Input byte from port into ES:(E)DI
6D	**insl**		Input long from port into ES:(E)DI
6D	**insl**	*(dx)*	Input long from port into ES:(E)DI
6D	**insw**		Input word from port into ES:(E)DI
6D	**insw**	*(dx)*	Input word from port into ES:(E)DI
CC	**int**	*con3*	Interrupt 3

CD	**int**	*imm8*	Interrupt
CE	**into**		Int 4 if overflow is 1
CF	**iret**		Interrupt return
CF	**iretd**		Different mode different opcode ?
07	**ja**	*reli*	Jump if above
03	**jae**	*reli*	Jump if above or equal
02	**jb**	*reli*	Jump if below
06	**jbe**	*reli*	Jump if below or equal
02	**jc**	*reli*	Jump if carry
E3	**jcxz**	*rel8*	Jump if CX is zero
04	**je**	*reli*	Jump if equal
E3	**jecxz**	*rel8*	Jump if CX is zero
0F	**jg**	*reli*	Jump if greater
0D	**jge**	*reli*	Jump if greater or equal
0C	**jl**	*reli*	Jump if less
0E	**jle**	*reli*	Jump if less or equal
E9	**jmp**	*reli*	Jump absolute
06	**jna**	*reli*	Jump if not above
02	**jnae**	*reli*	Jump if not above or equal
03	**jnb**	*reli*	Jump if not below
07	**jnbe**	*reli*	Jump if not below or equal
03	**jnc**	*reli*	Jump if no carry
05	**jne**	*reli*	Jump if not equal
0E	**jng**	*reli*	Jump if not greater
0C	**jnge**	*reli*	Jump if not greater or equal
0D	**jnl**	*reli*	Jump if not less
0F	**jnle**	*reli*	Jump if not less or equal
01	**jno**	*reli*	Jump if not overflow
0B	**jnp**	*reli*	Jump if not parity
09	**jns**	*reli*	Jump if not sign
05	**jnz**	*reli*	Jump if not zero
00	**jo**	*reli*	Jump if overflow
0A	**jp**	*reli*	Jump if parity
0A	**jpe**	*reli*	Jump if parity even
0B	**jpo**	*reli*	Jump if parity odd
08	**js**	*reli*	Jump if sign
04	**jz**	*reli*	Jump if zero
04	**jz**	*reli*	Jump if zero
9F	**lahf**		Load flags into AH register
	lar		Load access rights byte
0F 02 /r	**larw**	*rm16,r16*	
0F 02 /r	**larl**	*rm32,r32*	
9A	**lcall**	*imm16x,imm32*	Long call
	lds		load full pointer DS:r16
C5 /r	**ldsw**	*m16,r16*	
C5 /r	**ldsl**	*m32,r32*	
	lea		Load effective address
8D /r	**leaw**	*m16,r16*	
8D /r	**leal**	*m32,r32*	
C9	**leave**		High level procedure exit
	les		Load full pointer ES:r16
C4 /r	**lesw**	*m16,r16*	
C4 /r	**lesl**	*m32,r32*	
	lfs		Load full pointer FS:r16
0F B4 /r	**lfsw**	*m16,r16*	
0F B4 /r	**lfsl**	*m32,r32*	
	lgdt		Load m into DGTR

0F 01 /2	**lgdtw**	*m16*	
0F 01 /2	**lgdtl**	*m32*	
	lgs		Load full pointer GS:r16
0F B5 /r	**lgsw**	*m16,r16*	
0F B5 /r	**lgsl**	*m32,r32*	
	lidt		Load m into IDTR
0F 01 /3	**lidtw**	*m16*	
0F 01 /3	**lidtl**	*m32*	
EA	**ljmp**	*imm16x,imm32*	Long jump
0F 00 /2	**lldt**	*rm16*	Load local descriptor table register
0F 01 /6	**lmsw**	*rm16*	Load machine status word
F0	**lock**		Assert lock signal for next instruction
AC	**lodsb**		Load string operand byte
AD	**lodsl**		Load string operand long
AD	**lodsw**		Load string operand word
E2	**loop**	*rel8*	Dec count jmp if count <> 0
E1	**loope**	*rel8*	Dec count jmp if count <> 0 and ZF = 1
E0	**loopne**	*rel8*	Dec count jmp if count <> 0 and ZF = 0
E0	**loopnz**	*rel8*	Dec count jmp if count <> 0 and ZF = 0
E1	**loopz**	*rel8*	Dec count jmp if count <> 0 and ZF = 1
CB	**lret**		Far return
CA	**lret**	*imm16*	Far return pop imm16 bytes of parms
	lsl		Load segment limit
0F 03 /r	**lslw**	*rm16,r16*	
0F 03 /r	**lsll**	*rm32,r32*	
	lss		Load full pointer SS:r16
0F B2 /r	**lssw**	*m16,r16*	
0F B2 /r	**lssl**	*m32,r32*	
0F 00 /3	**ltr**	*rm16*	Load task register
	mov		Move data
A0	**movb**	*moffs,al*	
A1	**movw**	*moffs,ax*	
A1	**movl**	*moffs,eax*	
A2	**movb**	*al,moffs*	
A3	**movw**	*ax,moffs*	
A3	**movl**	*eax,moffs*	
8A /r	**movb**	*rm8,r8*	
8B /r	**movw**	*rm16,r16*	
8B /r	**movl**	*rm32,r32*	
88 /r	**movb**	*r8,rm8*	
89 /r	**movw**	*r16,rm16*	
89 /r	**movl**	*r32,rm32*	
8C /r	**movw**	*sreg,rm16*	
8E /r	**movw**	*rm16,sreg*	
B0 +r	**movb**	*imm8,r8*	
B8 +r	**movw**	*imm16,r16*	
B8 +r	**movl**	*imm32,r32*	
C6	**movb**	*imm8,rm8*	
C7	**movw**	*imm16,rm16*	
C7	**movl**	*imm32,rm32*	
0F 20 /r	**movl**	*ctlreg,r32*	
0F 22 /r	**movl**	*r32,ctlreg*	
0F 21 /r	**movl**	*dbreg,r32*	
0F 23 /r	**movl**	*r32,dbreg*	
0F 24 /r	**movl**	*treg,r32*	
0F 26 /r	**movl**	*r32,treg*	
A4	**movsb**		Move bytes

A5	**movsl**		Move longs
A5	**movsw**		Move words
	movsx		Move with sign extend
0F BE /r	**movsxb**	*rm8,r16*	
0F BE /r	**movsxb**	*rm8,r32*	
0F BF /r	**movsxw**	*rm16,r32*	
0F BE /r	**movsbw**	*rm8,r16*	
0F BE /r	**movsbl**	*rm8,r32*	
0F BF /r	**movswl**	*rm16,r32*	
	movzx		Move with zero extend
0F B6 /r	**movzxb**	*rm8,r16*	
0F B6 /r	**movzxb**	*rm8,r32*	
0F B7 /r	**movzxw**	*rm16,r32*	
0F B6 /r	**movzbw**	*rm8,r16*	
0F B6 /r	**movzbl**	*rm8,r32*	
0F B7 /r	**movzwl**	*rm16,r32*	
	mul		Unsigned multiply
F6 /4	**mulb**	*rm8,al*	
F6 /4	**mulb**	*rm8*	
F7 /4	**mulw**	*rm16,ax*	
F7 /4	**mulw**	*rm16*	
F7 /4	**mull**	*rm32,eax*	
F7 /4	**mull**	*rm32*	
	neg		Negate
F6 /3	**negb**	*rm8*	
F7 /3	**negw**	*rm16*	
F7 /3	**negl**	*rm32*	
90	**nop**		No operation
	not		Invert bits
F6 /2	**notb**	*rm8*	
F7 /2	**notw**	*rm16*	
F7 /2	**notl**	*rm32*	
	or		Logical inclusive OR
83 /1	**orl**	*imm8s,rm32*	
83 /1	**orw**	*imm8s,rm16*	
0C	**orb**	*imm8,al*	
0D	**orw**	*imm16,ax*	
0D	**orl**	*imm32,eax*	
0D	**orl**	*imm32*	
80 /1	**orb**	*imm8,rm8*	
81 /1	**orw**	*imm16,rm16*	
81 /1	**orl**	*imm32,rm32*	
0A /r	**orb**	*rm8,r8*	
0B /r	**orw**	*rm16,r16*	
0B /r	**orl**	*rm32,r32*	
08 /r	**orb**	*r8,rm8*	
09 /r	**orw**	*r16,rm16*	
09 /r	**orl**	*r32,rm32*	
	out		Output from port
E6	**outb**	*imm8*	
E7	**outw**	*imm8*	
E7	**outl**	*imm8*	
EE	**outb**	*(dx)*	
EF	**outw**	*(dx)*	
EF	**outl**	*(dx)*	
6E	**outsb**		Output byte to port into ES:(E)DI

6F	**outsl**		Output long to port into ES:(E)DI
6F	**outsw**		Output word to port into ES:(E)DI
	pop		Pop a word from the stack
58 +r	**popw**	*r16*	
58 +r	**popl**	*r32*	
1F	**popw**	*ds*	
07	**popw**	*es*	
17	**popw**	*ss*	
0F A1	**popw**	*fs*	
0F A9	**popw**	*gs*	
8F /0	**popw**	*m16*	
8F /0	**popl**	*m32*	
	popa		Pop all
61	**popaw**		
61	**popal**		
	popf		Pop stack into flags
9D	**popfw**		
9D	**popfl**		
	push		Push a word on the stack
50 +r	**pushw**	*r16*	
50 +r	**pushl**	*r32*	
6A	**pushb**	*imm8s*	
68	**pushw**	*imm16*	
68	**pushl**	*imm32*	
0E	**pushw**	*cs*	
1E	**pushw**	*ds*	
06	**pushw**	*es*	
16	**pushw**	*ss*	
0F A0	**pushw**	*fs*	
0F A8	**pushw**	*gs*	
FF /6	**pushw**	*m16*	
FF /6	**pushl**	*m32*	
	pusha		Push all
60	**pushaw**		
60	**pushal**		
	pushf		Push flags
9C	**pushfw**		
9C	**pushfl**		
	rcl		Rotate carry left
D0 /2	**rclb**	*con1,rm8*	
D0 /2	**rclb**	*rm8*	
D2 /2	**rclb**	*cl,rm8*	
C0 /2	**rclb**	*imm8,rm8*	
D1 /2	**rclw**	*con1,rm16*	
D1 /2	**rclw**	*rm16*	
D3 /2	**rclw**	*cl,rm16*	
C1 /2	**rclw**	*imm8,rm16*	
D1 /2	**rcll**	*con1,rm32*	
D1 /2	**rcll**	*rm32*	
D3 /2	**rcll**	*cl,rm32*	
C1 /2	**rcll**	*imm8,rm32*	
	rcr		Rotate carry right
D0 /3	**rcrb**	*con1,rm8*	
D0 /3	**rcrb**	*rm8*	
D2 /3	**rcrb**	*cl,rm8*	
C0 /3	**rcrb**	*imm8,rm8*	
D1 /3	**rcrw**	*con1,rm16*	

D1 /3	**rcrw**	*rm16*	
D3 /3	**rcrw**	*cl,rm16*	
C1 /3	**rcrw**	*imm8,rm16*	
D1 /3	**rcrl**	*con1,rm32*	
D1 /3	**rcrl**	*rm32*	
D3 /3	**rcrl**	*cl,rm32*	
C1 /3	**rcrl**	*imm8,rm32*	
F3	**rep**		rep following instruction CX times
F3	**repe**		repe following instruction CX times or eq
F2	**repne**		repne following instruction CX times or ne
F2	**repnz**		alternate name for repnz
F3	**repz**		alternate name for repe
C3	**ret**		Return
C2	**ret**	*imm16*	Return pop imm16 bytes of parms
	rol		Rotate left
D0 /0	**rolb**	*con1,rm8*	
D0 /0	**rolb**	*rm8*	
D2 /0	**rolb**	*cl,rm8*	
C0 /0	**rolb**	*imm8,rm8*	
D1 /0	**rolw**	*con1,rm16*	
D1 /0	**rolw**	*rm16*	
D3 /0	**rolw**	*cl,rm16*	
C1 /0	**rolw**	*imm8,rm16*	
D1 /0	**roll**	*con1,rm32*	
D1 /0	**roll**	*rm32*	
D3 /0	**roll**	*cl,rm32*	
C1 /0	**roll**	*imm8,rm32*	
	ror		Rotate right
D0 /1	**rorb**	*con1,rm8*	
D0 /1	**rorb**	*rm8*	
D2 /1	**rorb**	*cl,rm8*	
C0 /1	**rorb**	*imm8,rm8*	
D1 /1	**rorw**	*con1,rm16*	
D1 /1	**rorw**	*rm16*	
D3 /1	**rorw**	*cl,rm16*	
C1 /1	**rorw**	*imm8,rm16*	
D1 /1	**rorl**	*con1,rm32*	
D1 /1	**rorl**	*rm32*	
D3 /1	**rorl**	*cl,rm32*	
C1 /1	**rorl**	*imm8,rm32*	
9E	**sahf**		Store AH into flags
	sal		Shift arithmetic left
D0 /4	**salb**	*con1,rm8*	
D0 /4	**salb**	*rm8*	
D2 /4	**salb**	*cl,rm8*	
C0 /4	**salb**	*imm8,rm8*	
D1 /4	**salw**	*con1,rm16*	
D1 /4	**salw**	*rm16*	
D3 /4	**salw**	*cl,rm16*	
C1 /4	**salw**	*imm8,rm16*	
D1 /4	**sall**	*con1,rm32*	
D1 /4	**sall**	*rm32*	
D3 /4	**sall**	*cl,rm32*	
C1 /4	**sall**	*imm8,rm32*	
	sar		Shift arithmetic right
D0 /7	**sarb**	*con1,rm8*	
D0 /7	**sarb**	*rm8*	
D2 /7	**sarb**	*cl,rm8*	

C0 /7	**sarb**	*imm8,rm8*	
D1 /7	**sarw**	*con1,rm16*	
D1 /7	**sarw**	*rm16*	
D3 /7	**sarw**	*cl,rm16*	
C1 /7	**sarw**	*imm8,rm16*	
D1 /7	**sarl**	*con1,rm32*	
D1 /7	**sarl**	*rm32*	
D3 /7	**sarl**	*cl,rm32*	
C1 /7	**sarl**	*imm8,rm32*	
	sbb		Subtract with borrow
83 /3	**sbbl**	*imm8s,rm32*	
83 /3	**sbbw**	*imm8s,rm16*	
1C	**sbbb**	*imm8,al*	
1D	**sbbw**	*imm16,ax*	
1D	**sbbl**	*imm32,eax*	
1D	**sbbl**	*imm32*	
80 /3	**sbbb**	*imm8,rm8*	
81 /3	**sbbw**	*imm16,rm16*	
81 /3	**sbbl**	*imm32,rm32*	
1A /r	**sbbb**	*rm8,r8*	
1B /r	**sbbw**	*rm16,r16*	
1B /r	**sbbl**	*rm32,r32*	
18 /r	**sbbb**	*r8,rm8*	
19 /r	**sbbw**	*r16,rm16*	
19 /r	**sbbl**	*r32,rm32*	
AE	**scasb**		Compare string bytes
AF	**scasl**		Compare string longs
AF	**scasw**		Compare string words
0F 97	**seta**	*rm8*	Set byte if above
0F 93	**setae**	*rm8*	Set byte if above or equal
0F 92	**setb**	*rm8*	Set byte if below
0F 96	**setbe**	*rm8*	Set byte if below or equal
0F 92	**setc**	*rm8*	Set byte if carry
0F 94	**sete**	*rm8*	Set byte if equal
0F 9F	**setg**	*rm8*	Set byte if greater
0F 9D	**setge**	*rm8*	Set byte if greater or equal
0F 9C	**setl**	*rm8*	Set byte if less
0F 9E	**setle**	*rm8*	Set byte if less or equal
0F 96	**setna**	*rm8*	Set byte if not above
0F 92	**setnae**	*rm8*	Set byte if not above or equal
0F 93	**setnb**	*rm8*	Set byte if not below
0F 97	**setnbe**	*rm8*	Set byte if not below or equal
0F 93	**setnc**	*rm8*	Set byte if no carry
0F 95	**setne**	*rm8*	Set byte if not equal
0F 9E	**setng**	*rm8*	Set byte if not greater
0F 9C	**setnge**	*rm8*	Set byte if not greater or equal
0F 9D	**setnl**	*rm8*	Set byte if not less
0F 9F	**setnle**	*rm8*	Set byte if not less or equal
0F 91	**setno**	*rm8*	Set byte if not overflow
0F 9B	**setnp**	*rm8*	Set byte if not parity
0F 99	**setns**	*rm8*	Set byte if not sign
0F 95	**setnz**	*rm8*	Set byte if not zero
0F 90	**seto**	*rm8*	Set byte if overflow
0F 9A	**setp**	*rm8*	Set byte if parity
0F 9A	**setpe**	*rm8*	Set byte if parity even
0F 9B	**setpo**	*rm8*	Set byte if parity odd
0F 98	**sets**	*rm8*	Set byte if sign
0F 94	**setz**	*rm8*	Set byte if zero
0F 94	**setz**	*rm8*	Set byte if zero
0F 01 /0	**sgdt**	*mem32*	Store gdtr

	shl		Shift arithmetic left
D0 /4	**shlb**	*con1,rm8*	
D0 /4	**shlb**	*rm8*	
D2 /4	**shlb**	*cl,rm8*	
C0 /4	**shlb**	*imm8,rm8*	
D1 /4	**shlw**	*con1,rm16*	
D1 /4	**shlw**	*rm16*	
D3 /4	**shlw**	*cl,rm16*	
C1 /4	**shlw**	*imm8,rm16*	
D1 /4	**shll**	*con1,rm32*	
D1 /4	**shll**	*rm32*	
D3 /4	**shll**	*cl,rm32*	
C1 /4	**shll**	*imm8,rm32*	
	shld		Shift double precision left
0F A4	**shldw**	*imm8,r16,rm16*	
0F A4	**shldl**	*imm8,r32,rm32*	
0F A5	**shldw**	*cl,r16,rm16*	
0F A5	**shldl**	*cl,r32,rm32*	
	shr		Shift right
D0 /5	**shrb**	*con1,rm8*	
D0 /5	**shrb**	*rm8*	
D2 /5	**shrb**	*cl,rm8*	
C0 /5	**shrb**	*imm8,rm8*	
D1 /5	**shrw**	*con1,rm16*	
D1 /5	**shrw**	*rm16*	
D3 /5	**shrw**	*cl,rm16*	
C1 /5	**shrw**	*imm8,rm16*	
D1 /5	**shrl**	*con1,rm32*	
D1 /5	**shrl**	*rm32*	
D3 /5	**shrl**	*cl,rm32*	
C1 /5	**shrl**	*imm8,rm32*	
	shrd		Shift double precision right
0F AC	**shrdw**	*imm8,r16,rm16*	
0F AC	**shrdl**	*imm8,r32,rm32*	
0F AD	**shrdw**	*cl,r16,rm16*	
0F AD	**shrdl**	*cl,r32,rm32*	
0F AD	**shrdw**	*r16,rm16*	
0F AD	**shrdl**	*r32,rm32*	
0F 01 /1	**sidt**	*mem32*	Store idtr
0F 00 /0	**sldt**	*rm16*	Store ldtr to EA word
0F 01 /4	**smsw**	*rm16*	Store machine status to EA word
F9	**stc**		Set carry flag
FD	**std**		Clear direction flag
FB	**sti**		Set interrupt flag
AA	**stosb**		Store string byte
AB	**stosl**		Store string long
AB	**stosw**		Store string word
0F 00 /1	**str**		Store task register
	sub		Subtract
83 /5	**subl**	*imm8s,rm32*	
83 /5	**subw**	*imm8s,rm16*	
2C	**subb**	*imm8,al*	
2D	**subw**	*imm16,ax*	
2D	**subl**	*imm32,eax*	
2D	**subl**	*imm32*	
80 /5	**subb**	*imm8,rm8*	
81 /5	**subw**	*imm16,rm16*	
81 /5	**subl**	*imm32,rm32*	

2A /r	**subb**	*rm8,r8*	
2B /r	**subw**	*rm16,r16*	
2B /r	**subl**	*rm32,r32*	
28 /r	**subb**	*r8,rm8*	
29 /r	**subw**	*r16,rm16*	
29 /r	**subl**	*r32,rm32*	
	test		Logical compare
A8	**testb**	*imm8,al*	
A9	**testw**	*imm16,ax*	
A9	**testl**	*imm32,eax*	
A9	**testl**	*imm32*	
F6 /0	**testb**	*imm8,rm8*	
F7 /0	**testw**	*imm16,rm16*	
F7 /0	**testl**	*imm32,rm32*	
84 /r	**testb**	*r8,rm8*	
85 /r	**testw**	*r16,rm16*	
85 /r	**testl**	*r32,rm32*	
0F 00 /4	**verr**	*rm16*	Verify segment for read
0F 00 /5	**verw**	*rm16*	Verify segment for write
9B	**wait**		Wait for coprocessor
	xchg		Exchange register
90 +r	**xchgw**	*r16,ax*	
90 +r	**xchgw**	*ax,r16*	
90 +r	**xchgl**	*r32,eax*	
90 +r	**xchgl**	*r32*	
90 +r	**xchgl**	*eax,r32*	
86 /r	**xchgb**	*rm8,r8*	
87 /r	**xchgw**	*rm16,r16*	
87 /r	**xchgl**	*rm32,r32*	
86 /r	**xchgb**	*r8,rm8*	
87 /r	**xchgw**	*r16,rm16*	
87 /r	**xchgl**	*r32,rm32*	
D7	**xlat**		Table lookup translation
D7	**xlatb**		Table lookup translation
	xor		Logical exclusive OR
83 /6	**xorl**	*imm8s,rm32*	
83 /6	**xorw**	*imm8s,rm16*	
34	**xorb**	*imm8,al*	
35	**xorw**	*imm16,ax*	
35	**xorl**	*imm32,eax*	
35	**xorl**	*imm32*	
80 /6	**xorb**	*imm8,rm8*	
81 /6	**xorw**	*imm16,rm16*	
81 /6	**xorl**	*imm32,rm32*	
32 /r	**xorb**	*rm8,r8*	
33 /r	**xorw**	*rm16,r16*	
33 /r	**xorl**	*rm32,r32*	
30 /r	**xorb**	*r8,rm8*	
31 /r	**xorw**	*r16,rm16*	
31 /r	**xorl**	*r32,rm32*	

Using C to Prototype Assembly Language

The COHERENT C compiler includes a switch, **-S**, that translates C code into COHERENT assembly language. The assembly language it produces cannot be directly assembled, but you can examine it to see just what the compiler does under given circumstances; and you can use it to prototype a routine in assembly language.

Suppose, for example, that you wish to write a function that takes two parameters: an integer, which gives a port number to read from; and an address where the data should go. Start by writing a C function with the correct calling sequence. For example, the following function is in a file called **proto.c**:

```
readstuff(addr, port)
register char *addr;
int port;
{
      register int dx = port;
      char *foo = addr;
}
```

Compile it with the following command line:

```
      cc -S proto.c
```

This produces file **proto.s**, which contains the following:

```
/     module name foo
      .alignoff

      .text
      .globl readstuff
readstuff:
      push  %ebp
      movl  %ebp, %esp
      push  %esi
      push  %edi
      push  %ebx
      movl  %ebx, 8(%ebp)
      movl  %esi, 12(%ebp)
      movl  %edi, %ebx
      pop   %ebx
      pop   %edi
      pop   %esi
      leave
      ret
      .align      4
```

This is your prototype. You can easily modify it into what you want; for example:

```
/     This will only work if you install it as a driver.
/     As the operating system will protect itself if
/     Ordinary users try to access ports. Ask about our
/     Device driver kits.

       .text
       .globl readstuff
readstuff:
       push       %ebp
       movl       %ebp, %esp
       push       %edi                      / Save the edi for the caller
       movl       %edx, 8(%ebp)             / Get the port number
       movl       %edi, 12(%ebp)            / Get the user address

      insb        / Read port (%dx) to %es:%edi

      pop   %edi  / See 386 calling conventions
      leave
      ret
```

Example

The following example echoes strings onto your screen.

```
/ sstatic void foo(i) { printf("Parm is %d\n", i); }

      .text
.L2:  .byte "Parm is %d0, 0
```

```
foo:
      push  %ebp        / set up stack frame
      movl  %ebp, %esp
      push  8(%ebp)           / push parms from right to left
      push  $.L2
      call  printf
      leave             / %esp <- %ebp; pop %ebp
      ret

/ main() { foo(5); }

      .globl main
main:
      push  %ebp
      movl  %ebp, %esp
      push  $5
      call  foo
      leave
      ret
```

See Also

asfix, calling conventions, cc, cdmp, commands
Intel Corporation: *386 DX Programmer's Reference Manual.* Santa Clara, CA: Intel Corporation, 1990. *Highly recommended.*

Diagnostics

The following gives the error messages returned by **as**. The messages are in alphabetical order. Each message is marked as to its degree of severity: A *fatal* message usually indicates a condition that caused the assembler to terminate execution. Often, they indicate internal problems in the assembler. An *error* message points to a condition in the source code that the assembler cannot resolve. This almost always occurs when the program does something illegal. A *warning* message points out code that is compilable, but may produce trouble when the program is executed.

.align must be 1, 2 or 4 *(error)*
: **.align** must work after the link. These are the only values for which this can be true.

Ambiguous operand length, *n* bytes selected *(warning)*
: The assembler cannot tell the operand length by looking at the opcode and the operands. You may want to do something like change **mov** to **movl**.

Arithmetic between addresses on different segments *(error)*
: You may only add or subtract addresses if they are in the same segment.

Bad scale *(error)*
: Address scale must be 0, 1, 2, 4, or 8.

16 bit addressing mode used in 32 bit code *(warning)*
: You probably don't want to do this. For example, you may want to say **(%esi)**, not **(%si)**.

32 bit addressing mode used in 16 bit code *(warning)*
: You probably don't want to do this. For example, you may want to say **(%si)**, not **(%esi)**.

Cannot fopen(*string*, *string*) *(fatal)*
Cannot insert \0 in string *(error)*
: NUL (\0) terminates strings. Instead of

```
      .byte "hello\n\0"
```

  use:

```
      .byte "hello\n", 0
```

Character constant *n* long *(error)*
: Character constants must be one byte long.

.comm must have tag *(error)*
: The format of **.comm** is **.comm name, size**.

Command option '*c*' missing its argument *(fatal)*
Data defined in .bss *(error)*
The **.bss** segment is uninitialized data. You cannot place actual values there.

.define must have a label *(error)*
Duplicate symbol '*string*' *(error)*
symbol is defined on two different lines.

.else detected logic type *n (fatal)*
Logic error in assembler. Please report this problem to Mark Williams technical support.

End of line after backslash reading parm *(error)*
Macro parmeters may not be broken up with backslash.

End of line after backslash *(error)*
End of line detected in character constant *(error)*
End of line detected in string *(error)*
End of macro building .while *(error)*
A **.macro** ended while reading in a **.while** loop.

.endi detected logic type *n (fatal)*
Logic error in assembler. Please report this problem to Mark Williams technical support.

Error in binary number *(error)*
Error in octal number *(error)*
Found *n* parms expected *n (error)*
Illegal combination of opcode and operands *(error)*
Although the opcode is valid and the operands are valid, there is no form of this opcode which takes this combination of operands in this order.

Illegal use of local symbol *(error)*
Illegal use of of predefined symbol *string*. *(error)*
Improper instruction following lock *(warning)*
Only a few instructions are valid after a lock instruction. See your machine documentation for details.

Improper instruction following rep *(warning)*
Only a few instructions are valid after a rep instruction. See your machine documentation for details.

Indirect mode on invalid instruction *(error)*
Indirection is only allowed on call and jump near instructions.

Internal error relative branch logic *(fatal)*
Logic error in assembler. Please report this problem to Mark Williams technical support.

Invalid .mlist option must be on or off *(error)*
Invalid character '*c*' *string* at position *n (error)*
Invalid character 0x*0xn string* at position *n (error)*
Invalid data type, must be symbol *(error)*
Invalid floating point register number *(error)*
Invalid opcode: '*string*' *(error)*
The string in the opcode position is not one of our opcodes or one of your macros.

Invalid operand type *(error)*
string is an improper register in this context *(error)*
Label ignored *(error)*
This statement cannot take a label.

Label on invalid operator *(error)*
Label required *(error)*
Length *n* string range exceeded *(error)*
Strings may not exceed 32 kilobytes.

Logic error in macro def '*string*' *n (fatal)*
Logic error in assembler. Please report this problem to Mark Williams technical support.

Logic error in umark *(fatal)*
Logic error in assembler. Please report this problem to Mark Williams technical support.

LEXICON

Macro definition must have a label *(error)*
.mexit not in macro *(error)*
Missing .endi *(error)*
Input ended leaving **.if** open.

Missing .endm *(error)*
Input ended leaving **.macro** open.

Missing .endw *(error)*
Input ended leaving **.while** open.

Mixed 386/286 addressing modes *(error)*
No opcode allows mixed 286 and 386 addressing modes.

Mixed 386/286 data modes *(error)*
No 386 opcode allows mixed 286 and 386 data modes.

Mixed length addressing registers *(error)*
Addressing registers must both be the same length.

more than one file to process *(fatal)*
The assembler will only process one file at a time.

Name required *(error)*
The format of **set** is **.set name, value**

no work *(fatal)*
There were no files listed on the command line.

NULL address in relative branch *(fatal)*
Logic error in assembler. Please report this problem to Mark Williams technical support.

Octal number *n* truncated to char *(error)*
An octal number in a string was too big.

Optype *n* in lex *(fatal)*
Logic error in assembler. Please report this problem to Mark Williams technical support.

Org to invalid value *(error)*
You may not **.org** to doubles or strings.

Org to wrong segment *(error)*
You must **.org** to the current segment.

Out of space *(fatal)*
A call to **malloc()** failed. The typical large consumers of RAM are macros and **.defines**; symbols consume less. Can you break your assembly into smaller pieces? Could you be in some sort of endless recursion or loop?

Parm *n* not found *(error)*
An attempt to **.shift** too far has been made.

.parmct not in macro *(error)*
.parmct returns the number of parameters in the current macro.

Phase error '*string*' *(error)*
A symbol is defined one way in one phase of the assembly and another way in the next phase.

Redefinition of '*string*' *(error)*
An assembler internal symbol is being redefined.

Seek error on object file *(fatal)*
Seek error on object file *(fatal)*
.shift not in macro *(error)*
.shift shifts macro parameters. It has no meaning outside a macro.

String must be on .byte *(error)*
For example:

```
        .byte "This is how we place a string", 0
```

Symbol may not be double *(error)*
: You may not convert a symbol to a floating-point value.

Symbol may not be float *(error)*
: You may not convert a symbol to a floating-point value.

Syntax error *(error)*
: The syntax of this statement makes no sense to the parser. This can be a variety of problems.

Table error kind *0xn* detected *(fatal)*
: Logic error in assembler. Please report this problem to Mark Williams technical support.

This code may not work the same way on all chips *(warning)*
: Some chips may not execute this code as expected.

Too many operands *(error)*
: No 386 opcode has more than three operands.

Undefined symbol '*string*' *(error)*
: A symbol was used without defining it or using a **-g** option. You must define local symbols.

Unexpected .else statement *(error)*
Unexpected .endi statement *(error)*
Unexpected .endm ignored *(error)*
Unexpected .endw *(error)*
Unexpected return from parser *(fatal)*
: Logic error in assembler. Please report this problem to Mark Williams technical support.

Unknown command option *c (fatal)*
Unlikely output file '*string*' *(fatal)*
: Output file-names should have **.o** suffixes. Because this is generally a typographical error, **as** aborts to avoid overwriting an important file.

Unmatched '*c*' *(error)*
: A delimeter, [, (,), or] is unmatched in this command.

Unmatched bracket in parmeter *(error)*
: Line ended leaving an open bracket or parenthesis.

Write error on object file *(fatal)*
: **as** could not write the object module. This error can have any of several causes; the most common is that you lack permission to write into the current directory, or you lack permission to overwrite an existing file of the same name.

Notes

We have designed **as** to ease porting of programs written in other dialects of UNIX 386 assembly language, as well as to be a powerful tool for development of new programs. We think you will find the features and documentation of our assembler considerably more complete than are available anywhere else. However, we have chosen *not* to duplicate behavior of other assemblers that leads to inefficient or incorrect output, or that generates code without warning when given questionable input. We have also chosen to support operator precedence rather than perpetuating antiquated left-to-right evaluation schemes seen elsewhere. *Caveat utilitor.*

In the course of writing this assembler, we have discovered that the UNIX implementation of **fdiv**, **fdivr**, **fsub**, and **fsubr** differs from that described in the Intel documents. The COHERENT assembler conforms to the UNIX standard by default. You should be very careful with the order of operands to these instructions. Once again, *caveat utilitor*.

ASCII — Definition

ASCII is an acronym for the American Standard Code for Information Interchange, as defined by the American National Standards Institute standard X3.4-1977. It is a table of seven-bit binary numbers that encode the letters of the alphabet, numerals, punctuation, and the most commonly used control sequences for printers and terminals. ASCII codes are used on all microcomputers sold in the United States.

The following table gives the ASCII characters in octal, decimal, and hexadecimal numbers, their definitions, and expands abbreviations where necessary.

000	0	0x00	NUL	**<ctrl-@>**	Null character
001	1	0x01	SOH	**<ctrl-A>**	Start of header
002	2	0x02	STX	**<ctrl-B>**	Start of text
003	3	0x03	ETX	**<ctrl-C>**	End of text
004	4	0x04	EOT	**<ctrl-D>**	End of transmission
005	5	0x05	ENQ	**<ctrl-E>**	Enquiry
006	6	0x06	ACK	**<ctrl-F>**	Positive acknowledgement
007	7	0x07	BEL	**<ctrl-G>**	Bell
010	8	0x08	BS	**<ctrl-H>**	Backspace
011	9	0x09	HT	**<ctrl-I>**	Horizontal tab
012	10	0x0A	LF	**<ctrl-J>**	Line feed
013	11	0x0B	VT	**<ctrl-K>**	Vertical tab
014	12	0x0C	FF	**<ctrl-L>**	Form feed
015	13	0x0D	CR	**<ctrl-M>**	Carriage return
016	14	0x0E	SO	**<ctrl-N>**	Shift out
017	15	0x0F	SI	**<ctrl-O>**	Shift in
020	16	0x10	DLE	**<ctrl-P>**	Data link escape
021	17	0x11	DC1	**<ctrl-Q>**	Device control 1 (XON)
022	18	0x12	DC2	**<ctrl-R>**	Device control 2 (tape on)
023	19	0x13	DC3	**<ctrl-S>**	Device control 3 (XOFF)
024	20	0x14	DC4	**<ctrl-T>**	Device control 4 (tape off)
025	21	0x15	NAK	**<ctrl-U>**	Negative acknowledgement
026	22	0x16	SYN	**<ctrl-V>**	Synchronize
027	23	0x17	ETB	**<ctrl-W>**	End of transmission block
030	24	0x18	CAN	**<ctrl-X>**	Cancel
031	25	0x19	EM	**<ctrl-Y>**	End of medium
032	26	0x1A	SUB	**<ctrl-Z>**	Substitute
033	27	0x1B	ESC	**<ctrl-[>**	Escape
034	28	0x1C	FS	**<ctrl-\>**	Form separator
035	29	0x1D	GS	**<ctrl-]>**	Group separator
036	30	0x1E	RS	**<ctrl-^>**	Record separator
037	31	0x1F	US	**<ctrl-_>**	Unit separator
040	32	0x20	SP	Space	
041	33	0x21	!	Exclamation point	
042	34	0x22	"	Quotation mark	
043	35	0x23	#	Pound sign (sharp)	
044	36	0x24	$	Dollar sign	
045	37	0x25	%	Percent sign	
046	38	0x26	&	Ampersand	
047	39	0x27	’	Apostrophe	
050	40	0x28	(	Left parenthesis	
051	41	0x29	)	Right parenthesis	
052	42	0x2A	*	Asterisk	
053	43	0x2B	+	Plus sign	
054	44	0x2C	,	Comma	
055	45	0x2D	-	Hyphen (minus sign)	
056	46	0x2E	.	Period	
057	47	0x2F	/	Virgule (slash)	
060	48	0x30	0		
061	49	0x31	1		
062	50	0x32	2		
063	51	0x33	3		
064	52	0x34	4		
065	53	0x35	5		
066	54	0x36	6		
067	55	0x37	7		
070	56	0x38	8		
071	57	0x39	9		
072	58	0x3A	:	Colon	
073	59	0x3B	;	Semicolon	
074	60	0x3C	<	Less-than symbol (left angle bracket)	

075	61	0x3D	=	Equal sign
076	62	0x3E	>	Greater-than symbol (right angle bracket)
077	63	0x3F	?	Question mark
0100	64	0x40	@	At sign
0101	65	0x41	A	
0102	66	0x42	B	
0103	67	0x43	C	
0104	68	0x44	D	
0105	69	0x45	E	
0106	70	0x46	F	
0107	71	0x47	G	
0110	72	0x48	H	
0111	73	0x49	I	
0112	74	0x4A	J	
0113	75	0x4B	K	
0114	76	0x4C	L	
0115	77	0x4D	M	
0116	78	0x4E	N	
0117	79	0x4F	O	
0120	80	0x50	P	
0121	81	0x51	Q	
0122	82	0x52	R	
0123	83	0x53	S	
0124	84	0x54	T	
0125	85	0x55	U	
0126	86	0x56	V	
0127	87	0x57	W	
0130	88	0x58	X	
0131	89	0x59	Y	
0132	90	0x5A	Z	
0133	91	0x5B	[	Left bracket (left square bracket)
0134	92	0x5C	\	Backslash
0135	93	0x5D	]	Right bracket (right square bracket)
0136	94	0x5E	^	Circumflex
0137	95	0x5F	_	Underscore
0140	96	0x60	`	Grave
0141	97	0x61	a	
0142	98	0x62	b	
0143	99	0x63	c	
0144	100	0x64	d	
0145	101	0x65	e	
0146	102	0x66	f	
0147	103	0x67	g	
0150	104	0x68	h	
0151	105	0x69	i	
0152	106	0x6A	j	
0153	107	0x6B	k	
0154	108	0x6C	l	
0155	109	0x6D	m	
0156	110	0x6E	n	
0157	111	0x6F	o	
0160	112	0x70	p	
0161	113	0x71	q	
0162	114	0x72	r	
0163	115	0x73	s	
0164	116	0x74	t	
0165	117	0x75	u	
0166	118	0x76	v	
0167	119	0x77	w	
0170	120	0x78	x	
0171	121	0x79	y	
0172	122	0x7A	z	

0173	123	0x7B	{	Left brace (left curly bracket)
0174	124	0x7C	|	Vertical bar
0175	125	0x7D	}	Right brace (right curly bracket)
0176	126	0x7E	~	Tilde
0177	127	0x7F	DEL	**<ctrl->** Delete

Files

/usr/pub/ascii

See Also

Latin 1, Programming COHERENT

asctime() — Time Function (libc)

Convert time structure to ASCII string
#include <time.h>
#include <sys/types.h>
char *asctime(*tmp***)**
struct tm ****tmp*;**

asctime() takes the data found in *tmp*, and turns it into an ASCII string. *tmp* is of the type **tm**, which is a structure defined in the header file **time.h**. This structure must first be initialized by either **gmtime()** or **localtime()** before it can be used by **asctime()**. For a further discussion of **tm**, see the entry for **time**.

asctime() returns a pointer to where it writes the text string it creates.

Example

The following example demonstrates the functions **asctime()**, **ctime()**, **gmtime()**, **localtime()**, and **time()**, and shows the effect of the environmental variable **TIMEZONE**. For a discussion of the variable **time_t**, see the entry for **time()**.

```
#include <time.h>
#include <sys/types.h>
main()
{
      time_t timenumber;
      struct tm *timestruct;

      /* read system time, print using ctime */
      time(&timenumber);
      printf("%s", ctime(&timenumber));

      /* use gmtime to fill tm, print with asctime */
      timestruct = gmtime(&timenumber);
      printf("%s", asctime(timestruct));

      /* use localtime to fill tm, print with asctime */
      timestruct = localtime(&timenumber);
      printf("%s", asctime(timestruct));
}
```

See Also

libc, time(), time [overview]
ANSI Standard, §7.12.3.1
POSIX Standard, §8.1.1

Notes

asctime() returns a pointer to a statically allocated data area that is overwritten by successive calls.

asfix — Command

Convert assembly-language programs into 80386 format
asfix < *oldfile* **>** *newfile*

The command **asfix** converts programs written in the COHERENT-286 assembly language into a form that can be assembled by the COHERENT 386 edition of **as**, the COHERENT assembler.

asfix reads the standard input and writes to the standard output. It changes DEC-form local symbols to the form

recognized by COHERENT-386 **as**, changes character constants from the form **'x** to the form **'x'**, and changes local symbols from the COHERENT-286 form to the COHERENT-386 form.

See Also

as, commands

ASHEAD — Environmental Variable

Append options to beginning of as command line
export ASHEAD=*options*

The COHERENT assembler **as** reads the environmental variables **ASHEAD** and **ASTAIL** before it begins its work. You can set these variables to hold the default options that you want the assembler always to use.

as appends the options in **ASHEAD** to the beginning of its command line.

See Also

as, ASTAIL, environmental variables

asin() — Mathematics Function (libm)

Calculate inverse sine
#include <math.h>
double asin(*arg***) double** *arg***;**

asin() calculates the inverse sine of *arg*, which must be in the range [-1., 1.]. The result will be in the range [-π/2, π/2].

If all goes well, **asin()** returns the inverse sine. However, if *arg* is out of range, **asin()** sets **errno** to **EDOM** and returns zero.

Example

For an example of this function, see the entry for **tan()**.

See Also

libm, sin()
ANSI Standard, §7.5.2.2
POSIX Standard, §8.1

ASKCC — Environmental Variable

Force prompting for CC names
ASKCC=*YES/NO*

The environmental variable **ASKCC** directs the mailer program **mail** to prompt for carbon-copy names. A carbon-copy (or CC) name gives another person to whom a mail message should be sent. To turn on prompting, use the command:

```
export ASKCC=YES
```

See Also

environmental variables, mail

assert() — Macro Diagnostics (assert.h)

Check assertion at run time
#include <assert.h>
void assert(*outcome***)**
int *outcome***;**

assert() checks the value of *outcome*, which usually is the product of an expression. If *outcome* is false (zero), **assert()** sends a message into the standard-error stream and calls **exit()**. It is useful for verifying that a necessary condition is true.

The error message includes the text of the assertion that failed, the name of the source file, and the line within the source file that holds the expression in question. These last two elements consist, respectively, of the values of the preprocessor macros _ _**FILE**_ _ and _ _**LINE**_ _.

assert() calls **exit()**, which never returns.

To turn off **assert()**, define the macro **NDEBUG** prior to including the header **assert.h**. This forces **assert()** to be redefined as

```
#define assert(ignore)
```

See Also

exit(), assert.h, C preprocessor,
ANSI Standard, §7.2.1.1
POSIX Standard, §8.1

Notes

The ANSI Standard requires that **assert()** be implemented as a macro, not a library function. If a program suppresses the macro definition in favor of a function call, its behavior is undefined.

Turning off **assert()** with the macro **NDEBUG** will affect the behavior of a program if the expression being evaluated normally generates side effects.

assert() is useful for debugging, and for testing boundary conditions for which more graceful error recovery has not yet been implemented.

assert.h — Header File

Define assert()
#include <assert.h>

assert.h is the header file that defines the macro **assert()**.

See Also

assert(), header files,
ANSI Standard, §7.2

ASTAIL — Environmental Variable

Append options to end of as command line
export ASTAIL=*options*

The COHERENT assembler **as** reads the environmental variables **ASHEAD** and **ASTAIL** before it begins its work. You can set these variables to hold the default options that you want the assembler always to use.

as appends the options in **ASTAIL** to the end of its command line.

See Also

as, ASHEAD, environmental variables,

asy — Device Driver

Device driver for asynchronous serial lines

The device driver **asy** supports serial ports. It uses major number 5.

asy can handle from one to 32 serial ports. The ports can be any mixture of 8250, 8250B, 16550, 16550A, and equivalent devices, including nearly all conventional COM1 through COM4 serial cards, and most non-intelligent multiport add-in cards. It automatically recognizes, and uses, on-chip FIFO, and it can specify groups of ports that share a single interrupt status.

Types of Port Configuration

Each port that **asy** serves has a base name, e.g., **/dev/com1r**. Each has its own minor device number. Different configurations of the port are selected by using different suffixes, as follows:

l (Local) "Local mode" means that the line will have a terminal plugged into it, or is connected to a modem running in command mode. Local mode uses the minor device with the modem-control bit (bit 7) set.

r (Remote) "Modem control" means that the line will have a modem plugged into it. Modem control is enabled on a serial line by resetting the modem control bit (bit 7) in the minor number for the device. This allows the system to generate a hangup signal when the modem indicates loss of carrier by dropping DCD (Data Carrier Detect). A modem line should always have its DSR, DCD and CTS pins connected. If left hanging, spurious

transitions can cause severe system thrashing. An **open()** to a modem-control line will block until a carrier is detected (i.e., until DCD goes true).

p (Polled mode) "Polled mode" means that the port cannot generate an interrupt, but must be checked (or polled) constantly by the COHERENT system to see if activity has occurred on it. Such polling takes a significant toll on system performance. The main reason for supporting polled devices is that older style COM equipment will not allow both **com1** and **com3** to use interrupts at the same time, nor will it allow both **com2** and **com4** to use interrupts at the same time. If you use a port in polled mode, you will get better performance using one of the newer FIFO parts, such as the 16550A.

To convert from using a polled to an interrupt driven device, edit file **/etc/ttys** and then type the command:

```
kill quit 1
```

For details, see the Lexicon entry for **ttys**.

f (Flow control) A device with hardware flow control. Here, signal CTS must be active for the driver to send data out the port, and signal RTS will be set active by the driver whenever it is ready for input. Some high-speed modems, and some serial printers, are capable of using these conventions. If your equipment does not support RTS/CTS handshaking, there is no benefit to using this option.

Due to limitations in the design of the ports, you can enable interrupts on either COM1 or COM3 (or on COM2 or COM4), but not both. If you wish to use both ports simultaneously, one must be run in polled mode. For example, if you wish to open all four serial lines, you can open two of the lines in interrupt mode: you can open either COM1 or COM3 in interrupt mode, and you can open either COM2 or COM4 in interrupt mode. The other two lines must be opened in polled mode.

Opening a device in polled mode consumes many CPU cycles, based upon the speed of the highest baud rate requested. For example, on a 20 MHz 80386-based machine, polling at 9600-baud was found to consume about 15% of the CPU time. As only one device can use the interrupt line at any given time, the best approach is to make the high-speed line of the pair interrupt driven and open the low-speed or less-frequently used line in polled mode. However, if you enable a polled line for logins, the port is open and will be polled as long as the port remains open (enabled). Thus, even if a port is not in use, the fact that it has a **getty** on it consumes CPU cycles. As a rule of thumb, try to open a port in interrupt mode. If you cannot, use the polled version.

If you intend to use a modem on your serial port, you must insure that the DCD signal from the modem actually *follows* the state of carrier detect. Some modems allow the user to "strap" or set the DCD signal so that it is always asserted (true). This incorrect setup will cause COHERENT to think that the modem is "connected" to a remote modem, even when there is no such connection.

There are eight possible configurations, and eight valid suffixes. In the example of the port whose base name is **com1**, the configurations would be found in the directory **/dev** as **/dev/com1l**, **/dev/com1r**, **/dev/com1pl**, **/dev/com1pr**, **/dev/com1fl**, **/dev/com1fr**, **/dev/com1fpl**, and **/dev/com1fpr**.

Driver Configuration

asy is usually configured — and proper names are created in directory **/dev** — when you install COHERENT. The following explains how to configure **asy**, in case you must modify the original installation.

To configure **asy**, do the following:

1. Type the following command to become the superuser **root**:

   ```
   su root
   ```

2. Change to directory **/etc/conf**.

3. Execute script **asy/mkdev**. This script walks you through the process of describing your serial ports to COHERENT.

4. When you have successfully completed **asy/mkdev**, type the command:

   ```
   bin/idmkcoh -o cohtest
   ```

   This generates a new kernel, called **cohtest**, which incorporates the changes you described when you ran **asy/mkdev**.

5. Boot your new kernel. If you do not know how to do this, read the Lexicon entry **booting**.

Editing /etc/default/async

The first step in reconfiguring **asy** is to edit **/etc/default/async**. This file holds the description of how the **asy** driver is to be configured.

asy ignores blank lines and lines that begin with a pound sign '#'; you can use them as comments if you wish. Each port that is not in a group must have a line beginning with the letter 'P', followed by seven numbers:

- The hexadecimal base address for the port.
- The IRQ number, in decimal, used by the port (use zero if no interrupt line is needed).
- The hexadecimal value used for control lines OUT1 and OUT2 when the port is open. Permissible values are 0, 4, 8, and C. Use 4 if OUT1 must be asserted, 8 if OUT2 must be asserted, and C if both signals are needed. The most common value needed in this field is 8.
- One if the port needs exclusive use of its interrupt line (true for conventional COM1/COM4 equipment), zero otherwise.
- Default baud rate for the port.
- Channel number for the port (0-31).
- A flag to indicate if modem-status interrupts are to be disabled for this board: one if they are to be disabled, zero if they are not.

The last field is required because some chips are defective and lock up the system if modem status interrupts are enabled. This flag protects you against such problems, but at the price of disabling hardware flow control.

Many multiport boards support a separate I/O address that can be read to determine which port requires service. Each group of up to 16 ports must have a line beginning with the letter 'G', followed by a separate line describing each port in the group. There are four different group types:

1. Bits in the status port are one when the corresponding port needs service, zero otherwise. (Sealevel, Comtrol, Star Gate, Connect Tech, Boca Research.)
2. Bits in the status port are zero when the corresponding port needs service, one otherwise. (Arnet.)
3. The low three bits in the status port give the slot number on the card for the port needing service. (GTEK.)
4. The low four bits in the status port give the slot number on the card for the port reading service. If no port needs service, the status port contains hexadecimal value FF. (Digiboard.)

The 'G' line requires the following fields. All are in decimal, except as noted:

- The hexadecimal address for the group-status port.
- The IRQ number used by the group. Use zero if no interrupt line is needed.
- The hexadecimal value used for control lines OUT1 and OUT2 when the port is open (usually eight).
- The type number of the group — one, two, or three, as described above.
- The number of ports in the group, 1 through 16.
- A flag to indicate if modem-status interrupts are to be disabled for this board: one if they are to be disabled, zero if they are not.

Each group line is followed by a separate 'M' line for each member of the group. Fields required on the 'M' line (in decimal, except as noted) are:

- The hexadecimal base address for the port.
- Default baud rate for the port.
- The slot number of the port within the group 0 through 7. For group types 1 and 2, slot 0 corresponds to the least-order bit in the status port, slot 7 to the highest order bit.
- Channel number for the port (0-31).

The following gives the **async** file for a system with standard **COM1** through **COM4** ports as channels 0 through 3, a Comtrol Hostess 550/16 as channels 4 through 19, and finally an Arnet Multiport as channels 20 through 27.

```
# /etc/default/async spec for standard com1-com4
#Record formats:
#P         Port       Irq        OUT[12]    Excl       Speed      Channel     No MS int
#G         Port       Irq        OUT[12]    Type       Number-of-Slots        No MS int
#M         Port       Speed      Slot       Channel

# com1/2/3/4
P          3f8        4          8          1          9600       0    0
P          2f8        3          8          1          9600       1    0
P          3e8        4          8          1          9600       2    0
P          2e8        3          8          1          9600       3    0

# Hostess 550 16 - two groups of 8 ports, using irq 12
G          507        12         8          1          8          0
M          500        9600       0          4
M          508        9600       1          5
M          510        9600       2          6
M          518        9600       3          7
M          520        9600       4          8
M          528        9600       5          9
M          530        9600       6          10
M          538        9600       7          11

G          547        12         8          1          8          0
M          540        9600       0          12
M          548        9600       1          13
M          550        9600       2          14
M          558        9600       3          15
M          560        9600       4          16
M          568        9600       5          17
M          570        9600       6          18
M          578        9600       7          19

# Arnet Multiport - one group of 8 ports, using irq 7
G          272        7          0          2          8          0
M          280        9600       0          20
M          288        9600       1          21
M          290        9600       2          22
M          298        9600       3          23
M          2A0        9600       4          24
M          2A8        9600       5          25
M          2B0        9600       6          26
M          2B8        9600       7          27
```

You should look at the version of **/etc/default/async** that is shipped with COHERENT for examples of all **async** features, including those described above. This file includes sample configurations for every board that Mark Williams Company had available for testing.

Building a New Kernel

Now that you have described how you want **asy** to be configured, the next step is to build a new kernel. Log in as the superuser **root** and execute the following commands:

```
cd /etc/conf
asy/mkdev
bin/idmkcoh -o /kernel_name
```

where *kernel_name* is the new kernel that includes the **asy** driver. To run this new kernel, simply reboot your machine.

See Also

asymkdev, device drivers, RS-232

Notes

If your system loses characters while transferring files on 4800-bps or higher-speed lines, *we strongly urge you to replace your existing 8250- or 16450-based UARTs with those based upon the 16550A design, such as the National Semiconductor NS16550AFN.* These newer UARTs are pin-compatible with the older UARTs. COHERENT automatically senses and enables them when it boots.

LEXICON

asymkdev — Command

Create nodes for asynchronous devices
/conf/asymkdev [-u] [*async_file* **[***outfile***]]**

The command **asymkdev** reads *async_file*, questions the user about her system, and writes a shell script into *outfile*. When run, the script creates the proper nodes (up to 256 of them) for the asynchronous devices in **/dev**.

If you name no *async_file*, **asymkdev /dev/default/async**. If you name no *outfile*, it writes its script into **asy_mknod**.

asymkdev asks about each asynchronous channel for which a port is configured. It asks for the basic device name (e.g., **asy00** or **com0**), and then asks which of the eight possible port configurations will be used. The options are:

l or **r** Local or remote.

i or **p** Interrupt-driven or polled.

f or **n** RTS-CTS flow control or no hardware flow control.

For details of what the options mean, see the Lexicon article for the device driver **asy**.

Suffix letters "rlipnf" respectively indicate remote, local, interrupt, polled, no-flow, and flow-control configurations, as explained in the Lexicon article for **asy**.

For each question, type the value that applies or press **<Enter>**, to select the default displayed in brackets. The option **-u** suppresses prompts.

See Also

asy, asypatch, commands

Notes

Only the superuser **root** can run this command.

asypatch — Command

Patch a kernel file for an asynchronous configuration
/conf/asypatch [-v] <*kernel_name***> <***async_file*

The command **asypatch** patches a kernel file for the asynchronous configuration specified in *async_file*. The format of *async_file* is described in the Lexicon article for the device driver **asy**.

See Also

asy, asymkdev, commands

at — Device Driver

Drivers for hard-disk partitions

/dev/at* are the COHERENT system's AT devices for the hard-disk's partitions. Each device is assigned major-device number 11, and may be accessed as a block- or character-special device.

at handles two drives with up to four partitions each:

- Minor devices 0 through 3 identify the partitions on drive 0.
- Minor devices 4 through 7 identify the partitions on drive 1.
- Minor device 128 allows access to all of drive 0.
- Minor device 129 allows access to all of drive 1.

To modify the offsets and sizes of the partitions, use the command **fdisk** on the special device for each drive (minor devices 128 and 129).

To access a disk partition through COHERENT, directory **/dev** must contain a device file that has the appropriate type, major and minor device numbers, and permissions. To create a special file for this device, invoke the command **mknod** as follows:

```
/etc/mknod /dev/at0a b 11   0 ; : drive 0, partition 0
/etc/mknod /dev/at0b b 11   1 ; : drive 0, partition 1
/etc/mknod /dev/at0c b 11   2 ; : drive 0, partition 2
/etc/mknod /dev/at0d b 11   3 ; : drive 0, partition 3
/etc/mknod /dev/at0x b 11 128 ; : drive 0, partition table
```

Drive Characteristics

When processing BIOS I/O requests prior to booting COHERENT, many IDE drives use translation-mode drive parameters: number of heads, cylinders, and sectors per track. These numbers are called "translation-mode" parameters because they do not reflect true physical drive geometry. The translation-mode parameters used by the BIOS code present on your host adapter can be obtained using the command **info** from within the tertiary-boot routine **tboot**. (For details on **info**, see the Lexicon entry for **tboot**.) It is often necessary to patch the **at** driver with BIOS values of translation-mode parameters in order to boot COHERENT on IDE hard drives. In COHERENT versions 3.1.0 and later, drive parameters are stored in table **atparm** in the driver. For the first hard drive, number of cylinders is a short (two-byte) value at **atparm+0**, number of heads is a single byte at **atparm+2**, and number of sectors per track is a single byte at **atparm+14**. For the second hard drive, number of cylinders is a short value at **atparm+16**, number of heads is a single byte at **atparm+18**, and number of sectors per track is a single byte at **atparm+30**. For example, if **testcoh** is a kernel linked with the **at** driver and you want to patch it for a second hard drive with 829 cylinders, 10 heads, and 26 sectors per track, you can do:

```
/conf/patch testcoh atparm+16=829:s atparm+18=10:c atparm+30=26:c
```

To read the characteristics of a hard disk once the **at** driver is running, use the call to **ioctl** of the following form:

```
#include <sys/hdioctl.h>
hdparm_t hdparms;
      . . .
ioctl(fd, HDGETA, (char *)&hdparms);
```

where *fd* is a file descriptor for the hard-disk device and *hdparms* receives the disk characteristics.

Non-Standard and Unsupported Types of Drives

Prior releases of the the COHERENT **at** hard-disk driver would not support disk drives whose geometry was not supported by the BIOS disk parameter tables. COHERENT adds support for these drives during installation by "patching" the disk parameters into the bootstrap and the **/coherent** image on the hard disk.

Files

/dev/at* — Block-special files
/dev/rat* — Character-special files

See Also

device drivers, fdisk, hai, ideinfo

Notes

The driver **at** offers two varieties of polling: normal and alternate. Normal, as its name implies, is used with most varieties of AT controllers. Alternate polling is for Perstor controllers and some other older equipment. Using the wrong type of polling causes frequent controller timeouts and bad-track messages.

at also lets you specify the number of seconds to wait for a response from the drive after an I/O request. The default value is six. Some IDE drives occasionally become unresponsive for long intervals (several seconds) while control firmware makes adjustments to drive operation.

To set either the type of polling or the default waiting period, **su** to the superuser **root**; then **cd** to directory **/etc/conf** and run the script **at/mkdev**. This script will walk you through describing your AT controller to COHERENT. Once you have run this script, execute the command

```
/etc/conf/bin/idmkcoh -o cohtest
```

to create a test kernel that incorporates your changes; then reboot your system and invoke the new kernel, as described in the Lexicon entry **booting**. Note that the changes you make to the driver will not be seen by your COHERENT system until you boot the new kernel.

The **at** driver lets you have up to two AT hard disks on your system. Note, however, that in our experience, it is very difficult to combine different brands of AT hard disks and have both run successfully. This is especially true with Conner drives, which apparently do not cooperate with other IDE drives as master and slave. *Caveat utilitor.*

at — Command

Execute commands at given time
at [**-v**] [**-c** *command*] *time* [[*day*] *week*] [*file*]
at [**-v**] [**-c** *command*] *time month day* [*file*]

at executes commands at a given time in the future.

If the **-c** option is used, **at** executes the following *command*. If *file* is named, **at** reads the commands from it. If neither is given, **at** reads the standard input for commands.

If *time* is a one-digit or two-digit number, **at** interprets it as specifying an hour. If *time* is a three-digit or four-digit number, **at** interprets it as specifying an hour and minutes. If *time* is followed by **a**, **p**, **n**, or **m**, **at** assumes **AM**, **PM**, **noon**, or **midnight**, respectively; otherwise, it assumes that *time* indicates a 24-hour clock. Note that you should *not* type a colon ':' in the time string.

For example, the command

```
at -c "time | msg henry" 1450
```

set the **time** command to be executed at 2:50 PM, and pipe **time**'s output to the **msg** command, which will pass it to the terminal of user **henry**. The argument to the **-c** option had to be enclosed in quotation marks because it contains spaces and special characters; if this were not done, **at** would not be able to tell when the argument ended, and so would generate an error message. If you wish to pass information to a user's terminal with the **at** command, you must tell **at** to whom to send the information. The command

```
at 250p commandfile
```

sets the file **commandfile** to be read and executed at 2:50 PM. It is *not* necessary to use the file's full path name. Also, if the suffix **p** were not appended to the time, the file would be set to be read at 2:50 AM.

The time set in **at**'s command line is *not* the exact time that the command is executed. Rather, the daemon **cron** periodically executes the command **/usr/lib/atrun** to see if any commands have been scheduled commands to be executed at or before the present time. The frequency with which **cron** executes **atrun** determines the "granularity" of **at** execution times. To change when **cron** executes **atrun**, edit file **/usr/spool/cron/crontabs/root**. For example, the entry

```
0,5,10,15,20,25,30,35,40,45,50,55 * * * *   /usr/lib/atrun
```

sets **/usr/lib/atrun** to be executed every five minutes. Thus, the **at** command that is set, for example, to 2:53 PM will actually be executed at 2:55 PM. **atrun** executes specified commands when it discovers that the given time is past; therefore, **at** commands are executed even if the system is down at the specified time or if the system's time is changed.

The **at** command has two forms, as shown above. In the first form, the option *day* names a day of the week (lower case, spelled out). If **week** is specified, **at** interprets the given *time* and *day* as meaning that time and day the following week. For example, the command

```
at -c "time | msg henry" 1450 friday week
```

executes **time** and sends its output to **henry**'s terminal one week from Friday at 2:50 PM.

In the second form given above, *month* specifies a month name (lower case, spelled out) and the number *day* specifies a day of the month. For example, the command

```
at 1450 july 4 commandfile
```

set the file **commandfile** to be read at 2:50 PM on July 4.

If the **-v** flag is given, **at** prints the time when the commands will be executed, giving you enough information to plan for the execution of the command. For example, if it is now August 13, 1990, at 2:30 PM, and you type the command

```
at -v -c "/usr/games/fortune | msg henry" 1435
```

at will reply:

```
Tue Aug 13 14:35:00
```

indicating that the command will be executed five minutes from now. However, if you type

```
at -v -c "/usr/games/fortune | msg henry" 1435 august 10
```

at will reply

```
Sun Aug 10 14:35:00 1991
```

which indicates that on Sunday, August 10 of next year, at 2:35 PM, the COHERENT system will print a **fortune** onto your terminal.

Should you create such a long-distance **at** file by accident, you can correct the error by simply deleting the file that encodes it from the directory **/usr/spool/at**. The file will be named after the time that it is set to execute, plus a unique two-character suffix, should more than one command be scheduled to run at the same time. For example, the file for the above command would be named **9108101435.aa**.

Finally, note that the current working directory, exported shell variables, file creation mask, user id, and group id are restored when the given command is executed.

Example

The following example invokes the command **wall** at 11 P.M. to confirm that the **at** command is working properly:

```
at -c "echo 'testing to see if cron is working' | /etc/wall" 2300
```

Files

/bin/pwd — To find current directory
/usr/lib/atrun — Execute scheduled commands
/usr/spool/at — Scheduled activity directory
/usr/spool/at/ *yymmddhhmm.xx* — Commands scheduled at given time

See Also

at, commands, cron

atan() — Mathematics Function (libm)

Calculate inverse tangent
#include <math.h>
double atan(*arg***) double** *arg***;**

atan() calculates the inverse tangent of *arg*, which may be any real number. The result will be in the range [-π/2, π/2].

Example

For an example of this function, see the entry for **acos()**.

See Also

errno, libm, tan(),
ANSI Standard, §7.5.2.3
POSIX Standard, §8.1

atan2() — Mathematics Function (libm)

Calculate inverse tangent
#include <math.h>
double atan2(*num***,** *den***) double** *num***,** *den***;**

atan2() calculates the inverse tangent of the quotient of its arguments *num*/*den*. *num* and *den* may be any real numbers. The result will be in the range [-π, π]. The sign of the result will have the same sign as *num*, and the cosine will have the same sign as *den*.

Example

For an example of this function, see the entry for **hypot()**.

See Also

errno, libm
ANSI Standard, §7.5.2.4
POSIX Standard, §8.1

ATclock — Command

Read or set the AT realtime clock
/etc/ATclock *[yy[mm[dd[hh[mm[.ss]]]]]]*

ATclock reads or sets your system's "hardware" time, which is stored in your system's CMOS. This clock should contain the current standard time for your locale.

With no argument, the command **ATclock** reads the hardware clock and returns a string in the format expected by the command **date**. With an argument, it sets the hardware clock to the given date. For example, to set your hardware clock to October 24, 1994, at 9:30 PM, use the command:

```
/etc/ATclock 9410242130
```

ATclock also lets you reset the time incrementally: that is, you can reset only the year; the year and month; the year, month, and day; and so on down to the second.

Note that if you use **ATclock** to reset your hardware clock, you *must* reset it to the standard time in your locale, even if daylight-savings time happens to be in effect when you reset the clock. If you do not, COHERENT's commands that set the local time on your system (e.g., the command **date**) will be off by one hour when daylight-savings time is in effect.

The system startup file **/etc/brc** typically contains a command of the form

```
date -s `/etc/ATclock`
```

to reset the time properly when the COHERENT system starts up.

See Also

brc, clock, CMOS, commands, date

atexit() — General Function (libc)

Register a function to be called when the program exits
#include <stdlib.h>
int atexit(void (*function***)**
void (**function***)();**

atexit() registers one or more functions to be called when the program exits. These registered functions can, for example, perform clean-up beyond what is ordinarily performed when a program exits. **atexit()** can register up to 32 functions.

function points to the function to be called. A registered function takes no arguments and returns nothing.

The functions that **atexit()** registers are called when the program exits normally, i.e., when the function **exit()** is called or when **main()** returns. They are called in *reverse* order of registration.

atexit() returns zero if *function* could be registered, and a value other than zero if it could not.

Example

This example registers two functions to be executed upon exiting: one displays a message, and the other waits for the user to press a key before terminating.

```
#include <stdlib.h>
#include <stdio.h>

void
lastgasp()
{
	fprintf(stderr, "Type return to continue");
}

void
get1()
{
	getchar();
}
```

```
main()
{
      /* set up get1() as last exit routine */
      atexit(get1);
      /* set up lastgasp() as exit routine */
      atexit(lastgasp);

      /* exit, which invokes exit routines */
      exit(EXIT_SUCCESS);
}
```

See Also

exit(), libc
ANSI Standard, §7.10.4.2

atof() — General Function (libc)

Convert ASCII strings to floating point
#include <stdlib.h>
double atof(*string***) char** * *string***;**

atof converts *string* into the binary representation of a double-precision floating point number. *string* must be the ASCII representation of a floating-point number. It can contain a leading sign, any number of decimal digits, and a decimal point. It can be terminated with an exponent, which consists of the letter **'e'** or **'E'** followed by an optional leading sign and any number of decimal digits. For example,

```
123e-2
```

is a string that can be converted by **atof()**.

atof() ignores leading blanks and tabs; it stops scanning when it encounters any unrecognized character.

Example

For an example of this function, see the entry for **acos()**.

See Also

atoi(), atol(), float, libc, long, printf(), scanf(), stdlib.h
ANSI Standard, §7.10.1.1
POSIX Standard, §8.1

Notes

atof does not check to see if the value represented by *string* fits into a **double**. It returns zero if you hand it a string that it cannot interpret.

atoi() — General Function (libc)

Convert ASCII strings to integers
#include <stdlib.h>
int atoi(*string***) char** **string***;**

atoi() converts *string* into the binary representation of an integer. *string* may contain a leading sign and any number of decimal digits. **atoi()** ignores leading blanks and tabs; it stops scanning when it encounters any non-numeral other than the leading sign, and returns the resulting **int**.

Example

The following demonstrates **atoi()**. It takes a string typed at the terminal, turns it into an integer, then prints that integer on the screen. To exit, type **<ctrl-C>**.

```
#include <stdlib.h>

main()
{
      extern char *gets();
      extern int atoi();
      char string[64];
```

```
      for(;;) {
            printf("Enter numeric string: ");
            if(gets(string))
                  printf("%d\n", atoi(string));
            else
                  break;
      }
}
```

See Also

libc
ANSI Standard, §7.10.1.2
POSIX Standard, §8.1

Notes

atoi does not check to see if the number represented by *string* fits into an **int**. It returns zero if you hand it a string that it cannot interpret.

atol() — General Function (libc)

Convert ASCII strings to long integers
#include <stdlib.h>
long atol(*string***) char** ***string*;**

atol() converts the argument *string* to a binary representation of a **long**. *string* may contain a leading sign (but no trailing sign) and any number of decimal digits. **atol()** ignores leading blanks and tabs; it stops scanning when it encounters any non-numeral other than the leading sign, and returns the resulting **long**.

Example

```
#include <stdlib.h>

main()
{
      extern char *gets();
      extern long atol();
      char string[64];

      for(;;) {
            printf("Enter numeric string: ");
            if(gets(string))
                  printf("%ld\n", atol(string));
            else
                  break;
      }
}
```

See Also

atof(), atoi(), float, libc, long, printf(), scanf(), stdlib.h
ANSI Standard, §7.10.1.3
POSIX Standard, §8.1

Notes

No overflow checks are performed. **atol()** returns zero if it receives a string it cannot interpret.

atrun — System Administration

Execute commands at a preset time

atrun is a program that executes programs at a time set by the command **at**.

When user **steve** types

```
at 1230 /v/steve/lunchtime
```

the command **at** creates a shell script in directory **/usr/spool/at** that contains the information needed to execute command **/v/steve/lunchtime** at a later time — in this instance, 12:30 PM. The spooled file sits in **/usr/spool/at** until **/usr/lib/atrun** sees that the specified time has been reached. **atrun** then executes the

spooled command and removes it from **/usr/spool/at**.

atrun is not a daemon; that is, it is invoked by another program, does its work, and exits. Thus, it is typically run periodically from an entry in the **cron** file owned by the superuser **root**.

See Also

Administering COHERENT, at

Notes

Although **atrun** technically is a command, it is never invoked by a user.

auto — C Keyword

Note an automatic variable

auto is an abbreviation for an *automatic variable*. This is a variable that applies only to the function that invokes it, and vanishes when the functions exits. The word **auto** is a C keyword, and must not be used to name any function, macro, or variable.

See Also

C keywords, extern, static, storage class,
ANSI Standard, §6.5.1

awk — Command

Pattern-scanning language
awk [*POSIX or GNU style options* **] -f** *program-file* **[--]** *file ...*
awk [*POSIX or GNU style options* **] [--]** *program-text file ...*

awk is a general-purpose language designed for processing input data. Its features allow you to write programs that scan for patterns, produce reports, and filter relevant information from a mass of input data. It acts upon the contents of each *program-file*, or the standard input if no *-program-file* is specified.

You can specify the program either as an argument (usually enclosed in quotation marks to prevent interpretation by the shell **sh**) or in the form **-f** *program-file*. If no **-f** option appears, the first non-option argument is the **awk** program.

awk views its input as a sequence of records, each consisting of zero or more fields. By default, newlines separate records and white space (spaces or tabs) separates fields. The option **-F***c* changes the input field separator characters to the characters in the string *c*. An **awk** program can also change the field and record separators. The program can access the values of each field and the entire record through built-in variables.

For details on the construction of **awk** programs, consult the tutorial to **awk** that appears in this manual. Briefly, an **awk** program consists of one or more lines, each containing a *pattern* or an *action,* or both. A *pattern* determines whether **awk** performs the associated *action.* It may consist of regular expressions, line ranges, boolean combinations of variables, and beginning and end of input-text predicates. If no *pattern* is specified, **awk** executes the *action* (the pattern matches by default).

An *action* is enclosed in braces. The syntax of actions is C-like, and consists of simple and compound statements constructed from constants (numbers, strings), input fields, built-in and user-defined variables, and built-in functions. If an *action* is missing, **awk** prints the entire input record (line).

Unlike **lex** or **yacc**, **awk** does not compile programs into an executable image, but interprets them directly. Thus, **awk** is ideal for quickly-implemented, one-shot efforts.

Examples

The following examples illustrate the economy of expression of **awk** programs.

The first example reads the standad input, and echoes all lines containing the string "COHERENT":

```
awk '/COHERENT/'
```

To exit, type **<ctrl-D>/**

The built-in variable **NR** is the number of the current input record. The next example reads the standard input, and prints the number of records you typed after you exit (again, by typing **<ctrl-D>**):

```
awk 'END { print NR }'
```

The built-in variable **$3** gives the value of the third field of the current record. The last example sums the third field from each record you type on the standard input, and prints the total when you exit:

```
awk '{ sum += $3 }
     END   { print sum }'
```

See Also

commands, gawk, lex, sed, yacc
Introduction to the awk Language, tutorial.

Notes

Beginning with release 4.2.14 of COHERENT, **awk** has been replaced by **gawk**, the GNU implementation of this language. For details on this implementation of the **awk** language, see the Lexicon entry for **gawk**.

backups — Technical Information

Strategies for backing up COHERENT

This entry describes how to backup files — that is, how to copy one or more selected files onto floppy disks. You should do this regularly to provide yourself with a spare copy of valuable files should your system suffer a catastrophe.

The strategy you adopt for backups will vary quite a bit, depending upon the medium onto which you back up your files: tapes or floppy disks. Floppy disks are inexpensive, but their limited capacity means that you have to plan carefully. Tapes are simpler to use than floppy disks, but are more expensive. The following sections describe first the strategies for backing up onto floppy disks; and then for backing up onto tapes.

Backing up Onto Floppy Disks

There are two general strategies for backing up files onto floppy disks:

- Use the command **tar** to create archives of files on a floppy disk. This is fine for archiving a limited set of files on an irregular basis.
- The other strategy uses the command **gnucpio** to implement a system of regular dumps. This strategy is preferred for systems that daily amass data of importance for a real-world job, such as running a business or managing a research project.

You should always have a procedure of backups for your system. Which strategy you use depends on how you are using your system. The following sections describe how to implement each strategy of backups. Note that COHERENT includes a version of the UNIX utility **dump** for the sake of compatibility with older versions of UNIX and COHERENT; however, **dump** is obsolete, should not be used, and will not be described here.

Please note that the following descriptions assume that you are using a 5.25-inch, high-density floppy disks set in drive 0 (drive A). For a list of available floppy-disk devices, see the Lexicon entry for **floppy disks**.

The following describes how to use **tar** to back up onto floppy disks.

The first step is to prepare floppy disks to receive files. Insert a 5.25-inch floppy disk into drive 0, and then type the following command:

```
/etc/fdformat -v /dev/rfha0
```

The command **fdformat** formats the floppy disk, verifying that no media defects exist. You must perform this task of formatting a floppy disk before you use it the first time.

The next step is to create an archive of the files you wish to back up. Use the portable archive command **tar** to collect a mass of files into an archive on the floppy disks. For example, to archive all files in directory **source**, use the following command:

```
tar cvf /dev/rfha0 source
```

The options **cvf** tell **tar** to create an archive, run in verbose mode, and write the archive onto the device or into the file named in the next argument. **/dev/rfha0** names the floppy device onto which you wish to write the archive. Finally, **source** is the directory whose files you wish to back up.

To perform a listing of the contents of the newly created archive, type

```
tar tvf /dev/rfha0
```

The options **tvf** tell **tar** to list the contents of the archive, run in verbose mode, and read the archive from the device or file named in the next argument.

To extract several files from the archive, enter a command of the form

```
tar xvf /dev/rfha0 source/myfile 'source/*.c'
```

The options **xvf** tell **tar** to extract or unarchive the specified files, run in verbose mode, and read the archive from the device or file named in the next argument. Note that the second file argument contains a "wildcard" character and thus must be quoted to prevent expansion by the shell.

For more information on how to use **tar**, see its entry in the Lexicon.

The following describes how to back up using **gnucpio**.

The COHERENT utility **gnucpio** performs mass dumps and restores of files using a universally recognized file format.

In this example, dumps are performed monthly, weekly, and daily. You should prepare at least three sets of floppy disks for the monthly saves, giving you three months of full backup. You will use the floppy disks in rotation, with the oldest always used next.

Once a month, you should dump the entire system.

Once a week, you should dump information in the system that is new or has been changed since the end of the previous week. You will need five sets of floppy disks, because some months have five weekends in them.

Finally, every day you should save information that has changed that day. For these dumps, you will need five sets of floppy disks: one for each working day. You may need extras in case of weekend work.

Label each set of disks carefully as *monthly*, *weekly*, or *daily*. Label the daily floppy disks "Monday" through "Friday", the weekly floppy disks "Week 1" through "Week 5", and the monthly floppy disks "Month 1" through "Month 3". When you perform the dump, write the date on the label.

The following gives a step-by-step description of how to use **gnucpio** to back up files. The next samples are given with the suggestion that your system has only one 5.25-inch floppy-disk drive.

1. Log into the system as the superuser **root**.
2. If you have not yet done so, use the command **fdformat** to format a set of floppy disks, as shown above. With high-density, 5.25-inch floppy disks, a rule of thumb is to prepare one floppy disk for each megabyte of data to be dumped.
3. If other users are logged into the system, use the command **wall** to request that they log off. For example:

   ```
   /etc/wall
   Please log off.
   Time for file dump.
   <ctrl-D>
   ```

4. Be sure that all users are logged off the system by typing the command:

   ```
   who
   ```

   This command names all users who are still on the system.

   If they have not logged off in a few minutes, send another message. Repeat the process until **who** shows no users except yourself.

5. When all other users have logged off, execute the command **shutdown** as described in its Lexicon entry.
6. Run the script **mount.all** to mount all of your file systems. Then, run the COHERENT command **fsck** on each file system to check its integrity.
7. If this is the last workday of the month, perform a *monthly* dump, to back up the entire system. Insert the first volume of the correct monthly dump floppy disk into the floppy drive, after adding today's date to the label, and type the commands:

   ```
   cd /
   find . -print | gnucpio -ocF /dev/rfha0
   ```

   Option **-F** tells **gnucpio** to write everything to the raw, 2400-block, floppy-disk device **/dev/rfha0**.

Note that if you want to split your dump across different media (i.e., write the first volume onto tape and the second onto a floppy disk), you should not use the option **-F**; **gnucpio** will write its output to the standard output, and you can use the shell operator '>' to redirect that to the device **/dev/rfha0**. If you do not use **-F**, **gnucpio** will ask you, after it finishes writing a volume, for the name of the device into which it should redirect the next volume of output.

As more floppies are needed, **gnucpio** will ask you to insert them. Be sure to label each floppy disk with its volume number.

8. If this is the last work day of the week, but not the last workday of the month, perform a *weekly* dump. Prepare the correct weekly dump floppy disks, add today's date to the label, insert the first floppy disk, and type the command:

```
cd /
find . -newer cpio.weekly -print | gnucpio -ocF /dev/rfha0
touch cpio.weekly
```

This will dump all files that are younger than file **cpios.weekly**.

9. If this is neither the last workday of the month nor the last workday of the week, you will perform a *daily* dump. Prepare the daily dump floppy disk with today's day of the week, add today's date to the label, insert the first floppy disk into the drive, and type the command:

```
cd /
find . -newer cpio.daily -print | gnucpio -ocF /dev/rfha0
touch cpio.daily
```

This will dump files that are younger than file **cpio.daily**.

10. Type **sync** to ensure that all buffers are flushed.

11. When you are finished dumping data, type the command **/etc/reboot** to return your system to multi-user mode.

For more information on how to use **gnucpio** and **find**, see their respective entries in the Lexicon.

If you wish, you can back up only limited portions of your system. To do so, just name in your **find** command the directories you wish to back up. For example, to back up everything in your home directory and in **/usr/lib**, use the following command:

```
find $HOME /usr/lib -type f -newer cpio.daily -print | gnucpio -ocF /dev/rfha0
touch cpio.daily
```

When you determine the backup strategy you wish to use, you should save the appropriate commands into a script, to ensure that backups are run correctly every time.

The following describes how to restore files from floppy disks.

If you find that a file has been inadvertently destroyed, you can restore the information to disk from backup floppy disks.

To restore information from backups created with **gnucpio** or **tar**, you must first determine the date and time that the file was last known to have been modified. From this date, determine on which set of disks the file was last correctly dumped. Find the set of floppy disks labeled with that date, and insert into the floppy-disk drive the first one in the set. For example, if you wish to restore the file **myfile**, from a **gnucpio** archive, use the command:

```
gnucpio -icdvF /dev/rfha0 myfile
```

To retrieve **myfile** from a **tar** archive, use the command:

```
tar xvf /dev/rfha0 myfile
```

Both of these commands assume that the disks are high-density, 5.25-inch floppies in drive 0 (drive A). See the Lexicon article **floppy disk** for a table that shows which COHERENT device is associated with which size and density of disk, and which disk drive. You may have to insert more than one disk from the set of backups until you find the one that holds the file you want.

Backing up Onto Tapes

The strategy for backing up onto tape resembles that for floppy disks, with the exception that in many instances the tape medium is larger than the device being backed up. This makes it worth your while to back up the entire device every time you do a back up, rather than perform incremental backups. The reason for this is simple: the

fewer tapes over which you have spread your backups, the lower the risk that one will fail.

To back up an entire partition, do the following:

1. Pop a tape into your tape device. Make sure the tape is appropriately labeled.
2. Log in as the superuser **root**, and type the following command:

   ```
   /etc/shutdown single 0
   ```

   This returns your system to single-user mode immediately.
3. Use the command **gtar** to back up your partition, as follows:

   ```
   gtar -cvzf /dev/tape directory
   ```

   tape identifies the tape device onto which the backup will be written, and *directory* identifies the file system to back up. For example, tape device **/dev/rStp2** is a SCSI tape device that has SCSI identifier 2 and performs autorewinding. For a list of recognized tape devices, see the article for **tape**. in the Lexicon.

   Please note two points about *directory*. First, do *not* use the absolute path name when specifying a directory: that is, use **usr**, *not* **/usr**. **gtar** strips the leading '/' in any event, but it's always best to use relative path names whenever possible. Second, in single-user mode only the root file system is mounted by default; therefore, if the file system you wish to back up resides on its own partition, you must mount that file system by hand before you begin to back it up.

   Note that the **z** option to the **gtar** command tells **gtar** to use **gzip** to compress the files automatically. File compression is a good idea: because fewer bits are being written to the tape, the backup will go faster; and because less tape is used, the risk of a tape failure is lessened.
3. When **gtar** has finished writing to the tape, wait until the tape finishes rewinding; then remove it from its drive and put it in a safe place (i.e., away from magnets and children). Then type **<ctrl-D>** to return your system to multi-user mode.

That's all there is to it. To restore information from the tape, put the tape into the drive and use the **gtar** command to fetch the file you want. For example, to restore file **/v/fwb/myfile.c** from a SCSI tape drive that has SCSI identifier 2, use the following command:

```
gtar -xvzf /dev/rStp2 "v/fwb/myfile.c"
```

Note that the file will be written into a subdirectory of your current directory. For example, if your current directory is **/v/fwb**, then **myfile.c** will be restored into a file with the path name **/v/fwb/v/fwb/myfile.c**. This may be a little inconvenient, but is not nearly as inconvenient as having to create **myfile.c** by hand.

An Example of Using Floppy Tape

This section gives examples of how to use QIC-40/QIC-80 ("floppy tape") to write archives to floppy tape, and read them back. It uses the commands **tape**, which manipulates the tape device; and **gtar**, which writes archives onto the physical tape, and reads them back.

Suppose you have a directory named **dir1**, which contains files you want to backup. To back up all files in that directory onto a tape, insert a tape cartridge into the drive, then type:

```
gtar -cvf /dev/ft dir1
```

To verify that the contents of the tape match the original files, run **gtar** again in verification ("diff") mode:

```
gtar -df /dev/ft
```

We strongly urge you to verify tapes after they have been written, especially with floppy-tape devices. If a tape fails this test, throw it away and build a new archive; otherwise, you may receive a nasty surprise when you try to restore a file from that tape. Do not be surprised if an otherwise sound tape fails after time: a tape does wear out after a number of uses.

To later extract the files from the tape, use

```
gtar -xf /dev/ft
```

To use data compression, the preceding commands can be used with the addition of **gtar**'s option **-z**, as follows:

```
gtar -czvf /dev/ft dir1
gtar -dzf /dev/ft
gtar -xzf /dev/ft
```

To backup only selected files to tape, you could do the following:

```
find dir -type f -print | sort > Files
```

then manually edit the file **Files** so it contains only the names of the files you want to back up. Then use the command:

```
gtar -cv -T Files -f /dev/ft
```

The previous examples used **/dev/ft**, the device node that calls for the tape to be rewound when the device is closed. This is convenient if you are putting only one archive onto tape. To concatenate multiple archives on a single cartridge, use the no-rewind-on-close device. For example, suppose you have a second directory, **dir2**, and you want to back it up on the same tape, after an archive of **dir1**. The following commands accomplish this:

```
gtar -cvf /dev/nft dir1
gtar -cvf /dev/nft dir2
```

After each archive is written, the tape remains positioned at the end of the archive. To verify the contents of both archives, do the following:

```
# this command rewinds the tape:
tape rewind
# this command displays the contents of the first archive:
gtar -tvf /dev/nft
# this command displays the contents of the second archive:
gtar -tvf /dev/nft
```

If you make a note of the locations of archives as they are written, you can retrieve them later without having to read the preceding archives. For example:

```
# rewind the tape:
tape rewind
# write "dir1" archive at start of tape:
gtar -cvf /dev/nft dir1
# find current position of the tape:
tape tell
```

The command **tape tell** returns a string of the form:

```
Tape Is at Byte Offset 102400
```

Continuing:

```
# write "dir2" archive after "dir1":
gtar -cvf /dev/nft dir2
# read the current position:
tape tell
```

The second instance of **tape tell** returns a string of the form:

```
Tape Is at Byte Offset 235520
```

That is, it shows that the tape has advanced after the second archive was written onto it. At this point, the cartridge is removed, then reinserted into the tape drive at a later date:

```
tape seek 102400
gtar -tvf /dev/tape
```

The command **tape seek** moves the tape to the byte position **102400**, i.e., the end of the first archive. This command assumes that you jotted down the position displayed by the command **tape tell** executed earlier. The command **gtar** then displays the contents of the second archive.

See Also

Administering COHERENT, gnucpio, gtar, tape

bad — Command

Maintain list of bad blocks
bad [-acdl] *device* **[** *block ...* **]**

A hard disk or floppy disk may have bad blocks on it: a "bad block" is a portion of disk that is flawed, and so cannot reliably be read or written. It is the unusual disk that is free of bad blocks.

COHERENT keeps a list of bad blocks so it can avoid using them. The command **bad** maintains this bad-block list for the given *device*, which must be a block-special file. **bad** recognizes the following command-line options:

a Add each given *block* to the bad-block list
c Clear the bad-block list
d Delete each given *block* from the bad-block list
l List all blocks on the bad-block list

Note that **bad** merely adds a block to the list of bad blocks, or removes a block from that list. It does not deallocate any i-node associated with a block when adding it to the bad-block list. You should run the command **icheck** with the option **-s** immediately after **bad** to correct the problem, or run the command **fsck**. After you modify the list of bad blocks, you must reboot your system to force the kernel to use this modified list.

The file system on *device* should be unmounted if possible. You must have appropriate permissions for *device* before you can invoke **bad**. For many file systems, only the superuser may use **bad** to change the bad-block list. Use the command **badscan** to create a prototype file of bad blocks.

When the command **mkfs** creates a file system, the prototype specification may include a list of bad blocks for the new file system.

See Also

badscan, commands, icheck, mkfs

badscan — Command

Build bad block list
/etc/badscan [-v] [-o *proto* **] [-b** *boot* **]** *device size*
/etc/badscan [-v] [-o *proto* **] [-b** *boot* **]** *device xdevice*

badscan scans a floppy disk or a partition of the hard disk for bad blocks. It writes onto the standard output a prototype file that lists all bad blocks on the disk.

badscan recognizes the following options:

-v Print an estimate of time needed to finish examining the device.

-o *proto* Redirect output into file *proto*.

-b *boot* Insert a given *boot* into the proto file as the bootstrap. The default is **/conf/boot**.

device names the special device to scan.

The command line for **badscan** comes in two forms, as shown at the top of this article. The first version is for a floppy disk; *size* gives the size of the device, in blocks. The second version is for a hard-disk partition; *xdevice* specifies devices **/dev/at0x** or **/dev/at1x**, which hold the partition-table information for the disk in question. **badscan** reads the data from the boot block of the drive to find the size of the **device**.

Examples

The first example uses **badscan** to find all bad blocks on a high-density, 3.5-inch floppy disk in drive 1 (i.e., drive B), and writes its output into file **proto**:

```
/etc/badscan -v -o proto /dev/rfva1 2880
```

See the article **floppy disks** for a table that gives the device name and number of sectors to be found on the various types of floppy disk that COHERENT recognizes.

The second example uses **badscan** to prepare a list of bad blocks for partition 2 on hard-drive 0, which is an IDE drive accessed via COHERENT's **at** driver. Again, the output is written into file **proto**:

```
/etc/badscan -v -o /conf/proto.at0c /dev/rat0c /dev/at0x
```

See Also

at, bad, commands, floppy disks, mkfs

Notes

Because SCSI hard-disk drives maintain their own map of bad blocks, **badscan** is not required for SCSI drives. However, we recommend that you use it on removeable-media SCSI drives.

banner — Command

Print large letters
banner [*argument ...*]

banner prints large (seven-character by five-character) letters on the standard output. Each *argument* produces one large text output line. If there is no *argument*, each line from the standard input produces one line of large-text output.

See Also

commands, libmisc, lpr, pr

basename — Command

Strip path information from a file name
basename *file* [*suffix*]

basename strips its argument *file* of any leading directory prefixes. If the result contains the optional *suffix*, **basename** also strips it. **basename** prints the result on the standard output.

For example, the command

```
basename /usr/fred/source.c
```

returns

```
source.c
```

basename is most useful when it is used with other shell commands. For example, the command

```
for i in *.c
do
      cp $i `basename $i .c`.backup
done
```

copies every file that has the suffix **.c** into an identically named file that has the suffix **.backup**.

See Also

commands, ksh, sh

bc — Command

Interactive calculator with arbitrary precision
bc [-l] [*file ...*]

bc is a language that performs calculations on numbers with an arbitrary number of digits. **bc** is most commonly used as an interactive calculator, where the user types arithmetic expressions in a syntax reminiscent of C. If you invoke **bc** with no *file* argument, it reads the standard input. For example:

```
Input             Output
(1000+23)*42      42966
k = 2^10
16 * k            16384
2 ^ 100           1267650600228229401496703205376
```

You can invoke **bc** with one or more *file* arguments. After **bc** reads each *file*, it reads the standard input. This provides a convenient way to read programs that are stored in files. COHERENT includes a library of mathematical functions for **bc**; to use it, invoke **bc** with its option **-l**.

The following summarizes briefly the facilities provided by **bc**. More information is available in the tutorial to **bc** that is included with this manual.

The delimiters '**/***' and '***/**' enclose comments. Names of variables or functions consist of a lower-case letter followed by any number of letters or digits. (Names cannot begin with an upper-case letter because numbers with

a base greater than ten may need upper-case letters for their notation.) The three built-in variables **obase**, **ibase**, and **scale** represent, respectively, the number base for printing numbers (default, ten), the number base for reading numbers (default, ten), and the number of digits after the decimal (radix) point (default, zero). Variables may be simple variables or arrays, and need not be pre-declared, with the exception of variables internal to functions. Some examples of variables and array elements are **x25**, **array[10]**, and **number**.

Numbers are any string of digits, and may have one decimal point. Digits are taken from the ordinary digits (0-9) and then the upper-case letters (A-F), in that order.

Certain names are reserved for use as key words. The key words recognized by **bc** include the following:

if, for, do, while
Test conditions and define loops, with syntax identical to C

break, continue
Alter control flow within **for** and **while** loops.

quit Tell **bc** to exit immediately

define *function* **(***arg, ..., arg***)**
Define a **bc** function by a compound statement, as in C.

auto *var, ..., var*
Define variables that are local to a function, rather than having global scope.

return (*value***)**
Return a value from a function.

scale (*value***)**
Return the number of digits to the right of the decimal point in *value*.

sqrt (*value***)**
Return the square root of *value*

length (*value***)**
Return the number of decimal digits in *value*.

bc recognizes the following operators:

```
+       -       *       /       %       ^       ++
--      =       +=      -=      *=      /=      %=
^=      ==      !=      <       <=      >       >=
```

These operators are similar to those in C, with the exception of **^** and **^=**, which are exponentiation operators. Expressions can be grouped with parentheses. Statements are separated with semicolons or newlines, and may be grouped with braces into compound statements.

bc prints the value of any statement that is an expression but is not an assignment.

As in the editor **ed**, an '!' at the beginning of a line causes that line to be sent as a command to the COHERENT shell **sh**.

The library **lib.b** holds code written in **bc** for the following mathematical variables and functions:

atan(*z*) Arctangent of *z*
cos(*z*) Cosine of *z*
exp(*z*) Exponential function of *z*
j(*n,z*) *n*th order Bessel function of *z*
ln(*z*) Natural logarithm of *z*
pi Value of pi to 100 digits
sin(*z*) Sine of *z*

If you invoke **bc** with its option **-l**, it reads **lib.b** and thus makes the above functions and constants available to you.

Examples

The first example calculates the factorial of its positive integer argument by recursion.

```
/*
 * Factorial function implemented by recursion.
 */
define fact(n) {
      if (n <= 1) return (n);
      return (n * fact(n-1));
}
```

The second example also calculates the factorial of its positive integer argument, this time by iteration.

```
/*
 * Factorial function implemented by iteration.
 */
define fact(n) {
      auto result;

      result = 1;
      for (i=1; i<=n; i++) result *= i;
      return (result);
}
```

Files

/usr/lib/lib.b — Source code for the library

See Also

commands, conv, dc, libmp
bc Desk Calculator Language, tutorial

Notes

Line numbers do not accompany error messages in source files.

bc performs integer calculations with arbitrary precision, limited only by the memory available. However, the results of some calculations on numbers with fractional parts depends on the specified **scale**; see the tutorial for details.

bcmp() — String Function (libc)

Compare two chunks of memory
int bcmp (*source***,** *destination***,** *count***)**
VOID **source***, **destination***; **size_t** *count***;**

Function **bcmp()** compares the first *count* bytes of data at address *source* with the first *count* bytes of data at address *destination*. It returns the offset of the first character where *source* and *destination* differ; if they do not differ, it returns zero.

See Also

libsocket, memcmp()

Notes

This function is included for compatibility with Berkeley socket code. It is equivalent to the standard C function **memcmp()**, except that its first two arguments are reversed.

bcopy() — String Function (libc)

Berkeley function to copy memory
void bcopy(*source***,** *destination***,** *amount***)**
char **source***, **destination***;
int *size***;**

Function **bcopy()** copies *size* bytes of data from address *source* to address *destination*. *destination* must point to enough allocated memory to hold *size* bytes of data, or problems will result.

See Also

libc, memcpy(),

Notes

Please note the arguments of **bcopy()** are the opposite of those used by **memcpy()**. This function is included solely for compatibility with existing code; users are encouraged to use the standard function **memcpy()** instead.

bind() — Sockets Function (libsocket)

Bind a name to a socket
#include <sys/types.h>
#include <sys/socket.h>
int bind (*socket***,** *name***,** *namelen***)**
int *socket***,** *namelen***; struct sockaddr** **name***;*

Function **bind()** binds a name to an unnamed socket.

When function **socket()** creates a socket, that socket exists but has no name. **bind()** creates a special file, assigns it a name, and binds that file to a socket. Thereafter, the socket can be accessed by reading or writing the file.

socket is a file descriptor that identifies the socket in question. It must have been returned by a call to **socket()**. *name* points to the full path name of the file to which *socket* is to be bound. The calling process must unlink *name* when it no longer needs it. *namelen* gives the number of bytes in the path name name to which *name* points. Under COHERENT, no element of *name* can exceed 14 characters (not including separating '/' characters).

If all goes well, **bind()** returns zero. If something goes wrong, **bind()** returns -1 and sets **errno** to an appropriate value. The following lists the errors that can occur, by the value to which **bind()** sets **errno**:

EBADF *socket* is somehow not a valid descriptor.

ENOTSOCK
socket is not a socket.

EADDRNOTAVAIL
name is not available from the local machine.

EADDRINUSE
name is already bound to another socket.

EINVAL
socket is already bound to a name.

EACCES
The memory to which *name* points is protected and the user lacks permission to access it.

EFAULT
name points to an illegal address.

ENOTDIR
The path name to which *name* points contains an element that is not a directory.

EINVAL
The path name to which *name* points contains a character with the high-order bit set.

ENOENT
A prefix component of the path name does not exist.

EIO An I/O error occurred while creating the directory entry for *name* or allocating its inode.

EROFS *name* would reside on a read-only file system.

EISDIR
name points to an empty path name.

Example

For an example of this function, see the Lexicon entry for **libsocket**.

See Also

connect(), getsockname(), libsocket, listen(), socket()

bit — Definition

bit is an abbreviation for "binary digit". It is the basic unit of data processing. A bit can have a value of either zero or one. Bits can be concatenated to form bytes.

A bit can be used either as a placeholder to construct a number with an absolute value, or as a flag whose value has a particular meaning under specially defined circumstances. In the former use, a string of bits builds an integer. In the latter use, a string of bits forms a **map**, in which each bit has a meaning other than its numeric value.

See Also

bit map, byte, nybble, Programming COHERENT,
ANSI Standard, §1.6

bit-fields — Definition

A *bit-field* is a member of a structure or **union** that is defined to be a cluster of bits. It provides a way to represent data compactly. For example, in the following structure

```
struct example {
      int member1;
      long member2;
      unsigned int member3 :5;
}
```

member3 is declared to be a bit-field that consists of five bits. A colon ':' precedes the integral constant that indicates the *width*, or the number of bits in the bit-field. Also, the bit-field declarator must include a type, which must be one of **int**, **signed int**, or **unsigned int**.

A bit-field that is not given a name may not be accessed. Such an object is useful as "padding" within an object so that it conforms to a template designed elsewhere.

A bit-field that is unnamed and has a length of zero can be used to force adjacent bit-fields into separate objects. For example, in the following structure

```
struct example {
      int member1;
      int member2 :5;
      int :0;
      int member3 :5;
};
```

the zero-length bit-field forces **member2** and **member3** to be written into separate objects.

Finally, it is illegal to take the address of a bit-field.

See Also

bit, bit map, byte, Programming COHERENT,
ANSI Standard, §3.5.2.1

Notes

Because bit-fields have many implementation-specific properties, they are not considered to be highly portable. Bit-fields use minimal amounts of storage, but the amount of computation needed to manipulate and access them may negate this benefit. Bit-fields must be kept in integral-sized objects because many machines cannot directly access a quantity of storage smaller than a "word" (a word is generally used to store an **int**).

bit_count() — Sockets Function (libsocket)

Count bits in a bit-mask
int bit_count (*mask***)**
unsigned *mask***;**

The function **bit_count()** counts and returns the bits in *bitmask* that have been turned on.

See Also

libsocket

bit map — Definition

A **bit map** is a string of bits in which each bit has a symbolic, rather than numeric, value.

See Also

bit, byte, Programming COHERENT,

Notes

C permits the manipulation of bits within a byte through the use of bit-field routines. These generate code rather than calls to routines. Bit fields are generally less efficient than masking because they always generate masking and shifting.

block — Technical Information

A *block* is a mass of data that is read at one time. Blocks are different lengths under different operating systems; COHERENT defines a block as being **BSIZE** bytes long.

Information is read in blocks from block-special devices, such as the hard disk or floppy disks. This is done to increase the speed with which data are read from these devices; reading characters one at a time, such as is done with character-special devices such as terminals or modems, would be too slow.

See Also

Using COHERENT,
ANSI Standard, §3.6.2

boot — Driver

Boot block for hard-disk partition/nine-sector diskette

Several different programs are used to load COHERENT from a floppy or hard disk into memory. This process is called *bootstrapping* (from the old expression about pulling one's self up by one's bootstraps) or *booting* for short. The program used depends upon whether one is loading COHERENT from a hard-disk partition, from a 5.25-inch floppy disk, or from a 3.5-inch floppy disk. All of these programs are installed onto your computer during normal installation.

mboot is the master boot program. This is code that resides in the first 446 bytes of the first sector on the hard disk. Because this sector also contains the partition table for the hard disk, **mboot** is normally written to the hard drive only during installation and only by the **fdisk** utility.

boot, **boot.fha**, and **boot.fva** are variations of the same program. **boot** occupies the first sector of any bootable hard-drive partition. **boot.fha** occupies the first sector of a 5.25-inch, high-density floppy disk. **boot.fva** occupies the first sector of a 3.5-inch, high-density floppy disk.

boot is normally copied to the root partition automatically during installation by a command such as:

```
/bin/dd if=/conf/boot of=/dev/at0a count=1
```

In another example, the following commands format and create a file system on a high-density, 5.25-inch floppy disk:

```
/etc/fdformat -v /dev/fha0
/etc/mkfs /dev/fha0 2400
/bin/cp /conf/boot.fha /dev/fha0
```

When invoked, **boot** loads for the tertiary boot program **tboot**. This, in turn, searches the root directory '/' for file **autoboot**, which is the COHERENT kernel. If it finds this kernel, **boot** loads and invokes it. Otherwise, it gives the prompt ?, and you must type the name of the operating-system kernel to load (typically, "coherent"). If **boot** cannot find the requested kernel or if an error occurs, **boot** does not print an error message, but re-prompts with '?'.

Files

/conf/boot — Boot for AT partitions
/conf/boot.at — Boot for AT partitions (linked to **/conf/boot**)
/conf/boot.atx — AT master boot (linked to **/conf/mboot**)
/conf/boot.f9a — Boot for single-density, nine-sector, 5.25-inch floppy disk
/conf/boot.fha — Boot for 15-sector, 5.25-inch floppy disk
/conf/boot.fqa — Boot for quad-density, nine-sector, 3.5-inch floppy disk

/conf/boot.fva — Boot for 18-sector, 3.5-inch floppy disk
/conf/mboot — AT master boot

See Also

device drivers, fdisk, mboot, mkfs, tboot

boot.fha — Device Driver

Boot block for floppy disk

To be bootable, a COHERENT file system must contain a boot block (either **boot** or **boot.fha**). In addition, all hard disks must contain the master boot block **mboot** or an equivalent.

boot.fha is a boot block for a hard disk partition or a 15-sector floppy. It must be installed as the first sector of the partition or diskette, as follows:

```
/etc/fdformat -a /dev/fha0
/etc/badscan -v -o proto1 /dev/fha0 2400
/etc/mkfs /dev/fha0 proto1
rm proto1
cp /conf/boot.fha /dev/fha0
```

boot.fha searches its root directory '/' for file **autoboot**. If it finds this kernel, **boot.fha** loads and runs it. Otherwise, it gives the prompt ?, to which the user must type the name of the operating-system kernel to load (typically, **coherent**). If **boot.fha** cannot find the requested kernel or if an error occurs, **boot.fha** repeats the prompt and the user must type another name.

Files

/conf/boot.fha — Partition or 15-sector 96tpi floppy boot block

See Also

badscan, boot, device drivers, fdisk, mboot, mkfs

booting — Technical Information

How booting works

Booting is the method by which COHERENT is loaded from a hard disk or floppy disk and set into action. The term comes from the old expression about pulling one's self up by one's bootstraps.

This article discusses the events that take place while booting the COHERENT system. You do not need to read this article to know how to boot COHERENT, as all booting details are handled by COHERENT automatically. However, if you are interested in the details, or want to tailor the system to your needs, it will help.

Two I/O devices are involved in booting. The first device is called the *boot* device; it contains the program necessary to invoke the COHERENT system and start it running. The second device is called the *root* device; it contains the root file system after the system is running. In most cases, these two devices are the same physical device.

Initial Startup

When you boot from a hard disk, your computer's BIOS loads the master boot from the first sector of your hard disk into memory. The master boot then loads the secondary boot from the first sector of your boot partition. When you boot from a floppy disk, however, the BIOS loads the secondary boot directly.

This program, called the *bootstrap* or *secondary boot*, is very small (only 512 bytes), so it cannot do very much. Therefore, its main purpose is to read in a larger, more complex program called the *tertiary boot*, or **/tboot**. It is **/tboot** that actually performs the work of loading the COHERENT system into memory.

If the secondary boot does not find a file called **/tboot**, it print a '?' to prompt for the boot image you want it to load. This indicates a severe error because it means that the tertiary boot can not be found.

If the secondary boot finds **/tboot**, it loads it into memory and lets it take over booting. The first thing **/tboot** does is search for a file called **/autoboot** in the root directory of the device being booted. If **/tboot** finds **/autoboot**, it first pauses for five seconds, so you can abort the process and boot another kernel if you wish. If you do not abort booting within five seconds, **/tboot** then loads **/autoboot** into memory and runs it. If, however, **/tboot** cannot find **/autoboot**, it prompts you to type the name of the COHERENT image to boot, usually **/coherent**. You can type the commands **dir** or **ls** if you do not remember the name of the image you wish to boot. Note that **/autoboot** is usually a link to **/coherent**.

If you need to find the file name of the kernel you are now running (usually **/coherent**), use the program **fifo()**, which is kept in library **libmisc**. See the Lexicon entry **libmisc** for details.

After it loads the system image **/autoboot** from the root device, the system initializes all devices, as well as starting the *idle* process and program **/etc/init**. The idle process uses any leftover computer time.

init controls the operation of the system from this point on. It first executes the command **/etc/brc** (i.e., "boot run commands"), which can run commands like **fsck**. **brc** can request a reboot, remain in single-user mode, or enter multi-user mode automatically. **init** then calls the *shell* to handle commands from the system console. The shell responds by prompting with **#**, and expects regular commands. At this point, the system is in *single-user mode*, which means that no other users can log in to the system. The shell is running in superuser mode and only the console's user is logged in.

At this point, you can enter commands to the system in a normal fashion. One difference from normal, multi-user operation is that the system is in single-user mode, to allow special processing to take place before other users log in. Being in single-user mode gives you the opportunity to run **fsck** to check the file system and perform other administrative tasks before other users log into the system.

When administrative activities are finished, you should type **<ctrl-D>**. This terminates single-user operation; **init** then opens the system to other users.

The file **/etc/rc** contains shell commands that the system executes just before making the system available to other users. This file typically includes commands to delete temporary files and mount standard devices. It also performs any installation-specific commands you require. As system administrator, you maintain this file. You must be sure that it is properly updated and never removed.

One command that must be included in **/etc/rc** is **/etc/update**, which periodically calls **sync()** to update buffered data to the disk.

init also maintains the file **/etc/utmp**, which notes users' login and logout.

Features of the Master Bootstrap

The COHERENT master bootstrap allows you to boot different operating systems from different partitions of any hard drive. It is more powerful than similar programs of other operating systems, and we strongly recommend that you use it. If you do not use the MWC bootstrap, you may have to use floppy disks to boot up MS-DOS and COHERENT. If you have two hard drives and you are placing COHERENT on the second drive, you must use the MWC bootstrap.

The bootstrap can be configured in three ways:

1. No active partition. With this configuration, you have the greatest degree of flexibility. When you boot your system, the following prompt appears on the screen:

   ```
   Select Partition 0-7
   ```

   This means that you must press the number key that corresponds to the partition that holds the root partition of the operating system you wish to boot. (For example, if you wish to boot COHERENT and its root partition is on partition 2, then press the '2' key in response to this prompt.) If you have one hard drive, only partitions 0 through 3 are relevant to you. The bootstrap waits indefinitely until you tell it what to boot.

2. COHERENT is active partition. Under this configuration, the system will automatically boot COHERENT unless you press the number key that represents the root partition of another operating system (e.g., MS-DOS) while the A-drive light is on.

3. MS-DOS (or another operating system) is active partition. Under this configuration, the system automatically boots MS-DOS unless you hit the number key that represents the root partition of another operating system (e.g., COHERENT) while the A-drive light is on.

Under some hardware configurations, particularly faster 80386 machines, having an active partition can cause difficulties when you try to boot a non-active partition. It often is difficult to press the appropriate number key at the right time, and the right time itself can vary. For this reason, the default setting of the master bootstrap is to have no active partition. If at any time you wish to reconfigure the bootstrap, you need only to run the **fdisk** utility under COHERENT and access option 1 (Change active partition) of the option menu. Make the desired change and then save the updated partition table.

Files Used During Startup

The following files are used when the system is in single-user mode:

/etc/drvld.all Device tables to load. This usually names the keyboard driver to use, should you be using the keyboard driver **vtnkb**.

/etc/init Initiate a process on each terminal line, call login when appropriate.

/etc/brc Shell commands for booting.

/etc/checklist List of partitions for **fsck** to check.

/bin/sh Bourne shell.

/bin/ksh Korn shell.

The following files are needed after the system has entered multi-user mode:

/bin/login This file holds the program that controls logging in.

/etc/getty This file holds the executable program that permits a user to log in on a port.

/etc/logmsg This file holds the text of the login prompt.

/etc/motd This file holds the message of the day.

/etc/mount.all Shell script to mount partitions.

/etc/rc This file holds a series of shell commands that **coherent** executes when it enters multi-user startup.

/etc/ttys This file is holds information about terminals. Its contents are read by **getty** to ensure that it sets the port to the correct baud rate and terminal type.

/etc/utmp This file holds information about who is logged in right now. It is read by the command **who**.

Building a Bootable Floppy Disk

Building a bootable floppy disk for COHERENT requires a few more steps than are required to build a bootable floppy for MS-DOS. The task is not particularly painful, it simply requires a little more attention to detail.

The following details the steps required to build a version of COHERENT that can be booted off a floppy disk. Note that the following describes an extremely minimal configuration, which can be used only in single-user mode.

1. Format the Floppy Disk

To begin, format the floppy disk with the command **/etc/fdformat**. After you format the floppy disk, use the command **/etc/mkfs** command to write a blank file system onto it.

2. Write a Bootstrap to the Floppy Disk

To make the floppy disk bootable, you must copy a special program, or *bootstrap*, into the first sector (or *boot block*) of the floppy disk. (This is the same program that is called the *secondary boot* in the above sections.) If a floppy disk is to be bootable, a set of instructions must be present in the boot block that tell the system the name of the kernel — that is, the file on the floppy disk to be loaded and executed.

To write the bootstrap to the floppy disk, you must copy it to the *device* that the floppy disk is in. This ensures that the bootstrap is copied to the first sector, or boot block, of the floppy disk. For example, to copy the bootstrap for a 1.2-megabyte floppy disk in floppy drive 0 (or A), type the command:

```
cp /conf/boot.fha /dev/fha0
```

To copy the bootstrap for a 1.44-megabyte floppy disk to floppy drive 0, type the command:

```
cp /conf/boot.fva /dev/fva0
```

3. Write Tertiary Boot

After you have copied the boot sector, you must mount the floppy device and copy **/tboot** to it. To mount a 1.44-megabyte floppy disk to floppy drive 0, type the command:

```
/etc/mount /dev/fva0 /f0
```

Copy **/tboot** with the following command:

```
cp /tboot /f0
```

Warning: *Never* mount the floppy disk before you copy the bootstrap to it!

See the Lexicon article on **floppy disks** for the table of floppy disk devices to use with the above commands.

4. Copy the Necessary Files

Once the bootstrap is properly written to the floppy disk, it is now time to copy the essential files to it. Type the following commands:

```
mkdir /f0/etc
mkdir /f0/dev
mkdir /f0/bin
mkdir /f0/tmp
cp /tboot /coherent /coherent.sym /f0
cp /etc/init /etc/brc /etc/profile /f0/etc
cp /dev/* /f0/dev
cp /bin/sh /bin/sync /f0/bin
```

If you are using either of the loadable keyboard drivers **nkt** or **vtnkb**, also execute the following commands:

```
mkdir /f0/drv
mkdir /f0/conf
mkdir /f0/conf/kbd
cp /etc/drvld.all /f0/etc
cp /drv/* /f0/drv
cp /conf/kbd/* /f0/conf/kbd
```

The above files will let you run COHERENT in single-user mode, which is all that you need when you boot COHERENT from a floppy disk.

Note that the files **/etc/brc** and **/etc/drvld.all** are scripts that you must modify to suit your needs. The file **/etc/brc** is a key file in the booting process, so be prepared to modify its contents. The significance of this will be reviewed in depth in the next section.

Warning: After you have finished copying files to the floppy disk, execute the command **umount** to unmount the floppy disk. If you do not, the files will be damaged or lost!

5. The Boot Sequence, Modifications To Make the Disk Work

When the computer system powers up and accesses the floppy disk, it reads the boot sector of the disk, which in turn looks for the file **/tboot** and executes it. **/tboot** looks for the kernel named **/autoboot**, reads it, and executes it. If **/tboot** cannot find **/autoboot**, it prompts you to type the name of the kernel to boot.

The kernel loads and invokes **/etc/init** which, in part, looks for and executes the statements in **/etc/brc**, which, in turn, typically loads loadable drivers and runs **/etc/fsck** to check the file systems. If you wish to run **fsck** on the floppy disk, you must copy it from the hard drive.

What is truly important is the *exit status* of **/etc/brc**. If its exit status is not zero, the system remains in single-user mode. If its exit status is zero, the system attempts to enter multiuser mode.

The above-listed files are the bare minimum for a single-user floppy disk. To build a floppy disk with the minimum files needed, your **/etc/brc** file should look like this:

```
/etc/drvld.all
exit 1
```

This forces an exit status of one and causes COHERENT to spawn a single-user shell, **/bin/sh**.

From the shell prompt, you can do whatever you wish, but you are limited to the commands and functions copied to the floppy disk.

/etc/brc is not the only file that may need modification. The kernel (**/coherent** or **/autoboot**) must have the values **rootdev** and **pipedev** patched for the floppy disk's major and minor device numbers. This patching can be done with the commands **/bin/db** or **/conf/patch**.

To patch the kernel on the floppy disk mounted on **/f0** for a 5.25-inch, high-density disk as the root and pipe device, type:

```
/conf/patch /f0/coherent rootdev=makedev\(4,14\)
/conf/patch /f0/coherent pipedev=makedev\(4,14\)
```

For a 3.5-inch, high-density disk, type:

```
/conf/patch /f0/coherent rootdev=makedev\(4,15\)
/conf/patch /f0/coherent pipedev=makedev\(4,15\)
```

Finally, note that when you boot your floppy disk, the disk must *not* be write protected. This is because COHERENT must be able to write temporary files into directory **/tmp**; if it cannot do so, booting will fail.

Uses of a Bootable Floppy Disk

A bootable floppy disk can be a lifesaver should something occur to corrupt the COHERENT file system on the hard drive. A properly prepared floppy can be used to recover a damaged file system by running **/etc/fsck**. You can also use it to copy files from the hard drive should you decide to re-install COHERENT on the hard drive.

Multiuser-mode floppy disks can also be built for the fun of seeing such a system run from a floppy disk. The capacity of such a system is limited, of course, but it can be done.

See Also

Administering COHERENT, boot, libmisc, tboot

Notes

Some users have attempted to use Norton Utilities or similar tools to rearrange the partition table, only to find that COHERENT no longer boots. That is because the kernel has embedded within it the name of the partition on which it and its root file system live. By using Norton Utilities to shuffle the partition table, the kernel will no longer be able to find any of the files or utilities it needs to boot your system.

If you still wish to shuffle your disk's partition table, be sure to change the name of the root device within the kernel *before* you change the partition table.

boottime — System Administration

File that holds time system was last booted

/etc/boottime is an empty file maintained by the **init** process and the **date** command. The modification time of **boottime**, as displayed by the command **ls -l**, is the time that the system was last booted. You can read the time shown by **boottime** with **ls -l**, or with the system calls **stat** or **fstat**.

Files

/etc/boottime

See Also

Administering COHERENT, date, init, mount

Notes

Commands that depend upon **/etc/boottime** may malfunction if the system's date is not set correctly. For instance, the **mount** command depends on the relative modification times of **/etc/boottime** and **/etc/mtab** to detect whether the mount table has been invalidated by a system boot. If the date is set sufficiently far into the past, the mount table may appear to be valid when in fact it is not.

brc — System Administration

Perform maintenance chores, single-user mode
/etc/brc

The shell script **/etc/brc** is executed by the **init** process when the COHERENT system enters single-user mode. The commands in **brc** do such things as set system clock, set the local time zone, and call **fsck** to scan and (if necessary) fix the file systems that that are named in the file **/etc/checklist**.

See Also

Administering COHERENT, checklist, init, rc

Notes

The default message consists of the bell character **<ctrl-G>** plus the text **Going multiuser**. If the bell annoys you, simply delete the **<ctrl-G>** from this string.

break — Command

Exit from shell construct
break [*n*]

The command **break** is used with the shell to control how it performs loops. It is analogous to the **break** keyword in C.

When it is used without an argument, **break** forces the shell to exit from the innermost current **for**, **until**, or **while** loop. If used with an argument, **break** exits from *n* levels of **for**, **until**, or **while** loops.

The shell executes **break** directly.

See Also

commands, continue, for, ksh, sh, until, while

break — C Keyword

Exit from loop or switch statement

break is a C statement that causes an immediate exit from a **switch** sequence, or from a **while**, **for**, or **do** loop.

See Also

C keywords
ANSI Standard, §6.6.6.3

brk() — System Call (libc)

Change size of data area
#include <unistd.h>
brk(*addr***)**
char **addr***;*

The *break* is the lowest address above the data area of a process. **brk()** sets the break to the given *addr*, possibly rounding up by some machine-dependent factor.

See Also

libc, malloc(), sbrk(), unistd.h If the request succeeds, **brk()** returns zero. Otherwise, it returns -1 and sets **errno** to **ENOMEM**.

bsearch() — General Function (libc)

Search an array
#include <stdlib.h>
char ***bsearch(***key, array, number, size, comparison***)**
char ***key*, ****array*;**
size_t *number*, *size*;**
int (*comparison***)();**

bsearch() searches a sorted array for a given item. *item* points to the object sought. *array* points to the base of the array; it has *number* elements, each of which is *size* bytes long. Its elements must be sorted into ascending order before it is searched by **bsearch()**.

comparison points to the function that compares array elements. *comparison* must return zero if its arguments match, a number greater than zero if the element pointed to by *arg1* is greater than the element pointed to by *arg2*, and a number less than zero if the element pointed to by *arg1* is less than the element pointed to by *arg2*.

bsearch() returns a pointer to the array element that matches *item*. If no element matches *item*, then **bsearch()** returns NULL. If more than one element within *array* matches *item*, which element is matched is unspecified.

Example

This example uses **bsearch()** to translate English into "bureaucrat-ese".

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct syntab {
      char *english, *bureaucratic;
} cdtab[] = {
/* The left column is in alphabetical order */

      "affect",        "impact",
      "after",         "subsequent to",
      "broke",         "revenue shortfall",
      "building",      "physical facility",
      "call",          "refer to as",
      "do",            "implement",

      "false",         "inoperative",
      "finish",        "finalize",
      "first",         "initial",
      "full",          "in-depth",
      "help",          "facilitate",

      "idiot",         "elected representative",
      "kill",          "terminate with extreme prejudice",
      "lie",           "inoperative statement",
      "order",         "prioritize",
      "talk",          "interpersonal communication",
      "then",          "at that point in time",
      "use",           "utilize"
};

int
comparator(key, item)
char *key;
struct syntab *item;
{
      return(strcmp(key, item->english));
}

main()
{
      struct syntab *ans;
      char buf[80];

      for(;;) {
            printf("Enter an English word: ");
            fflush(stdout);

            if(gets(buf) || !strcmp(buf, "quit") == NULL)
                  break;

            if((ans = bsearch(buf, (char *)cdtab,
                        sizeof(cdtab)/ sizeof(struct syntab),
                        sizeof(struct syntab),
                        comparator)) == NULL)
                  printf("%s not found\n");

            else
                  printf("Don't say \"%s\"; say \"%s\"!\n",
                        ans->english, ans->bureaucratic);
      }

      return(EXIT_SUCCESS);
}
```

See Also

libc, qsort(), stdlib.h
ANSI Standard, §7.10.6.2
POSIX Standard, §8.1

LEXICON

Notes

The name *bsearch* implies that this function performs a binary search. A binary search looks at the midpoint of the array, and compares it with the element being sought. If that element matches, then the work is done. If it does not, then **bsearch()** checks the midpoint of either the upper half of the array or of the lower half, depending upon whether the midpoint of the array is larger or smaller than the item being sought. **bsearch()** bisects smaller and smaller regions of the array until it either finds a match or can bisect no further.

It is important that the input *array* be sorted, or **bsearch()** will not function correctly.

buf.h — Header File

Buffer header
#include <sys/buf.h>

Header file **<sys/buf.h>** defines the structure used to hold buffers.

See Also

header files

buffer — Definition

A *buffer* is a portion of memory set aside to hold data read from or to be written to another process or device. Often, although not always, this involves setting aside a portion of the arena with **malloc** or its related functions.

Buffering, and problems therewith, are encountered most often when using the standard input and output (STDIO) routines. Many operating systems (including COHERENT) automatically place data from a peripheral device into a buffer. Buffers normally can be cleared with **fflush()**, by pressing the carriage return key on routines that perform input, or by sending a newline character on routines that perform output. The function **fclose()**, which closes a file stream, flushes all buffers associated with that stream. **exit()** calls **fclose()**.

Combining unbuffered and buffered I/O functions on the same file or device within one program will produce results that are at best unpredictable.

Example

The following example demonstrates what does and does not happen when you use **fflush()** with the output buffer.

```
#include <stdio.h>
main()
{
      extern char *malloc();
      char *buffer;

      /* use malloc() to create a 120-char buffer */
      if ((buffer = malloc(120)) == NULL) {
            /* if malloc() fails, bail out */
            fprintf(stderr, "malloc failed\n");
            exit(1);
      }

      printf("Type your name:  ");
      fflush(stdout);
      gets(buffer);
      printf("Your name is %s\n", buffer);
}
```

See Also

arena, array, close(), exit(), fflush(), malloc(), Programming COHERENT, stdio.h

build — Command

Install COHERENT onto a hard disk
/etc/build

build installs COHERENT onto your hard disk. COHERENT runs **/etc/build** to install itself onto your hard disk. After installation, you should never have an occasion to run **build**.

See Also

commands

builtin — Command

Execute a command as a built-in command
builtin *command* **[** *arg ...* **]**

The command **ksh** is used by the Korn shell **ksh** to establish *command* as a built-in command.

See Also

commands, ksh

byte — Definition

A **byte** is a group of bits that encodes a character or a small-integer quantity. A byte, like a dollar, consists of eight bits.

The ANSI Standard defines the data type **char** as being equal to one byte. It defines all other data types as multiples of **char**.

See Also

bit, char, data formats, nybble, Programming COHERENT
ANSI Standard, §1.6

byte ordering — Definition

Machine-dependent ordering of bytes

Byte ordering is the order in which a given machine stores successive bytes of a multibyte data item. Different machines order bytes differently.

The following example displays a few simple examples of byte ordering:

```
main()
{
      union
      {
            char b[4];
            int i[2];
            long l;
      } u;
      u.l = 0x12345678L;

      printf("%x %x %x %x\n",
            u.b[0], u.b[1], u.b[2], u.b[3]);
      printf("%x %x\n", u.i[0], u.i[1]);
      printf("%lx\n", u.l);
}
```

When run on "big-endian" machines, such as the M68000 or the Z8000, the program gives the following results:

```
12 34 56 78
1234 5678
12345678
```

As you can see, the order of bytes and words from low to high memory is the same as is represented on the screen.

However, when this program is run on "little-endian" machines, such as the PDP-11, you see these results:

```
34 12 78 56
1234 5678
12345678
```

As you can see, the PDP-11 inverts the order of bytes within words in memory.

Finally, when the program is run on the i8086 you see these results:

```
78 56 34 12
5678 1234
12345678
```

The i8086 inverts both words and long words.

See Also

C language, canon.h, data formats, Programming COHERENT

bzero() — Sockets Function (libsocket)

Initialize memory to NUL
void bzero(*address***,** *size***)**
char **address***;*
int *size***;**

The function **bzero()** initializes *size* bytes of memory to NUL, beginning at *address*.

See Also

libsocket, memset()

Notes

bzero() is included for compatibility with Berkeley socket code. It is equivalent to the standard C function **memset()**.

c — Command

Print multi-column output
c [-l*N* **] [-w***N* **] [-012]**

c reads lines from the standard input and writes them in columns on the standard output. The longest input line and the width of the page determine how many columns will fit across the page.

c recognizes the following options:

-l*N* Set the length of the page to *N* lines. **c** columnizes its output by pages when this option is used with mode 1 or mode 2.

-w*N* Set the width of the page to *N* characters. The default is 80.

-0 Multi-column mode 0. Order the fields horizontally across the page.

-1 Multi-column mode 1 (default mode). Order the fields vertically down each column; the last column may be short.

-2 Multi-column mode 2. Order the fields similarly to mode 1, but place blank fields in the last output line rather than the last column.

Options may also be given in the environmental variable **C**, separated by white space. Command line options override options in the environment. For example,

```
export C="-l56 -w72 -2"
c -w80 <file1
```

has the same effect as

```
c -l56 -w72 -2 -w80 <file1
```

This command sets the page width to 80 rather than to 72.

See Also

commands, export, pr

Diagnostics

c prints "out of memory" and returns an exit status of one if it cannot allocate enough memory to process its input.

C keywords — Overview

A **keyword** is a word that is reserved within C, and must not be used to name variables, functions, or macros. COHERENT recognizes the following C keywords:

alien	**auto**	**break**
case	**char**	**const**
continue	**default**	**do**
double	**else**	**enum**
extern	**float**	**for**
goto	**if**	**int**
long	**register**	**return**
short	**signed**	**sizeof**
static	**struct**	**switch**
typedef	**union**	**unsigned**
void	**volatile**	**while**

In conformity with the ANSI standard, the keywords **entry** and **readonly** are no longer recognized. The ANSI Standard transfers the functionality for **readonly** to the keyword **const**. For details, see the Lexicon entry for **const**.

The COHERENT C compiler recognizes the keywords **const** and **volatile**, but ignores them.

The following tokens are C++ keywords:

class
inline
private
protected
public

Your programs should avoid using them in the interest of compatibility with future versions of the COHERENT C compiler.

See Also

C language

C language — Overview

COHERENT includes a C compiler that fully implements the Kernighan and Ritchie standard of C, with extensions taken from the ANSI standard.

Please note that in the following discussion, *word* indicates an object 16 bits long; *dword*, an object 32 bits long; and *qword*, an object 64 bits long:

Identifiers

Characters allowed: **A-Z**, **a-z**, **_**, **0-9**
Case sensitive
Number of significant characters in a variable name: **255**

Escape Sequences

The COHERENT C compiler recognizes the following escape sequences:

	ASCII	*Ctrl*	*Hex*	*Description*
\a	BEL	<ctrl-G>	0x07	audible tone (bell)
\b	BS	<ctrl-H>	0x08	backspace
\f	FF	<ctrl-L>	0x12	formfeed
\n	LF	<ctrl-J>	0x0A	linefeed (newline)
\r	CR	<ctrl-M>	0x0D	carriage return
\t	HT	<ctrl-I>	0x09	horizontal tab
\v	VT	<ctrl-K>	0x0B	vertical tab
\xhh			0xhh	hex (one to four hex digits [0-9a-fA-F])
\ooo				octal (one to four octal digits [0-7])

Trigraphs

The COHERENT C compiler recognizes the following trigraphs:

Trigraph Sequence	*Character Represented*
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	|
??>	}
??-	~

For details, see the Lexicon entry **trigraph**.

Reserved Identifiers (Keywords)

See the Lexicon entry for **C keywords**.

Data Formats (in bits)

char	8
unsigned char	8
double	64
enum	8|32
float	32
int	32
unsigned int	32
long	32
unsigned long	32
pointer	32
short	16
unsigned short	16

Floating-Point Formats

IEEE floating-point **float**:
1 sign bit
8-bit exponent
24-bit normalized fraction with hidden bit
Bias of 127

IEEE floating-point **double**:
1 sign bit
11-bit exponent
53-bit fraction
Bias of 1,023

Reserved values:
+- infinity, -0

All floating-point operations are done as **double**s.

Limits

Maximum bitfield size: 32 bits
Maximum number of **case**s in a **switch**: no formal limit
Maximum number of arguments in function declaration: 32
Maximum number of arguments in function call: no formal limit
Maximum block nesting depth: no formal limit
Maximum parentheses nesting depth: no formal limit
Maximum structure size: no formal limit
Maximum array size: no formal limit

Preprocessor Instructions

#define **#ifdef**
#else **#ifndef**
#elif **#include**
#endif **#line**
#if **#undef**
#pragma

Structure Name-Spaces
Supports both Berkeley and Kernighan-Ritchie conventions for structure in union.

Function Linkage
Return values in EAX
Return values for **double**s:
With software floating-point emulation returns in EDX:EAX
Hardware floating-point (-VNDP) returns in the NDP stacktop **%st0**
Parameters pushed on stack in reverse order:
chars, **short**s, and pointers pushed as dwords
Structures copied onto the stack
Caller must clear parameters off stack
Stack frame linkage is done through ESP register

Structures and Alignment
Structure members are aligned according to the most strictly aligned type within the structure. For example, a structure is word-aligned if it contains only **short**s, but on dword if it contains an **int** or **long**. **#pragma align** *n* can override this feature.

Registers
Registers EBX, EDI, and ESI are available for register variables. Only 32-bit objects go into registers.

Special Features and Optimizations

Both implementations of C perform the following optimizations:

- Branch optimization is performed: this uses the smallest branch instruction for the required range.
- Unreached code is eliminated.
- Duplicate instruction sequences are removed.
- Jumps to jumps are eliminated.
- Multiplication and division by constant powers of two are changed to shifts when the results are the same.
- Sequences that can be resolved at compile time are identified and resolved.

Compilation Environments

COHERENT supports a number of different compilation environments. For example, you can compile a program to use the environment for UNIX System V release 4 or release 3, or the Berkeley environment. This is done by setting manifest constants on your C compiler's command line, which, in turn, invokes various settings within the header files. For details, see the Lexicon entry for **header files**.

Example

The following gives an example C program, which does something interesting. It was writen by Charles Fiterman:

```
char *x="char *x=%c%s%c;%cmain(){printf(x,34,x,34,10,10);}%c";
main(){printf(x,34,x,34,10,10);}
```

See Also

argc, argv, C keywords, C preprocessor, environ, envp, header files, initialization, libraries, main(), name space, offsetof(), Programming COHERENT, trigraph

C preprocessor — Overview

Preprocessing encompasses all tasks that logically precede the translation of a program. The preprocessor processes headers, expands macros, and conditionally includes or excludes source code.

Directives

The C preprocessor recognizes the following directives:

#if . . . . . Include code if a condition is true
#elif. . . . . Include code if directive is true
#else . . . . . Include code if preceding directives fail
#endif . . . . . End of code to be included conditionally

#ifdef. . . . . Include code if a given macro is defined
#ifndef . . . . . Include code if a given macro is not defined

#define. . . . . Define a macro
#undef . . . . . Undefine a macro
#include . . . . . Read another file and include it
#line . . . . . Reset current line number

The COHERENT preprocessor also recognizes the directive **#pragma**, which performs implementation-specific tasks. See the Lexicon entry on **#pragma** for details.

A preprocessing directive is always introduced by the '#' character. The '#' must be the first non-white space character on a line, but it may be preceded by white space and it may be separated from the directive name that follows it by one or more white space characters.

Preprocessing Operators

The Standard defines two operators that are recognized by the preprocessor: the "stringize" operator **#**, and the "token-paste" operator **##**. It also defines a new keyword associated with preprocessor statements: **defined**.

The operator **#** indicates that the following argument is to be replaced by a string literal; this literal names the preprocessing token that replaces the argument. For example, consider the macro:

```
#define display(x) show((long)(x), #x)
```

When the preprocessor reads the line

```
display(abs(-5));
```

it replaces it with the following:

```
show((long)(abs(-5)), "abs(-5)");
```

The **##** operator performs "token pasting" — that is, it joins two tokens together, to create a single token. For example, consider the macro:

```
#define printvar(x) printf("%d\n", variable ## x)
```

When the preprocessor reads the line

```
printvar(3);
```

it translates it into:

```
printf("%d\n", variable3);
```

In the past, token pasting had been performed by inserting a comment between the tokens to be pasted. This no longer works.

Predefined Macros

The ANSI Standard describes the following macros that must be recognized by the preprocessor:

__DATE__ Date of translation
__FILE__ Source-file name
__LINE__ Current line within source file
__STDC__ Conforming translator and level
__TIME__ Time of translation

For more information on any one of these macros, see its entry.

Conditional Inclusion

The preprocessor will conditionally include lines of code within a program. The directives that include code conditionally are defined in such a way that you can construct a chain of inclusion directives to include exactly the

material you want.

The preprocessor keyword **defined** determines whether a symbol is defined to the **#if** preprocessor directive. For example,

```
#if defined(SYMBOL)
```

or

```
#if defined SYMBOL
```

is equivalent to

```
#ifdef SYMBOL
```

except that it can be used in more complex expressions, such as

```
#if defined FOO && defined BAR && FOO==10
```

defined is recognized only in lines beginning with **#if** or **#elif**.

Note that **defined** is a preprocessor keyword, not a preprocessor directive or a C keyword. You could, for example, write a function called **defined()** without any complaint from the C compiler.

The COHERENT preprocessor implicitly defines the following macros:

```
__COHERENT__
__MWC__
__IEEE__
__I386__

_IEEE
_I386
MWC
COHERENT
```

These can be used to include conditionally code that applies to a specific edition of COHERENT. COHERENT 286 uses DECVAX floating-point code; whereas COHERENT 386 uses IEEE. If you were writing code that intensively used floating-point numbers and you wanted to compile the code under both editions of COHERENT, you could write code of the form:

```
#ifdef _DECVAX
      ...
#elif _IEEE
      ...
#endif
```

The C preprocessor under each edition of COHERENT would ensure that the correct code was included for compilation.

Macro Definition and Replacement

The preprocessor performs simple types of macro replacement. To define a macro, use the preprocessor directive **#define** *identifier value*. The preprocessor scans the translation unit for preprocessor tokens that match *identifier*; when one is found, the preprocessor substitutes *value* for it.

Inclusion of Macros or Functions

The ANSI standard demands that every routine implemented as a macro also be implemented as a function, with the exception of the macro **va_arg()**. For example, COHERENT implements the STDIO routines **toupper()** and **tolower()** both as macros and functions.

By default, COHERENT uses the macro version of routines. To force it to use the function of a routine, you must undefine the macro version. You can do that either by using the preprocessor instruction **#undef** in your code, or by using the option **-U** on the **cc** command line. For example, to compel COHERENT to use the function version of **tolower()**, include the statement

```
#undef tolower
```

in your program, or include the argument

```
-Utolower
```

on the **cc** command line.

cpp

Under COHERENT, C preprocessing is done by the program **cpp**. The **cc** command runs **cpp** as the first step in compiling a C program. **cpp** can also be run by itself.

cpp reads each input *file*; it processes directives, and writes its product on **stdout**.

If its **-E** option is not used, **cpp** also writes into its output statements of the form **#line** *n filename*, so that the parser **cc0** can connect its error messages and debugger output with the original line numbers in your source files.

See the Lexicon entry on **cpp** for more information.

See Also

C language, cc, cpp, defined, macro, manifest constant,

cabs() — Mathematics Function (libm)

Complex absolute value function
#include <math.h>
double cabs(*z***) struct { double** *r***,** *i***; }** *z***;**

cabs() computes the absolute value, or modulus, of its complex argument *z*. The absolute value of a complex number is the length of the hypotenuse of a right triangle whose sides are given by the real part *r* and the imaginary part *i*. The result is the square root of the sum of the squares of the parts.

Example

For an example of this function, see the entry for **acos()**.

See Also

hypot(), lib

cal — Command

Print a calendar
cal [*month*] [*year*]

cal prints a calendar for the specified *year* (by default, the current year), or for the given *month* if one is specified. If neither is specified, a calendar of the current month is printed. *year* must be between 1 and 9999. *month* may be either the month name (lower case, spelled out or first three letters) or a number between 1 and 12.

For example, try:

```
cal september 1752
```

See Also

commands

Notes

cal assumes that the Gregorian calendar was adopted on September 3, 1752, which is the date of its adoption throughout the British empire.

calendar — Command

Reminder service
calendar [-a] [-f*file* **]... [-d[***date***]] [-w[***date***]] [-m[***month***]]**

calendar is the COHERENT system's "reminder service". It reads a calendar file, which should contain information organized by date; if an event is scheduled to happen today or tomorrow, **calendar** prints the entry on the standard output. Thus, you can use **calendar** to remind you of both one-time events (such as appointments) and yearly events (such as anniversaries).

calendar recognizes the following command-line options:

-a Search the calendars of all users and send mail. Default is to search only your calendar.

-f*file* Search each "file" in order given. Default is **$HOME/.calendar**.

-d[*date***]** Print all entries for "date". Default date is today.

-w[*date***]** Print all entries for the week beginning with "date".

-m[*month***]** Print entries for the given "month".

By default, **calendar** print entries for today and tomorrow, with "tomorrow" encompassing the following Monday should "today" be a Friday or Saturday. If an entry in your **.calendar** has an at-sign '@' embedded in it, **calendar** prints it regardless of when it is to occur, until its date has passed.

The following gives an example of a calendar file. As you can see, **calendar** understands different formats of dates:

```
Apr 16    Dave's birthday
7/6       Dad's birthday
Sep 26    Mom's birthday
Jun 30    Barry's birthday
10/4      Marianne's birthday
Jul 31    Anniversary!
Mar 16    Pot luck luncheon
```

You can run **calendar** automatically by embedding the command

```
calendar
```

in your **.profile**.

If you wish, you can run **calendar** automatically for yourself, by inserting it into file **/usr/spool/cron/crontabs/root**. In this case, **calendar** should be used with its **-a** option, to force it to search each user's **$HOME** directory for **.calendar** and mail the appointments it finds to that user.

See Also

commands

Notes

calendar's notion of tomorrow understands weekends but not holidays. Thus, if you invoke **calendar** on a Friday, it returns the events for that day and the following Saturday, Sunday, and Monday. If Monday is a holiday, however, you will not receive appointments for Tuesday.

calling conventions — Definition

The following presents the calling conventions for COHERENT.

The calling conventions of C take into account machine architecture and the fact that the number of arguments passed to a function may vary, as in the functions **printf()** and **scanf()**.

For example, consider the following C program, called **foo.c**:

```
short a;
long b;
char c;

foo()
{
        example(a, b, c);
}
```

Compiling this program with the command

```
cc -S foo.c
```

generates the assembly-language code (with added comments):

```
  .alignoff
  .comm     a,          2       / a, b, and c are commons in the .bss
  .comm     b,          4
  .comm     c,          1
```

```
foo:
        .text
        .globl foo

foo:
        push      %ebp
        movl      %ebp, %esp
        movsxb    %eax, c           / move c to %eax with sign extend
        push      %eax              / pass c
        push      b                 / pass b
        movsxw    %eax, a           / move a to %eax with sign extend
        push      %eax              / pass a
        call      example

        leave                       / epilog code for foo
        ret
        .align    4
```

Note the following points:

- Parameters are pushed in reverse order. You should not depend on this feature, as the ANSI standard says that parameters may be calculated and pushed in any order.
- The stack is reset by the caller, not the callee. Only the caller knows the number of parameters pushed.
- All parameters become **int** or **double** when passed under Kernighan & Ritchie C. This changes under ANSI C.

Now consider the module **example.c**, which gives the receiving end:

```
double
example(x, y, z)
short x;
long y;
char z;
{
      int tmp;

      tmp = x * y;
      return (tmp + z);
}
```

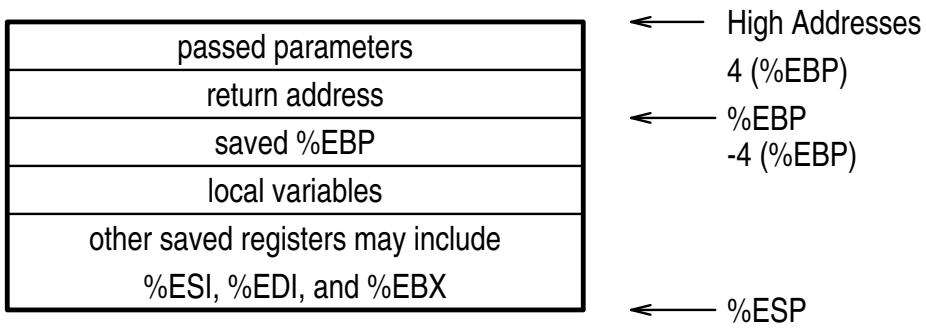
The command

```
cc -S example.c
```

generates the code:

```
        .alignoff

        .text
        .globl example

example:
        enter     $4, $0            / 4 bytes of local variables
        push      %edi
        movl      %eax, 12(%ebp)    / x * y
        imull     8(%ebp)           / 8 == 4 + sizeof(int)
        movl      -4(%ebp), %eax    / save into tmp
        movl      %edi, 16(%ebp)    / tmp + z
        addl      %edi, %eax        / return double in EDX:EAX
        movl      %eax, %edi
        call      _dicvt
        pop       %edi

        leave                       / leave with result in %eax:%edx
        ret
        .align    4
```

After the prologue code, the stack always looks like



Notice that parameters start at

```
[4 + first parm size](%ebp)
```

and go to higher addresses, whereas local variables start at

```
-4(%ebp)
```

and go to lower addresses. Therefore, if you have a local array and overwrite it in the forward direction, you clobber your caller's **%ebp**; if you overwrite it in the backward direction, you clobber your caller's register variables (although if the caller has no register variable, it's harmless).

On the 80386, the stack starts at 0x80000000 and grows down being expanded by the system as it is needed. Reasonable programs should never have stack-overflow problems, as they did under COHERENT 286.

Note that the convention for returning floating-point numbers differ depending upon whether a program uses software floating-point emulation, or hardware floating-point code as invoked by the **cc** option **-VNDP**. Programs that use hardware floating point return **double** in the NDP stack top **$st0**.

See Also

C language, Programming COHERENT,

calloc() — General Function (libc)

Allocate dynamic memory
#include <stdlib.h>
char *calloc(*count***,** *size***)**
unsigned *count***,** *size***;**

The function **calloc()** is one of a set of routines that helps manage a program's arena. **calloc()** calls **malloc()** to obtain a block large enough to contain *count* items of *size* bytes each; it then initializes the block to zeroes. When this memory is no longer needed, you can return it to the free pool by using the function **free()**.

calloc() returns the address of the chunk of memory it has allocated, or NULL if it could not allocate memory.

Example

This example attempts to **calloc()** a small portion of memory; it then reallocates it to demonstrate **realloc()**.

```
#include <stdio.h>
#include <stdlib.h>

main()
{
      register char *ptr, *ptr2;
      extern char *calloc(), *realloc();
      unsigned count, size;

      count = 4;
      size = sizeof(char *);
```

```
      if ((ptr = calloc(count, size)) != NULL)
            printf("%u blocks of size %u calloced\n",
                    count, size);
      else
            printf("Insuff. memory for %u blocks of size %u\n",
                 count, size);

      if ((ptr2 = realloc(ptr,(count*size) + 1)) != NULL)
            printf("1 block of size %u realloced\n",
                  (count*size)+1);
}
```

See Also

alloca(), arena, free(), libc, malloc(), memok(), realloc(), setbuf(), stdlib.h
ANSI Standard, §7.10.3.1
POSIX Standard, §8.1

Notes

The function **alloca()** allocates space on the stack. The space so allocated does not need to be freed when the function that allocated the space exits.

cancel — Command

Cancel a print job
cancel [*job* **[** *... job***]] [-all]**

The command **cancel** cancels execution of a printing job. It recognizes the following options:

-all Cancel all requests that are currently executing.

job Cancel each *job*. Each *job* is identified by the number printed by **lp** when the job was first spooled.

When a job is cancelled, it remains in the print queue for the remainder of its "lifetime", and may be printed later. When it cancels a job, **cancel** sends mail to the owner of the job to notify him of the job's cancellation.

cancel does not affect jobs that have already been downloaded into their destination printers. The only way to stop a job from printing after it has been downloaded is to clear the printer's memory. See the documentation that came with your printer for instructions on how to do that.

See Also

commands, lp, printer

Notes

cancel is a link to **lpstat**.

cancel is available only under COHERENT release 4.2 and subsequent releases.

canon.h — Header file

Portable layout of binary data
#include <canon.h>
#include <sys/types.h>

The routines declared in **canon.h** were designed to aid the transfer of binary information among different implementations of COHERENT. For technical reasons, these routines are slated to be dropped from a future release of COHERENT. Their use is strongly discouraged.

See Also

ar.h, byte ordering, header files,

captoinfo — Command

Convert termcap data to terminfo form
captoinfo [*filename***]**

The command **captoinfo** converts a file of terminal information that is in the **termcap** format into **terminfo** source format.

captoinfo reads *filename*; if no file is named on the command line, it reads the standard input. It writes its product to the standard output.

The input to **captoinfo** must be in correct **termcap** format. **captoinfo** complains about all constructs that it cannot interpret.

See Also

commands, termcap, terminfo, tic

Notes

The original code for **captoinfo** was written by Robert Viduya of the Georgia Institute of Technology, and was adapted for COHERENT by Mark Williams Company.

case — Command

Execute commands conditionally according to pattern
case *token* **in [***pattern* **[|** *pattern***] ...)** *sequence* **;;] ... esac**

case is a construct that used by the shell. It tells the shell to execute commands conditionally, according to a pattern. It tests the given *token* successively against each *pattern*, in the order given. It then executes the commands in the *sequence* corresponding to the first matching pattern. Optional '|' clauses specify additional patterns corresponding to a single *sequence.* If no *pattern* matches the *token*, the **case** construct executes no commands.

Each *pattern* can include text characters (which match themselves), special characters '?' (which matches any character except newline) and '*' (which matches any sequence of non-newline characters), and character classes enclosed in brackets '[ ]'; ranges of characters within a class may be separated by '-'. In particular, the last *pattern* in a **case** construct is often '*', which will match any *token*.

The shell executes **case** directly.

Example

The following example prints a string in response to a command-line option:

```
case $1 in
      FOO)  echo "This is option FOO";;
      BAR)  echo "This is option BAR";;
      BAZ)  echo "This is option BAZ";;
      *)    echo "An asterisk marks the default option";;
esac
```

See Also

commands, ksh, sh

case — C Keyword

Introduce entry in switch statement

The C keyword **case** is a label within a **switch** statement. For example:

```
while ((int = getchar()) != EOF)
      switch (foo) {
      case 'q':
      case 'Q':
            exit(0);
      case ' ':
            n++;
      default:
            break;
      }
```

case labels each of the three possibilities recognized by the **switch** statement: a space, 'q', and 'Q'. The statements that follow a **case** statement behave as if they were enclosed within braces.

Note that a **case** statement is simply a label: it sets a point to which the **switch** statement jumps, and execution continues from that point. Once a **switch** statement jumps to the point marked by a given **case** label, execution continues until an **exit**, **break**, or **return** is read, or the closing brace of the **switch** statement is encountered.

See Also

break, C keywords, switch
ANSI Standard, §6.6.4.2

cast — Definition

The *cast* operation "coerces" a variable from one data type to another.

There are two reasons to cast a variable. The first is to convert a variable's data into a form acceptable to a given function. For example, the function **hypot** takes two **double**s. If the variables **leg_x** and **leg_y** are **float**s, the rules of C require that they be cast automatically to **double**. If the compiler did not do not do this, **hypot** would grab a **double**'s worth of memory: the four bytes of your **float**, plus four bytes of whatever happens to be sitting on the stack. The leads to results that are less than totally accurate.

The other reason to cast a variable is when you cast one type of pointer to another. For example,

```
char *foo;
int *bar;
bar = (int *)foo;
```

Although **foo** and **bar** are of the same length, you would cast **foo** in this instance to stop the C compiler from complaining about a type mismatch.

See Also

data formats, data types, Programming COHERENT

cat — Command

Concatenate the contents of a file to the standard output
cat [**-u**][*file ...*]

cat copies each *file* arguments to the standard output. A '-' tells **cat** to read the standard input. If no *file* is specified, **cat** reads the standard input.

The **-u** option makes the output unbuffered. Otherwise, **cat** buffers the output in units of the machine's disk block size (e.g., 512 bytes).

See Also

commands

Notes

If you redirect **cat**'s the output to one of its input files, it will loop forever, reading from the file the text that it has just written into it: in effect, **cat** will chase its own tail endlessly.

caveat utilitor — Definition

Latin (sort of): "Let the user beware." Cf, "Heads up!" in the American dialect.

See Also

Using COHERENT

cc — Command

C compiler
cc [*compiler options***]** *file . . . .* **[***linker options***]**

cc is the program that compiles C programs. It guides files of source and object code through each phase of compilation and linking. **cc** has many options to assist in the compilation of C programs; in essence, however, all you need to do to produce an executable file from your C program is type **cc** followed by the name of the file or files that hold your program. **cc** checks whether the file names you give it are reasonable, selects the right phase for each file, and performs other tasks that ease the compilation of your programs.

How cc Works

cc works as follows:

- If a file ends in **.c**, **cc** assumes that it contains C code, and compiles it. The compiler generates a relocatable object module with the suffix **.o**.

- If the file has the suffix **.s**, **cc** assumes that it is a file of assembly language, and invokes the assembler **as** to assemble it. The assembler also generates a relocatable object module with the suffix **.o**.
- **cc** assumes that all files with the suffix **.o** are relocatable object modules. It also assumes that all files with the suffix **.a** are libraries of object modules. It passes both directly to the linker **ld**. Additional libraries can also be invoked by using the **-l** option **cc**, described below.
- Once all files of C code and assembly language have been compiled or assembled, **cc** then invokes the linker **ld** to link the newly created object files with any objects and libraries you named on **cc** command line. It also automatically includes the C runtime startup routine and the standard C library, so you do not have to name these on your **cc** command line.
- **cc** also cleans up after itself. It removes all of its temporary files automatically. If only one object file is created during compilation, **cc** deletes it after linking; however, if more than one object file is created, or if an object file of the same name existed before you began to compile, then the object file or files are not deleted.

Assuming that no error occurs along the way, **cc** writes the linked result into a file named after the *file* on its command line, minus that file's suffix — **.c**, **.s**, or **.o**, depending upon the type of data *file* holds. It is now ready to be executed.

Options

The following lists all of **cc**'s command-line options. **cc** passes some options through to the linker **ld** unchanged, and correctly interprets for it the options **-o** and **-u**.

A number of the options are esoteric and normally are not used when compiling a C program. The following are the most commonly used options:

-c	Compile only; do not link
-f	Link in floating-point **printf()**
-l*name*	Pass library **lib***name***.a** to linker
-o *name*	Call output file *name*
-V	Print verbose listing of **cc**'s action

? Print a detailed usage message that describes available **cc**'s options to the standard output.

-A MicroEMACS option. If an error occurs during compilation, **cc** automatically invokes the MicroEMACS screen editor. The error or errors are displayed in one window and the source code file in the other, with the cursor set to the line number indicated by the first error message. Typing **<ctrl-X>>** moves to the next error, **<ctrl-X><** moves to the previous error. To recompile, close the edited file with **<ctrl-Z>**. Compilation will continue either until the program compiles without error, or until you exit from the editor by typing **<ctrl-U>** followed by **<ctrl-X><ctrl-C>**.

-a By default, **cc** generates an executable file that is named after the source module. For example, the command

```
cc foo.c
```

generates an executable named **foo**. If you name more than source module on the **cc** command line, by default it names the executable after the first module you name. The option **-a** tells **cc** to create an executable file named **a.out**. This is for compatibility with other versions of UNIX. Note that option **-o**, described below, overrides the effect of **-a**.

-B[*path***]**
Backup option. Use an alternative *path* for the compiler phases **cc0**, **cc1**, **cc2**, and **cc3**. If *path* is supplied, **cc** prefixes it onto the name of each phase of the compiler, to form the name of the new compiler phase, and the path to the directory in which it lives. If you do not supply a *string*, **cc** prefixes the name of the current directory.

If you precede a **-B** option with a **-t** option, the **-B** option affects only the phase of the compiler that the **-t** option names. For example, the command

```
cc -t0 -B/usr/fred/bin hello.c
```

compiles **hello.c** using the version of **cc0** found in directory **/usr/fred/bin**. You can include any number of pairs of **-t** and **-B** options, with each **-t** option naming phase of the compiler that the subsequent **-B** option affects.

If followed by the prefix option **-M**, the name of the compiler phase in question is prefixed by the string named in the **-M** option. For example, the command

```
cc -t0 -B/usr/fred/cc -Mnew a.c
```

tells the compiler to look for **/usr/fred/cc/newcc0** and execute instead of the usual **cc0**.

-c Compile option. Suppress linking and the removal of the object files.

-D*name***[=***value***]**
Define *name* to the preprocessor, as if set by a **#define** directive. If *value* is present, it is used to initialize the definition.

-E Expand option. Run the C preprocessor **cpp** and write its output onto the standard output.

-f Floating-point option. Include the version of **printf()** that converts floating-point numbers to text. If a program is compiled without the **-f** option but attempts to print a floating-point number during execution by using the **e**, **f**, or **g** format specifications to **printf()**, the program prints the error message

```
You must compile with -f option for floating point
```

and exits.

Note that if you wish to include the **libm** library routines that perform floating-point mathematics functions, you must specify **-lm** on the command line to load the library **libm.a**.

-g Generate debugging information. Same as option **-VDB**, described below.

-I*name*
Include option. Specify a directory the preprocessor should search for files given in **#include** directives, using the following criteria: If the **#include** statement reads

```
#include "file.h"
```

cc searches for **file.h** first in the source directory, then in the directory named in the **-I***name* option, and finally in the system's default directories. If the **#include** statement reads

```
#include <file.h>
```

cc searches for **file.h** first in the directories named in the **-I***name* option, and then in the system's default directories. Multiple **-I***name* options are executed in the order of their appearance.

-K Keep option. Do not erase the intermediate files generated during compilation. Temporary files will be written into the current directory.

-L*directory*
Tell the linker **ld** to search *directory* for its libraries before it searches the directories named in the environmental variable **LIBPATH**. You can use multiple **-L** options in a **cc** command.

-l*name*
Pass the name of a library to the linker. **cc** expands **-l***name* into **/lib/lib***name***.a**. If an alternative library prefix has been specified by the **-tl** and **-B***string* options, then **-l***name* expands to *string***lib***name***.a**. Note that this is a *linker option*, and so must appear at the end of the **cc** command line, or it will not be processed correctly.

-M*string*
Machine option. Use an alternate version of **cc0**, **cc1**, **cc1a**, **cc1b**, **cc2**, **cc3**, **as**, **lib*.a**, and **crts0.o,** named by fixing *string* between the directory name and the pass and file names. For examples, see the description of option **-B**, above. Before release 4.0 of COHERENT, **cc** executed the compiler phases **/lib/cc0** through **/lib/cc3**. Beginning with release 4.0, **cc** itself contains all the compiler phases; the preprocessor **/lib/cpp** executes the parser **/lib/cc0**, but compiler phases **/lib/cc[123]** do not exist for **cc**.

-o *name*
Output option. Rename the executable file from the default to *name*. Unlike UNIX, the COHERENT implementation of **cc** by default names an executable after the first first **.c** or **.o** file given on the command line, instead of naming it **a.out**. If you want **cc** to conform to the UNIX standard, set include the option **-o a.out** when you set the environmental variable **CCHEAD**. This environmental variable is described below. Another approach is to invoke **make** to control compilation. For details, see the Lexicon entry for **make**.

-O Optimize option. Run the code generated by the C compiler through the peephole optimizer. The optimizer pass is mandatory for the i8086, Z8000, and M68000 compilers, and need not be requested. It is optional for the PDP-11 compiler, but is recommended for all files except those that consist entirely of initialized tables of data.

-p Generate code to profile functions calls. Programs compiled with this option can be run with the command **prof** to print a summary of how much time the program spends in each subroutine, to help you optimize your programs. You must use this option to compile each module whose functions you wish to examine; and you must also use this option on the **cc** command line with which you link the program, to ensure that the appropriate library routines are linked into your executable.

-Q Quiet option. Suppress all messages, no matter how awful an error they indicate.

-S Suppress the object-writing and link phases, and invoke the disassembler **cc3**. This option produces an assembly-language version of a C program for examination, for example if a compiler problem is suspected. The assembly-language output file name replaces the **.c** suffix with **.s**. This is equivalent to the **-VASM** option.

-T*size*

cc writes its temporary data into two 64-kilobytes buffers that grow as needed. The **-T** option tells **cc** to use buffers of *size* bytes each. Setting these to a larger size may help large files compile faster. Setting *size* to zero forces **cc** to use temporary files written onto the disk.

-t*phase*

Take option. Use an alternate versions of the phase or phases of the compiler specified by **phase**, which must consist of one or more of the characters **01ab23sdlrt**. If no *phase* string appears, **cc** uses alternate version of every phase of the compiler, except the preprocessor. If the **-t** option is followed by a **-B** option, **cc** prefixes the path named in the **-B** option to the phases and files named in the **-t** option. For examples, see the description of option **-B**, above,

-U*name*

Undefine symbol *name*. Use this option to undefine symbols that the preprocessor defines implicitly, such as the name of the native system or machine. Users who wants serious ISO namespace compliance should compile with the options:

```
-UCOHERENT -UMWC -U_I386 -U_IEEE
```

These options turn off the macros **COHERENT**, **MWC**, **_I386**, and **_IEEE**, all of which are automatically defined by the COHERENT preprocessor.

-V Verbose option. **cc** prints onto the standard output a step-by-step description of each action it takes.

-V*string*

Variant option. Toggle (i.e., turn on or off) the variant *string* during the compilation. Variants that are marked **on** are turned on by default. Options marked **Strict:** generate messages that warn of the conditions in question. **cc** recognizes the following variants:

-VASM
Output assembly-language code. Identical to **-S** option, above. Default is **off**.

-VCOMM
Permit **.com**-style data items. Default is **on**.

-VCPLUS
Ignore C++-style comments, which are deliminted by '//'.

-VDB
Generate debugging information, same as option **-g** described above. Default is **off**.

-VFLOAT
Include floating-point **printf()** code. Same as option **-f**, described above.

-VNDP
Generate code to execute hardware floating-point arithmetic. **cc** executes floating-point arithmetic on an 80387 or 80486-DX, if present; or use software emulation if it is not. For more information, see the section on hardware floating-point arithmetic, below.

-VNOWARN
Suppress all warning and strict messages. Use this option if you wish to suppress cascades of warning message about, say, nested comments.

-VPROF
Same as the option **-p**, described above.

-VPSTR
"imPure" strings: Place all string literals into the **.data** segment rather than in **.text**. This may be necessary for sloppily written code that assumes it can overwrite string literals.

-VQUIET
Suppress all messages. Identical to **-Q** option. Default is **off**.

-VS Turn on all strict checking. Default is **on**.

-VSBOOK
Strict: note deviations from *The C Programming Language*, ed. 1. Default is **off**.

-VSCCON
Strict: note constant conditional. Default is **off**.

-VSINU
Implement struct-in-union rules instead of Berkeley-member resolution rules. Default is **off**, i.e., Berkeley rules are the default.

-VSLCON
Strict: **int** constant promoted to **long** because value is too big. Default is **on**.

-VSMEMB
Strict: check use of structure/union members for adherence to standard rules of C. Default is **on**.

-VSNREG
Strict: register declaration reduced to auto. Default is **on**.

-VSPVAL
Strict: pointer value truncated. Default is **off**.

-VSRTVC
Strict: risky types in truth contexts. Default is **off**.

-VSTAT
Give statistics on optimization.

-VSUREG
Strict: note unused registers. Default is **off**.

-VSUVAR
Strict: note unused variables. Default is **on**.

-VVERSION
Print to the standard error the compiler's version number. This information is useful when reporting bugs.

-VWIDEN
Warn the user if a parameter is widened from **char** or **short** to **int**, or from **float** to **double**. Default is **off**.

-V3GRAPH
Translate ANSI trigraphs. Default is **off**.

cc reads the environmental variables **CCHEAD** and **CCTAIL** and appends their contents to, respectively, the beginning and the end of the **cc** command. For example, if you insert the following entries into your **.profile**

```
export CCHEAD='-f -o a.out'
export CCTAIL=-lm
```

then **cc** will always use the floating-point version of **printf()**, always write its executable into file **a.out**, and always link in the mathematics library **libm**. In effect, it turns the command

```
cc hello.c
```

into:

```
cc -f -o a.out hello.c -lm
```

If you set a command option in **CCHEAD** or **CCTAIL**, you can always override it for specific **cc** commands. For example, if you have set **-o a.out** in **CCHEAD**, typing the command

```
cc -o hello hello.c
```

generates the command:

```
cc -o a.out -o hello hello.c
```

The latter **-o** option is the one used, and in effect cancels the effect of the **CCHEAD** entry. Thus, setting **CCHEAD** and **CCTAIL** give you a flexible way to set **cc**'s default behavior.

Note that

```
CCHEAD='-Wa,-f -Wl,-oa.out'
```

will give you a compilation environment that matches that of the UNIX operating system.

Linking Objects

The linker **ld** does not know about paths: it links exactly what you tell it to link via the **cc** command line. **cc** looks for compiler phases and for runtime startoff and library by searching the directories named in the environmental variable **LIBPATH**. If you do not define **LIBPATH** in your environment, it searches the default **LIBPATH** as defined in **/usr/include/path.h**. If you define **LIBPATH**, **cc** searches the directories in the order you specify. For example, a typical definition is:

```
export LIBPATH=:/lib:/usr/lib
```

This searches the current directory '.', then **/lib**, then **/usr/lib**.

Hardware Floating-Point Arithmetic

The C compiler shipped with version of COHERENT prior to release 4.2 generated software floating-point calls. That is, floating-point code such as

```
d1 = d2 + 2.5;
```

generated calls to software routines to perform the desired operations. This is called "software floating-point arithmetic".

Beginning with release 4.2.05 of COHERENT, **cc** generates software floating-point arithmetic by default, but let you select "hardware floating-point arithmetic". With hardware floating-point arithmetic, **cc** generates calls to execute floating-point operations on a numeric data processor (NDP), such as the 80387. To do so, use the option **-VNDP**. A program compiled to perform hardware floating-point arithmetic runs correctly on any computer: if the computer contains an NDP, the code executes on that part; but if the computer does not contain an NDP, the code emulates the operation of the NDP. Note that persons who do *not* have an NDP on their system must have the floating-point emulation module linked into their kernels; those who do have an NDP, however, do not need this module. The libraries in directories **/lib** and **/usr/lib** are compiled using software floating-point arithmetic; the libraries compiled with hardware floating-point arithmetic are kept in sub-directories **/lib/ndp** and **/usr/lib/ndp**.

As mentioned above, code compiled to use hardware floating-point arithmetic runs much faster when your machine has an NDP installed. If your system does not have a numeric co-processor (i.e., an 80387, 80487, an 80486DX, or a Pentium) and you wish to run programs that intensively use floating-point arithmetic, we strongly urge you to consider upgrading your system to use an NDP.

Files

/bin/cc — C compiler

See Also

as, C language, cc0, cc1, cc2, cc3, commands, C preprocessor, cpp, ld, LIBPATH, make, makedepend, TMPDIR
The C Language tutorial

Diagnostics

The following gives the error messages returned by the COHERENT C compiler. The messages are in alphabetical order, and each is marked as to whether it is a *fatal*, *error*, *warning*, or *strict* condition. A fatal message usually indicates a condition that caused the compiler to terminate execution. Fatal errors from the later phases of compilation often cannot be fixed, and may indicate problems in the compiler or assembler. An error message points to a condition in the source code that the compiler cannot resolve. This almost always occurs when the program does something illegal, e.g., has unbalanced braces. Warning messages point out code that is compilable, but may produce trouble when the program is executed. A strict message refers to a passage in the code that is unorthodox and may not be portable. For error messages produced by the assembler **as**, the linker **ld**, and the preprocessor **cpp**, see their respective entries in the Lexicon.

ambiguous reference to "*string*" *(error)*
: *string* is defined as a member of more than one **struct** or **union**, is referenced via a pointer to one of those **struct**s or **union**s, and there is more than one offset that could be assigned.

argument list has incorrect syntax *(error)*
: The argument list of a function declaration contains something other than a comma-separated list of formal parameters.

array bound must be a constant *(error)*
: An array's size can be declared only with a constant; you cannot declare an array's size by using a variable. For example, it is correct to say **foo[5]**, but illegal to say

```
bar = 5;
foo[bar];
```

array bound must be positive *(error)*
: An array must be declared to have a positive number of elements. The array flagged here was declared to have a negative size, e.g., **foo[-5]**.

array bound too large *(error)*
: The array is too large to be compiled with 32-bit index arithmetic. You should devise a way to divide the array into compilable portions.

array row has 0 length *(error)*
: This message can be triggered by either of two problems. The first problem is declaring an array to have a length of zero; e.g., **foo[0]**. The second problem is failing to declare the size of a dimension *other than the first* in a multi-dimensional array. C allows you to declare an indefinite number of array elements of *n* bytes each, but you cannot declare *n* array elements of an indefinite length. For example, it is correct say **foo[][5]** but illegal to say **foo[5][]**.

associative expression too complex *(fatal)*
: An expression that uses associative binary operators (e.g., '+') has too many operators; for example, **i=i1+i2+i3+ . . . +i30;**. You should simplify the expression.

bad argument storage class *(error)*
: An argument was assigned a storage class that the compiler does not recognize. The only valid storage class is **register**.

bad external storage class *(error)*
: An **extern** has been declared with an invalid storage class, e.g., **register** or **auto**.

bad field width *(error)*
: A field width was declared either to be negative or to be larger than the object that holds it. For example, **char foo:9** or **char foo:-1** will trigger this error.

bad filler field width *(error)*
: A filler field width was declared either to be negative or to be larger than the object that holds it. For example, **char foo:9** or **char foo:-1** will trigger this error.

bad flexible array declaration *(error)*
: A flexible array is missing an array boundary; e.g., **foo[5][]**. C permits you to declare an indefinite number of array elements of *n* bytes each, but you cannot declare an array to have *n* elements of an indefinite number of bytes each.

break not in a loop *(error)*
A **break** occurs that is not inside a loop or a **switch** statement.

call of non function *(error)*
What the program attempted to call is not a function. Check to make sure that you have not accidentally declared a function as a variable; e.g., typing **char *foo;** when you meant **char *foo();**.

cannot add pointers *(error)*
The program attempted to add two pointers. **int**s or **long**s may be added to or subtracted from pointers, and two pointers to the same type may be subtracted, but no other arithmetic operations are legal on pointers.

cannot apply unary '&' to a register variable *(error)*
Because register variables are stored within registers, they do not have addresses, which means that the unary **&** operator cannot be used with them.

cannot cast double to pointer *(error)*
The program attempted to cast a **double** to a pointer. This is illegal.

cannot cast pointer to double *(error)*
The program attempted to cast a pointer to a **double**. This is illegal.

cannot cast structure or union *(error)*
The program attempted to cast a **struct** or a **union**. This is illegal.

cannot cast to structure or union *(error)*
The program attempted to cast a variable to a **union** or **struct**. This is illegal.

cannot declare array of functions *(error)*
For example, the declaration **extern int (*f)[]();** declares **f** to be an array of pointers to functions that return **int**s. Arrays of functions are illegal.

cannot declare flexible automatic array *(error)*
The program does not explicitly declare the number of elements in an automatic array.

cannot initialize fields *(error)*
The program attempted to initialize bit fields within a structure. This is not supported.

cannot initialize unions *(error)*
The program attempted to initialize a **union** within its declaration. **union**s cannot be initialized in this way.

string: cannot reopen *(fatal)*
The optimizer cannot reopen a file with which it has worked. Make sure that your mass storage device is working correctly and that it is not full.

case not in a switch *(error)*
The program uses a **case** label outside of a **switch** statement. See the Lexicon entry for **case**.

character constant overflows long *(error)*
The character constant is too large to fit into a **long**. It should be redefined.

character constant promoted to long *(warning)*
A character constant has been promoted to a **long**.

class not allowed in structure body *(error)*
A storage class such as **register** or **auto** was specified within a structure.

compound statement required *(error)*
A construction that requires a compound statement does not have one, e.g., a function definition, array initialization, or **switch** statement.

constant expression required *(error)*
The expression used with a **#if** statement cannot be evaluated to a numeric constant. It probably uses a variable in a statement rather than a constant.

constant "*number*" promoted to long *(warning)*
The compiler promoted a constant in your program to **long**; although this is not strictly illegal, it may create problems when you attempt to port your code to another system, especially if the constant appears

in an argument list.

constant used in truth context *(strict)*
A conditional expression for an **if**, **while**, or **for** statement has turned out to be always true or always false. For example, **while(1)** will trigger this message.

construction not in Kernighan and Ritchie *(strict)*
This construction is not found in *The C Programming Language*; although it can be compiled by COHERENT, it may not be portable to another compiler.

continue not in a loop *(error)*
The program uses a **continue** statement that is not inside a **for** for **while** loop.

declarator syntax *(error)*
The program used incorrect syntax in a declaration.

default label not in a switch *(error)*
The program used a **default** label outside a **switch** construct. See the Lexicon entry for **default**.

divide by zero *(warning)*
The program will divide by zero if this code is executed. Although the program can be parsed, this statement may create trouble if executed.

duplicated case constant *(error)*
A **case** value can appear only once in a **switch** statement. See the Lexicon entries for **case** and **switch**.

empty switch *(warning)*
A **switch** statement has no **case** labels and no **default** labels. See the Lexicon entry for **switch**.

error in enumeration list syntax *(error)*
The syntax of an enumeration declaration contains an error.

error in expression syntax *(error)*
The parser expected to see a valid expression, but did not find one.

exponent overflow in floating point constant *(warning)*
The exponent in a floating point constant has overflowed. The compiler has set the constant to the maximum allowable value, with the expected sign.

exponent underflow in floating point constant *(warning)*
The exponent in a floating point constant has underflowed. The compiler has set the constant to zero, with the expected sign.

expression too complex *(fatal)*
The code generator cannot generate code for an expression. You should simplify your code.

external syntax *(error)*
This could be one of several errors, most often a missing '{'.

file ends within a comment *(error)*
The source file ended in the middle of a comment. If the program uses nested comments, it may have mismatched numbers of begin-comment and end-comment markers. If not, the program began a comment and did not end it, perhaps inadvertently when dividing by **something*, e.g., **a=b/*cd;**.

function cannot return a function *(error)*
The function is declared to return another function, which is illegal. A function, however, can return a *pointer* to a function, e.g., **int (*signal(n, a))()**

function cannot return an array *(error)*
A function is declared to return an array, which is illegal. A function, however, can return a pointer to a structure or array.

functions cannot be parameters *(error)*
The program uses a function as a parameter, e.g., **int q(); x(q);**. This is illegal.

identifier "*string*" is being redeclared *(error)*
The program declares variable *string* to be of two different types. This often is due to an implicit declaration, which occurs when a function is used before it is explicitly declared. Check for name conflicts.

LEXICON

identifier "*string*" is not a label *(error)*
The program attempts to **goto** a nonexistent label.

identifier "*string*" is not a parameter *(error)*
The variable "*string*" did not appear in the parameter list.

identifier "*string*" is not defined *(error)*
The program uses identifier *string* but does not define it.

identifier "*string*" not usable *(error)*
string is probably a member of a structure or **union** which appears by itself in an expression.

illegal character constant *(error)*
A legal character constant consists of a a backslash '\' followed by **a**, **b**, **f**, **n**, **r**, **t**, **v**, **x**, or up to three octal digits.

illegal character (*number* decimal) *(error)*
A control character was embedded within the source code. *number* is the decimal value of the character.

illegal # construct *(error)*
The parser recognizes control lines of the form **#***line_number* (decimal) or **#***file_name*. Anything else is illegal.

illegal integer constant suffix *(error)*
Integer constants may be suffixed with **u**, **U**, **l**, or **L** to indicate **unsigned**, **long**, or **unsigned long**.

illegal label "*string*" *(error)*
The program uses the keyword *string* as a **goto** label. Remember that each label must end with a colon.

illegal operation on "void" type *(error)*
The program tried to manipulate a value returned by a function that had been declared to be of type **void**.

illegal structure assignment *(error)*
The structures have different sizes.

illegal subtraction of pointers *(error)*
A pointer can be subtracted from another pointer only if both point to objects of the same size.

illegal use of a pointer *(error)*
A pointer was used illegally, e.g., multiplied, divided, or &-ed. You may get the result you want if you cast the pointer to a **long**.

illegal use of a structure or union *(error)*
You may take the address of a **struct**, access one of its members, assign it to another structure, pass it as an argument, and return. All else is illegal.

illegal use of floating point *(error)*
A **float** was used illegally, e.g., in a bit-field structure.

illegal use of "void" type *(error)*
The program used **void** improperly. Strictly, there are only **void** functions; COHERENT also supports the cast to **void** of a function call.

illegal use of void type in cast *(error)*
The program uses a pointer where it should be using a variable.

inappropriate signed *(error)*
The **signed** modifier may only be applied to **char**, **short**, **int**, or **long** types.

inappropriate "long" *(error)*
Your program used the type **long** inappropriately.

inappropriate "short" *(error)*
Your program used the type **short** inappropriately.

inappropriate "unsigned" *(error)*
Your program used the type **unsigned** inappropriately.

indirection through non pointer *(error)*
The program attempted to use a scalar (e.g., a **long** or **int**) as a pointer. This may be due to not dereferencing the scalar.

initializer too complex *(error)*
An initializer was too complex to be calculated at compile time. You should simplify the initializer to correct this problem.

integer pointer comparison *(strict)*
The program compares an integer or **long** with a pointer without casting one to the type of the other. Although this is legal, the comparison may not work on machines with non-integer size pointers, e.g., Z8001 or LARGE-model on the i8086 family, or on machines with pointers larger than **int**s, e.g., the M68000 family of microprocessors.

integer pointer pun *(strict)*
The program assigns a pointer to an integer, or vice versa, without casting the right-hand side of the assignment to the type of the left-hand side. For example,

```
char *foo;
long bar;
foo = bar;
```

Although this is permitted, it is often an error if the integer has less precision than the pointer does. Make sure that you properly declare all functions that returns pointers.

internal compiler error *(fatal)*
The program produced a state that should not happen during compilation. Try to localize the offending statement if at all possible. Forward a minimal program that exhibits the error, preferably on a machine-readable medium, to Mark Williams Company, together with the version number of the compiler, the command line used to compile the program, and the system configuration. For immediate advice during business hours, telephone Mark Williams Company technical support.

"*string*" is a enum tag *(error)*
"*string*" is a struct tag *(error)*
"*string*" is a union tag *(error)*
string has been previously declared as a tag name for a **struct**, **union**, or **enum**, and is now being declared as another tag. Perhaps the structure declarations have been included twice.

"*string*" is not a tag *(error)*
A **struct** or **union** with tag *string* is referenced before any such **struct** or **union** is declared. Check your declarations against the reference.

"*string*" is not a typedef name *(error)*
string was found in a declaration in the position in which the base type of the declaration should have appeared. *string* is not one of the predefined types or a **typedef** name. See the Lexicon entry on **typedef** for more information.

"*string*" is not an "enum" tag *(error)*
An **enum** with tag *string* is referenced before any such **enum** has been declared. See the Lexicon entry for **enum** for more information.

class "*string*" [*number*] is not used *(strict)*
Your program declares variable *string* or *number* but does not use it.

label "*string*" undefined *(error)*
The program does not declare the label *string*, but it is referenced in a **goto** statement.

left side of "*string*" not usable *(error)*
The left side of the expression *string* should be a pointer, but is not.

lvalue required *(error)*
The left-hand value of a declaration is missing or incorrect. See the Lexicon entries for **lvalue** and **rvalue**.

member "*string*" is not addressable *(error)*
The array *string* has exceeded the machine's addressing capability. Structure members are addressed with 16-bit signed offsets on most machines.

member "*string*" is not defined *(error)*
The program references a structure member that has not been declared.

mismatched conditional *(error)*
In a '?:' expression, the colon and all three expressions must be present.

misplaced ":" operator *(error)*
The program used a colon without a preceding question mark. It may be a misplaced label.

missing "(" *(error)*
The **if**, **while**, **for**, and **switch** keywords must be followed by parenthesized expressions.

missing "=" *(warning)*
An equal sign is missing from the initialization of a variable declaration. Note that this is a warning, not an error: this allows COHERENT to compile programs with "old style" initializers, such as **int i 1**. Use of this feature is strongly discouraged, and it will disappear when the ANSI standard for the C language is adopted in full.

missing "," *(error)*
A comma is missing from an enumeration member list.

missing ":" *(error)*
A colon ':' is missing after a **case** label, after a default label, or after the '?' in a '?'-':' construction.

missing ";" *(error)*
A semicolon ';' does not appear after an external data definition or declaration, after a **struct** or **union** member declaration, after an automatic data declaration or definition, after a statement, or in a **for(;;)** statement.

missing "]" *(error)*
A right bracket ']' is missing from an array declaration, or from an array reference; for example, **foo[5**.

missing "{" *(error)*
A left brace '{' is missing after a **struct** *tag*, **union** *tag*, or **enum** *tag* in a definition.

missing "}" *(error)*
A right brace '}' is missing from a **struct**, **union**, or **enum** definition, from an initialization, or from a compound statement.

missing "while" *(error)*
A **while** command does not appear after a **do** in a **do**-**while()** statement.

missing label name in goto *(error)*
A **goto** statement does not have a label.

missing member *(error)*
A '.' or '->' is not followed by a member name.

missing right brace *(error)*
A right brace is missing at end of file. The missing brace probably precedes lines with errors reported earlier.

missing "*string*" *(error)*
The parser **cc0** expects to see token *string*, but sees something else.

missing semicolon *(error)*
External declarations should continue with ',' or end with ';'.

missing type in structure body *(error)*
A structure member declaration has no type.

multiple classes *(error)*
An element has been asigned to more than one storage class, e.g., **extern register**.

multiple types *(error)*
An element has been assigned more than one data type, e.g., **int float**.

nonterminated string or character constant *(error)*
A line that contains single or double quotation marks left off the closing quotation mark. A newline in a string constant may be escaped with '\'.

number has too many digits *(error)*
A number is too big to fit into its type.

only one default label allowed *(error)*
The program uses more than one **default** label in a **switch** expression. See the Lexicon entries for **default** and **switch** for more information.

out of tree space *(fatal)*
The compiler allows a program to use up to 350 tree nodes; the program exceeded that allowance.

parameter *string* is not addressable *(error)*
The parameter has a stack frame offset greater than 32,767. Perhaps you should pass a pointer instead of a structure.

potentially nonportable structure access *(strict)*
A program that uses this construction may not be portable to another compiler.

return type/function type mismatch *(error)*
What the function was declared to return and what it actually returns do not match, and cannot be made to match.

return(e) illegal in void function *(error)*
A function that was declared to be type **void** has nevertheless attempted to return a value. Either the declaration or the function should be altered.

risky type in truth context *(strict)*
The program uses a variable declared to be a pointer, **long**, **unsigned long**, **float**, or **double** as the condition expression in an **if**, **while**, **do**, or '?'-':'. This could be misinterpreted by some C compilers.

size of *string* overflows size_t *(strict)*
A string was so large that it overran an internal compiler limit. You should try to break the string in question into several small strings.

size of union "*string*" is not known *(error)*
A pointer to a **struct** or **union** is being incremented, decremented, or subjected to array arithmetic, but the **struct** or **union** has not been defined.

size of *string* too large *(error)*
The program declared an array or **struct** that is too big to be addressable, e.g., **long a[20000];** on a machine that has a 64-kilobyte limit on data size and four-byte **long**s.

sizeof truncated to unsigned *(warning)*
An object's **sizeof** value has lost precision when truncated to a **size_t** integer.

sizeof(*string*) set to *number (warning)*
The program attempts to set the value of *string* by applying **sizeof** to a function or an **extern**; the compiler in this instance has set *string* to *number*.

storage class not allowed in cast *(error)*
The program **cast**s an item as a **register**, **static**, or other storage class.

string initializer not terminated by NUL *(warning)*
An array of **char**s that was initialized by a string is too small in dimension to hold the terminating NUL character. For example, **char foo[3] = "ABC"**.

structure "*string*" does not contain member "*m*" *(error)*
The program attempted to address the variable *string.m*, which is not defined as part of the structure *string*.

structure or union used in truth context *(error)*
The program uses a structure in an **if**, **while**, or **for**, or '?:' statement.

switch of non integer *(error)*
The expression in a **switch** statement is not type **int** or **char**. You should cast the **switch** expression to an **int** if the loss of precision is not critical.

switch overflow *(fatal)*
The program has more than ten nested **switch**es.

too many adjectives *(error)*
A variable's type was described with too many of **long**, **short**, or **unsigned**.

too many arguments *(fatal)*
No function may have more than 30 arguments.

too many cases *(fatal)*
The program cannot allocate space to build a **switch** statement.

too many initializers *(error)*
The program has more initializers than the space allocated can hold.

too many structure initializers *(error)*
The program contains a structure initialization that has more values than members.

trailing "," in initialization list *(warning)*
An initialization statement ends with a comma, which is legal.

type clash *(error)*
The parser expected to find matching types but did not. For example, the types of **e1** and **e2** in **(x) ? e1 : e2** must either both be pointers or neither be pointers.

type of function "*string*" adjusted to *string (warning)*
This warning is given when the type of a numeric constant is widened to **unsigned**, **long**, or **unsigned long** to preserve the constant's value. The type of the constant may be explicitly specified with the **u** or **L** constant suffixes.

type of parameter "*string*" adjusted to *string (warning)*
The program uses a parameter that the C language says must be adjusted to a wider type, e.g., **char** to **int** or **float** to **double**.

type required in cast *(error)*
The type is missing from a cast declaration.

unexpected end of enumeration list *(error)*
An end-of-file flag or a right brace occurred in the middle of the list of enumerators.

unexpected EOF *(fatal)*
EOF occurred in the middle of a statement. The temporary file may have been corrupted or truncated accidentally. Check your disk drive to see that it is working correctly.

union "*string*" does not contain member *m (error)*
The program attempted to address the variable string *m*, which is not defined as part of the structure *string*.

write error on output object file *(fatal)*
cc could not write the relocatable object module. Most likely, your mass storage device has run out of room. Check to see that your disk drive or hard disk has enough room to hold the object module, and that it is working correctly.

zero modulus *(warning)*
The program will perform a modulo operation by zero if the code just parsed is executed. Although the program can be parsed, this statement may create trouble if executed.

Notes

If you see the message

```
Out of memory
```

when compiling, this probably means that your program has exhausted the buffer space available to it. Use the option **-T0** to force **cc** to write its temporary files on the disk.

Prior to COHERENT release 4.2, **cc** wrote its diagnostic messages to the standard output device. **cc** now writes its diagnostic messages to the standard error. You may need to modify any scripts that redirect the output of **cc**.

cc0 — Definition

cc0 is the parser for the COHERENT C compiler **cc**. It parses C programs using the method of recursive descent and translates the program into a logical tree format.

See Also

cc, cc1, cc2, cc3, cpp, Programming COHERENT

cc1 — Definition

cc1 is the code generator for the COHERENT C compiler. This phase generates code from the trees created by the parser, **cc0**. The code generation is table driven, with entries for each operator and addressing mode.

See Also

cc, cc0, cc2, cc3, cpp, Programming COHERENT

cc2 — Definition

cc2 is the optimizer/object generator phase of the COHERENT C compiler. It optimizes the code generated by **cc1**, and writes the object code. COHERENT uses multiple optimization algorithms. One optimizes jump sequences: it eliminates common code, optimizes span-dependent jumps, and removes jumps to jumps. The other function scans the generated code repeatedly to eliminate unnecessary instructions.

See Also

cc, cc0, cc1, cc3, cpp, Programming COHERENT

cc3 — Definition

cc3 is the output phase of the COHERENT C compiler. It writes a file of assembly language rather than a relocatable object module. This phase is optional; it allows you to examine the code generated by the compiler. To produce an assembly-language output of a C program, use the **-S** option on the **cc** command line. For example,

```
cc -S foo.c
```

tells **cc** to produce a file of assembly language called **foo.s**, instead of an object module.

See Also

cc, cc0, cc1, cc2, cpp, Programming COHERENT

CCHEAD — Environmental Variable

Append options to beginning of cc command line
export CCHEAD=*options*

The COHERENT compiler **cc** reads the environmental variables **CCHEAD** and **CCTAIL** before it begins its work. You can set these variables to hold the default options that you want the compiler always to use.

cc appends the options in **CCHEAD** to the beginning of its command line.

See Also

cc, CCTAIL, environmental variables

CCTAIL — Environmental Variable

Append options to end of cc command line
export CCTAIL=*options*

The COHERENT compiler **cc** reads the environmental variables **CCHEAD** and **CCTAIL** before it begins its work. You can set these variables to hold the default options that you want the compiler always to use.

cc appends the options in **CCTAIL** to the end of its command line.

See Also

cc, CCHEAD, environmental variables

cd — Command

Change directory
cd *directory*

The shell keeps track of the directory in which the user is currently working. If a command is not specified by a complete path name beginning with '/', the shell prefixes it with the name of the current working directory. **cd** changes the current working directory to *directory*. If no *directory* is specified, the directory named in the **$HOME** environmental variable becomes the current working directory.

See Also

commands, ksh, pwd, sh

CD-ROM — Overview

COHERENT support for read-only compact disk devices

The term *CD-ROM* stands for "compact disk — read-only memory". COHERENT supports a variety of CD-ROM devices, from which you can read files or play music.

Devices Supported

As of this writing, COHERENT supports three varieties of CD-ROM drives:

- Sony CD-ROM models CDU31A or CDU33A, plugged its own dedicated controller.
- Mitsumi CD-ROM models FX001, FX001 high speed, FX001D, or LU005, plugged into its own dedicated controller. Mitsumi model FX001 also is known to work when plugged into the CD-ROM port of the SoundblasterPro sound card; the other Mitsumi drives have not yet been tested with the Soundblaster Pro card.
- Any SCSI CD-ROM drive plugged into an Adaptec 1542 SCSI controller.
- Any SCSI CD-ROM drive plugged Seagate host adapter models ST01 or ST02.

Please note that the NEC SCSI CD-ROM is support for ISO file systems, but *not* for audio disks. That is because the NEC drive does not use a standard interface for audio disks.

To use the driver for the Sony CDU31A drive, you must build a kernel that contains the driver **cdu31**. Normally, this is done when you install or update COHERENT. To add the driver to the kernel after installation or updating, do the following:

- Log in as the superuser **root**.
- **cd** to directory **/etc/conf**.
- Execute script **cdu31/mkdev**. This script will walk you through the process of adding the driver to the kernel. If you are unsure of the answer to any question that the script asks you, select the default; in most instances, this is correct.
- Execute the command:

  ```
  /etc/conf/bin/idmkcoh -o coh.test
  ```

  This builds a new kernel called **coh.test**.
- Boot the new kernel, as described in the Lexicon entry **booting**.

To use the driver for the Mitsumi drive, you must build a kernel that contains the driver **mcd**. Normally, this is done when you install or update COHERENT. To add the driver to the kernel after installation or updating, do the following:

- Log in as the superuser **root**.
- **cd** to directory **/etc/conf**.
- Execute script **mcd/mkdev**. This script will walk you through the process of adding the driver to the kernel. If you are unsure of the answer to any question that the script asks you, select the default; in most instances, this is correct.

- Execute the command:

```
/etc/conf/bin/idmkcoh -o coh.test
```

  This builds a new kernel called **coh.test**.

- Boot the new kernel, as described in the Lexicon entry **booting**.

If your CD-ROM drive is attached to an Adaptec 1542 SCSI controller, you must modify the driver **hai** to support the drive. Do so as follows:

- Log in as the superuser **root**.
- **cd** to directory **/etc/conf**.
- Execute script **hai/mkdev**. This script will walk you through the process of configuring **hai** to support your SCSI devices. If you are already using **hai** to support a SCSI disk or SCSI tape, be sure that you do not alter how they are configured. If you are unsure of the answer to any question that the script asks you, select the default; in most instances, this is correct.
- Execute the command:

```
/etc/conf/bin/idmkcoh -o coh.test
```

  This builds a new kernel called **coh.test**.

- Boot the new kernel, as described in the Lexicon entry **booting**.

Reading a CD-ROM

COHERENT at present includes three commands for manipulating CD-ROMs: **cdview**, **cdv**, and **cdplayer**.

cdplayer lets you play audio CDs on your CD-ROM drive. It uses a text-based interface to let you display the contents of a CD, select a track, set the volume, and otherwise manipulate your audio CDs.

cdv is a script with which you can play CD-ROM disks — that is, disks that hold an ISO-9660 file system. The interface is character-based and rather crude; however, with it you can read the contents of a directory on a CD-ROM, or copy a file from the CD-ROM into a COHERENT directory. **cdview** is a lower-level command that is invoked through **cdv**.

Files

/dev/cdrom — Device applications read by default for CD-ROMs
/dev/rscd0 — Device for accessing Sony CDU31A CD-ROM
/dev/rmcd0 — Device for accessing Mitsumi CD-ROM
/dev/Scdrom* — Block-special SCSI CD-ROM devices
/dev/rScdrom* — Character-special SCSI CD-ROM devices

See Also

Administering COHERENT, cdplayer, cdrom.h, cdv, cdview, device drivers, hai, mcd

Notes

At present, you cannot mount an ISO-9660 file system onto your COHERENT system. A future release of COHERENT will permit you to do so.

Please note that COHERENT, like most UUCP-like operating systems, does not support playing audio CDs on a NEC/Toshiba CD-ROM. This is because NEC uses a non-standard interface for audio CDs.

cdmp — Command

Dump COFF files into a readable form
cdmp [-adlrs] *filename*

cdmp dumps a file in COFF format into its most readable format. Its default is to dump all information; but as this can produce a very large output file, **cdmp** lets you use the following switches to mix-and-match its output:

-a Suppress auxiliary symbol entries.

-d Suppress data dumps

-l Suppress line numbers.

-r Suppress relocation entries.

-s Suppress symbol entries.

cc and **as** do not produce line numbers and auxiliary-symbol entries, and **ld** does not preserve them.

cdmp writes its dump into the "vertical hexadecimal format," like that produced by the function **xdump()**. For example, the vertical hexadecimal dump of the string "hello world.\n" is:

```
0 hell o wo rld. .
  6666.6276.7662.0
  85CC.F07F.2C4E.A
```

The hexadecimal value of 'h' is 0x68, which appears vertically under the 'h'. The dump is broken into groups of four bytes; every unprintable character appears as '.'.

For details on **xdump()**, see the Lexicon entry for **libmisc**.

See Also

as, asfix, coff.h, commands, ld, libmisc

Notes

cdmp is an analogue of the UNIX command **cdump**.

cdplayer — Command

Play audio CDs
cdplayer [eject info pause play [*track***] resume skip stop volume** *level***]**

cdplayer gives you a text-based interface with which you can play audio compact disks (CDs) through a COHERENT CD-ROM device. It reads environmental variable **CD_DEVICE** for the name of the device to manipulate. If this variable is not set, by default **cdplayer** manipulates device **/dev/cdrom**.

cdplayer normally is invoked with one of the following commands. If you invoke it without a command (or with a command it does not recognize), it prints a usage message and exits. If an error occurs, **cdplayer** returns an exit status of one. **cdplayer** recognizes the following commands:

eject Eject the CD from the drive. Note that not every CD drive supports this feature; in particular, the Mitsumi model LU005 does not.

info Display information about the CD that is in the drive: the total number of tracks, total playing time, playing time per track, drive status, and track being played.

pause Pause the audio CD. Unlike the command **stop**, described below, **cdplayer** remembers the point at which playing stopped, and will resume playing at that point. If the CD is not playing, **cdplayer** ignores this command. To restart a paused CD, use the command **cdplayer resume**.

play [*track***]**
Play the CD, beginning at *track*. If no *track* is given, it begins as track one.

resume
Resume playing a paused CD. If the CD had not been paused, **cdplayer** ignores this command.

skip Skip to the next track. If the CD is on its last track, **cdplayer** returns it to its first track.

stop Stop playing this CD. If the CD is not being played, **cdplayer** ignores this command. The CD player "forgets" the point at which it had been playing the CD. To begin playing this CD again, use the command **cdplayer play**.

volume *level*
Set the CD drive's volume to *level*, which must be a number between 0 (softest) and 255 (loudest). Note that not every drive supports this feature.

Environment

CD_DEVICE — The CD-ROM device to manipulate.

See Also

CD-ROM, cdv, commands

Notes

cdplayer was written by Mark Buckaway (mark@datasoft.com) for the Linux operating system. Please direct comments concerning its COHERENT port to support@mwc.com. It is is distributed under the GNU Public License. Full source code for this program is available on the Mark Williams Bulletin Board and on other publically available systems.

cdrom.h — Header File

Definitions for CD-ROM drives
#include <sys/cdrom.h>

The header file **<sys/cdrom.h>** defines structures and IOCTLs used to manipulate CD-ROM drives.

See Also

CD-ROM, header files, ioctl

cdu31 — Device Driver

Driver for the Sony CD-ROM drives

cdu31 is a device driver for the Sony CD-ROM drive, models CDU31A and CDU33A. It has major-device number 14.

Normally, this device driver is included in the kernel when you install or update COHERENT. To configure this driver, log in as the superuser **root**, and execute script **/etc/conf/cdu31/mkdev**. Then run the command

```
/etc/conf/bin/idmkcoh -o coh.test
```

to build a test kernel that includes the driver.

Files

/dev/cdrom — Device applications read for CD-ROMs by default
/dev/rscd0 — Device for accessing CDU31A CD-ROM

See Also

CD-ROM, device drivers, hai

cdv — Command

Interface to CD-ROM devices
cdv [*directory***]**

The script **cdv** provides a easy-to-use interface to the set of commands that interrogate an ISO-9660 CD-ROM. It is designed to spare you the trouble of having to remember the names and syntax used by each of these commands. If you name a *directory* on its command line, **cdv** uses that directory within the CD-ROM's file system as its root file system; otherwise, it begins its work in the CD-ROM's default root directory. The advantage of this option is that CD-ROM file systems tend to hold many files, and reading the CD-ROM can be quite slow (depending upon the speed of your system and of your CD-ROM reader); making *directory* the root directory lessens the number of files **cdv** must paw through before it finds the material that interests you. Obviously, you must have some idea of the CD-ROM's contents before you can use this option.

After you invoke **cdv**, it displays the prompt:

```
Command:
```

Enter the command that you want **cdv** to execute, as follows:

cd *directory*
: Change directory. *directory* is the directory to enter. This can be a relative path name or absolute path name. As with the COHERENT command **cd**, you can use '.' and '..' as synonyms for, respectively, the current directory and the parent directory.

G *directory*
: Read the contents of *directory*.

g *file* Get *file*; copy it into the current directory.

N
n Because the contents of a CD-ROM's directory may not fit onto the screen, **cdv** lets you display a directory's contents one page at a time. These commands display the next page of the current directory's contents.

P
p Display the previous page of the current directory's contents.

Q
q Quit.

v *file* View *file*, which is on the CD-ROM. **cdv** displays *file* with the pager named in the environmental variable **$PAGER**. If this variable is not defined, it uses **more**.

! Invoke the shell. To return to **cdv**, type **exit**, to exit from the shell.

See Also

CD-ROM, cdview, commands

Notes

cdv was written by Chris Hilton.

cdview — Command

Read a file from a CD-ROM
cdview [*file***]**

The command **cdview** reads *file* from an ISO-9660 CD-ROM, and writes its contents to the standard output. If *file* names a directory on the CD-ROM, **cdview** writes its contents to the standard output.

cdview normally is used with the script **cdv**, which provides a kinder, gentler way to interrogate the device.

See Also

CD-ROM, cdv, commands

ceil() — Mathematics Function (libm)

Set numeric ceiling
#include <math.h>
double ceil(*z***) double** *z***;**

ceil() returns a double-precision floating-point number whose value is the smallest integer greater than or equal to *z*.

Example

The following example demonstrates how to use **ceil()**:

```
#include <errno.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define display(x) dodisplay((double)(x), #x)

dodisplay(value, name)
double value; char *name;
{
      if (errno)
            perror(name);
      else
            printf("%10g %s\n", value, name);
      errno = 0;
}
```

```
main()
{
      extern char *gets();
      double x;
      char string[64];

      for (;;) {
            printf("Enter number: ");
            if (gets(string) == NULL)
                  break;
            x = atof(string);

            display(x);
            display(ceil(x));
            display(floor(x));
            display(fabs(x));
      }
      putchar('\n');
}
```

See Also

abs(), fabs(), floor(), frexp(), libm
ANSI Standard §7.5.6.1
POSIX Standard, §8.1

cfgetispeed() — termios Macro (termios.h)

Get terminal input speed
#include <termios.h>
int cfgetispeed(*tty***)**
termios ******tty***;**

Macro **cfgetispeed()** returns the input speed of the terminal device. *tty* gives the address of a structure of type **termios**. It must have been initialized by a call to the **termios** routine **tcgetattr()**.

See Also

termios
POSIX Standard, §7.1.3

cfgetospeed() — termios Macro (termios.h)

Get terminal output speed
#include <termios.h>
int cfgetospeed(*tty***)**
termios ******tty***;**

Macro **cfgetospeed()** returns the input speed of the terminal device. *tty* gives the address of a structure of type **termios**. It must have been initialized by a call to the **termios** routine **tcgetattr()**.

See Also

termios
POSIX Standard, §7.1.3

cfsetispeed() — termios Macro (termios.h)

Set terminal input speed
#include <termios.h>
int cfsetispeed(*tty***,** *speed***)**
termios ******tty***;**
int *speed***;**

Macro **cfsetispeed()** sets the input speed of the terminal device.

tty gives the address of a structure of type **termios**. It must have been initialized by a call to the **termios** routine **tcgetattr()**. *speed* gives the speed to which the terminal device should be set. It must be one of the following constants:

B50	50 baud
B75	75 baud
B110	110 baud
B134	134.5 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19200 baud
B38400	38400 baud

You must call routine **tcsetattr()** for *tty* before this change can take effect.

See Also

termios
POSIX Standard, §7.1.3

cfsetospeed() — termios Macro (termios.h)

Set terminal output speed
#include <termios.h>
int cfsetospeed(*tty***,** *speed***)**
termios **tty***;*
int *speed***;**

Macro **cfsetospeed()** sets the output speed of the terminal device.

tty gives the address of a structure of type **termios**. It must have been initialized by a call to the **termios** routine **tcgetattr()**. *speed* gives the speed to which the terminal device should be set. It must be one of the following constants:

B50	50 baud
B75	75 baud
B110	110 baud
B134	134.5 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19200 baud
B38400	38400 baud

You must call routine **tcsetattr()** for *tty* before this change can take effect.

See Also

termios
POSIX Standard, §7.1.3

cgrep — Command

Pattern search for C source programs
cgrep [-clnsA] [-r *new***]** *expression file* **...**

cgrep is a string-search utility. It resembles its cousins **grep** and **egrep**, except that it is specially designed to be used with C source files. It checks all C identifiers against *expression* and prints all lines in which it finds a

match. **cgrep** allows you to search for a variable named 'i' without finding every 'if' and 'int' in your program. **cgrep** defines an "identifier" to be any variable name or C keyword. *expression* can be a regular expression; if it includes wildcard characters or '|'s, you must "quote it" to protect it against being modified by the shell. For details on the expressions that **cgrep** can recognize, see the Lexicon entry for **egrep**.

cgrep tests names that include the '.' and '->' operators against *expression*. Thus, to look for **ptr->val**, type:

```
cgrep "ptr->val" x.c
```

This finds **ptr->val** even if it contains spaces, comments, or is spread across lines. If it is spread across lines, it will be reported on the line that contains the last token. The only exception is if you include the **-A** option, in which case it will be reported on the line which contains the first token. This is to simplify MicroEMACS macros, as will be described below.

To find **structure.member**, type:

```
cgrep "structure\.member"
```

because '.' in a regular expression matches any character.

Do not include spaces in any pattern. Only identifiers and '.' or '->' between identifiers are included in the tokens checked for pattern-matching.

Command-line Options

cgrep recognizes the following command-line options:

-A Write all lines in which *expression* is found into a temporary file. Then, call MicroEMACS with its error option to process the source file, with the contents of the temporary file serving as an "error" list. This option resembles the **-A** option to the **cc** command, and lets you build a MicroEMACS script to make systematic changes to the source file. To exit MicroEMACS and prevent **cgrep** from searching further, **<ctrl-U> <ctrl-X> <ctrl-C>**.

-c Print all comments in each *file*. This form takes no expression.

-l List only the names of the files in which *expression* is found.

-n Prefix each line in which *expression* is found with its line number in the file.

-r Replace all expression matches with *new*. This option may not be used with any others, and it can only match simple tokens, not items like **ptr->val**. When **-r** is used and the input is **stdin**, a new file will always be created as **stdout**.

-s Print all strings in each *file*. This form takes no expression.

Examples

The command

```
cgrep tmp *.c
```

will find the variable name **tmp**, but not **tmpname**, or any occurrence of **tmp** in a string or comment.

The script

```
cgrep -c < myfile.c | wc -l
```

count the lines of comments in **myfile.c**.

The command

```
cgrep "x|abc|d" *.c
```

will find **x**, **ab**, or **d**. Note this is a regular expressions with a surrounding "^( )$" which is applied to every identifier. Thus, **reg*** will not match **register**, but **reg.*** will.

See Also

commands, egrep, grep, me

LEXICON

char — C Keyword

Data type

char is a C data type. It is the smallest addressable unit of data. According to the ANSI Standard, a **char** consists of exactly one byte of storage; a byte, in turn, must be composed of at least eight bits. **sizeof(char)** returns one by definition, with all other data types defined as multiples thereof. All Mark Williams compilers sign-extend **char** when it is cast to a larger data type.

Under COHERENT, a **char** by default is signed.

See Also

byte, C keywords, data formats, unsigned
ANSI Standard, §6.1.2.5

chase — Command

Highly amusing video game
/usr/games/chase [-c] [*speed* **]**

chase is a COHERENT version of a popular video game. It runs on the console with input from the console keyboard. **chase** assumes that the system console is a monochrome display adapter unless you select the **-c** color-display option.

To accomodate different computer system speeds and different levels of skill, **chase** prompts the user to type a speed when the game begins. Press **<return>** to try out the game with the default speed of ten; typing a higher number makes the game slower, a lower number makes it faster. If you can play at speed zero on a fast computer system, you play too many video games. If you know the speed you want, you can enter it as a command-line argument. If you see the boss coming, quit by pressing **<ctrl-C>**.

The Rules

The player (represented by a blinking shaded rectangle) attempts to evade four "ghosts" (represented by shaded rectangles with arrows) while erasing dots from the playing-board maze.

At the beginning of a game, the four ghosts are in the *ghost box* above the center of the maze and the player is below it. The maze is filled with dots, including four blinking diamonds called *power pellets*. The ghosts emerge from the ghost box and chase the player. The console arrow keys move the player left, right, up, or down through the maze. Typing '0' stops the player. The player continues to move in the same direction until a wall of the maze stops him, you type a '0', or you type another arrow key.

When the player eats a power pellet, he acquires super power and can chase the ghosts briefly; the ghosts change color while the player has super power. If the player catches a ghost, he scores a bonus and the ghost returns to the ghost box temporarily. Once a player eats all the dots on the board, the game continues at the next level.

The upper left corner of the screen displays a score and the current board level. Each dot the player eats scores ten points. The first ghost a player eats while he has super power scores 200 points, the second 400, the third 800, and the fourth 1,600. At certain times during the game, a bonus letter appears below the ghost box; the player scores 100 points for eating the bonus letter on level 'A', 300 on level 'B', 500 on level 'C', and so on.

The lower left corner of the screen displays the number of extra players remaining in the current game (initally two). Another bonus player appears every 10,000 points, to a maximum of three extra players. The game ends when the ghosts eat the last player.

See Also

commands

chdir() — System Call (libc)

Change working directory
#include <unistd.h>
chdir(*directory***) char** ******directory***;**

The *working directory* (or *current directory*) is the directory from which the search for a file name begins if a path name does not begin with '/'. By convention, the working directory has the name '.'. **chdir()** changes the working directory to the directory pointed to by *directory*. This change is in effect until the program exits or calls **chdir()** again.

See Also

cd, chmod(), chroot(), directory, libc, unistd.h
POSIX Standard, §5.2.1

Diagnostics

chdir() returns zero if successful. It returns -1 if an error occurred, e.g., that *directory* does not exist, is not a directory, or is not searchable.

check — Command

Check file system
check [-s] *filesystem ...*

check uses the commands **icheck** and **dcheck** to check the consistency of a file system. It acts on each argument *filesystem* in turn; it calls first **icheck** and then **dcheck** on each to detect problems.

If **-s** is specified, **check** attempts to repair any errors automatically. You should first unmount the file system, if possible. If the root device is involved, you should be in single-user mode and then reboot the system immediately (without typing **sync**).

See Also

clri, commands, icheck, ncheck, sync, umount

Notes

Certain errors, such as duplicated blocks, cannot be fixed automatically. Decisions must be made by a human.

In earlier releases of COHERENT, **check** acted upon a default file system if none was specified.

This command has largely been superceded by **fsck**.

checkerr — Command

Check the mail system for errors
/usr/lib/mail/checkerr

The script **checkerr** reads error reports that have been deposited into the error directory **/usr/spool/smail/error**. If it finds an error, **checkerr** concatenates them into file **/usr/spool/smail/.checkerror**, and mails that file to user **postmaster** on your system. If mail cannot be sent to **postmaster** for any reason, **checkerr** leaves the file in place; when you next invoke this command, it will again try to mail the error messages.

See Also

commands, mail [overview], smail

checklist — System Administration

File systems to check when booting COHERENT
/etc/checklist

The file **/etc/checklist** names all COHERENT partitions on your hard disk. COHERENT executes **fsck** for each file named in this file. This ensures that the file-system of each partition is checked and cleaned before it is mounted.

When you add a new COHERENT partition to your system, you should insert its name (that is, the name of its raw device) into **/etc/checklist** to ensure that its file system is checked at boot time.

See Also

Administering COHERENT, brc

chgrp — Command

Change the group owner of a file
chgrp *group file ...*

chgrp changes the group owner of each *file* to *group.* The *group* may be specified by a valid group name or a valid numerical group identifier.

Only the superuser may use **chgrp**.

Files

/etc/group — Convert group name to group identifier

See Also

chmod, chmog, chown, commands

chmod — Command

Change the modes of a file
chmod +*modes file*
chmod -*modes file*

The COHERENT system assigns a *mode* to every file, to govern how users access the file. The mode grants or denies permission to read, write, or execute a file.

The mode grants permission separately to the owner of a file, to users from the owner's group, and to all other users. For a directory, execute permission grants or denies the right to search the directory, whereas write permission grants or denies the right to create and remove files.

In addition, the mode contains three bits that perform special tasks: the set-user-id bit, the set-group-id bit, and the save-text or "sticky" bit. See the Lexicon entry for the COHERENT system call **chmod()** for more information on how to use these bits.

The command **chmod** changes the permissions of each specified *file* according to the given *mode* argument. *mode* may be either an octal number or a symbolic mode. Only the owner of a *file* or the superuser may change a file's mode. Only the superuser may set the sticky bit.

A symbolic mode may have the following form. No spaces should separate the fields in the actual *mode* specification.

[which] how perm ... [, ...]

which specifies the permissions that are affected by the command. It may consist of one or more of the following:

a All permissions, equivalent to **gou**
g Group permissions
o Other permissions
u User permissions

If no *which* is given, **a** is assumed and **chmod** uses the file creation mask, as described in **umask**.

how specifies how the permissions will be changed. It can be

= Set permissions
+ Add permissions
- Take away permissions

perm specifies which permissions are changed. It may consist of one or more of the following:

g Current group permissions
o Current other permissions
r Read permission
s Setuid upon execution
t Save text (sticky bit)
u Current user permissions
w Write permission
x Execute permission

Multiple *how*/*perm* pairs have the same *which* applied to them. One or more specifications separated by commas tell **chmod** to apply each specification to the file successively.

An octal *mode* argument to **chmod** is obtained by *OR*ing the desired mode bits together. For a list of the recognized octal modes, see the Lexicon entry for **chmod()**.

Examples

The first example below sets the owner's permissions to read + write + execute, and the group and other permissions to read + execute. The second example adds execute permission for everyone.

```
chmod u=rwx,go=rx file
chmod +x file
```

See Also

chgrp, chmod(), chmog, chown, commands, ls, stat, umask

chmod() — System Call (libc)

Change file-protection modes
#include <sys/stat.h>
chmod(*file***,** *mode***)**
char ******file***; int** *mode***;**

chmod() sets the mode bits for *file*. The mode bits include protection bits, the set-user-id bit, and the sticky bit.

mode is constructed from the logical OR of the mode constants declared in the header file **stat.h**, as follows:

S_ISUID	Set user identifier on execution
S_ISGID	Set group identifier on execution
S_ISVTX	Save file on swap device ("sticky bit")
S_IRUSR	Read permission for owner
S_IWUSR	Write permission for owner
S_IXUSR	Execute permission for owner
S_IRGRP	Read permission for members of owner's group
S_IWGRP	Write permission for members of owner's group
S_IXGRP	Execute permission for members of owner's group
S_IROTH	Read permission for other users
S_IWOTH	Write permission for other users
S_IXOTH	Execute permission for other users

For directories, some protection bits have a different meaning: write permission means files may be created and removed, whereas execute permission means that the directory may be searched.

The save-text bit (or "sticky bit") is a flag to the system when it executes a shared for of a load module. After the system runs the program, it leaves shared segments on the swap device to speed subsequent reinvocation of the program. Setting this bit is restricted to the superuser (to control depletion of swap space which might result from overuse).

Only the owner of a file or the superuser may change its mode.

See Also

creat(), libc, stat.h
POSIX Standard, §5.6.4

Diagnostics

chmod() returns -1 for errors, such as *file* being nonexistent or the invoker being neither the owner nor the superuser.

chmog — Command

Change mode, owner, and group simultaneously
chmog *mod own grp file ...*

chmog combines the functionality of the commands **chmod**, **chown**, and **chgrp** into one command. This lets you fine-tune the permissions on *file*s without having to type three separate commands.

The arguments *mode*, *own*, and *grp* give, respectively, the mode, owner, and group to which **chmog** sets *file*. Setting any of these three arguments '-' means that that feature of *file* is not changed. For example, the command

```
chmog - bin bin file_name
```

changes the owner and group of file **file_name** to **bin** and does not alter **file_name**'s permissions.

For details on how to set *mode*, *own*, and *grp*, see the Lexicon entries for, respectively, **chmod**, **chown**, and **chgrp**.

See Also

chgrp, chmod, chown, commands

chown — Command

Change the owner of files
chown *owner file ...*

chown changes the owner of each *file* to *owner*. The *owner* may be specified by valid user name or a valid numerical user id.

Only the superuser may use **chown**

Files

/etc/passwd — To convert user name to user id

See Also

chgrp, chmod, chmog, commands

chown() — System Call (libc)

Change ownership of a file
#include <unistd.h>
chown(*file, uid, gid***)**
char **file***file***; short** *uid, gid***;**

chown() changes the owner of *file* to user id *uid* and group id *gid.*

To change only the user id without changing the group id, use **stat()** to determine the value of *gid* to pass to **chown()**.

chown() is restricted to the superuser, because granting the ordinary user the ability to change the ownership of files might circumvent file space quotas or accounting based upon file ownership.

chown() returns -1 for errors, such as nonexistent *file* or the caller not being the superuser.

See Also

chmod(), libc, passwd, stat(), unistd.h
POSIX Standard, §5.6.5

chreq — Command

Change priority, lifetime, or printer for a job
chreq [-d*printer***] [-l***lifetime***] [-p***priority***]** *job*

The command **chreq** lets you change the printer, lifetime, and priority of a *job*, which identifies a print job spooled with the command **lp**. It recognizes the following options:

-d*printer* Move *job* to the queue for *printer*.

-l*lifetime* Change the lifetime of *job*, where *lifetime* is one of **T** (temporary), **S** (short-term), or **L** (long-term). Temporary lifetime means that a job "survives" in the spool directory for two hours after being spooled; short-term means that it survives 48 hours; and long-term that it survives for 72 hours. After a job's lifetime has expired, the print daemon **lpsched** removes it.

-p*priority* Change the despooling priority of *job* to *priority*, which is one of **0** (highest priority) to **9** (lowest priority). Jobs with high priority are printed before those with low priority. The default priority is **2**.

See Also

commands, lp, MLP_PRIORITY, printer

Notes

You can reset the default priority for print jobs by setting the environmental variable **MLP_PRIORITY**.

chreq is available only under COHERENT release 4.2 and subsequent releases.

chroot — Command

Change root directory
chroot *directory program ...*

The command **chroot** runs program *program* with root directory *directory*.

See Also

commands

Notes

Only the superuser **root** can use **chroot**.

chroot() — System Call (libc)

Change the root directory
#include <unistd.h>
int chroot(*path***)**
char **path***;*

The COHERENT system call **chroot()** changes the current process's root directory to that specified by *path*. Once the **chroot()** system call completes, all references to absolute directories (i.e., ones starting with '/') will actually refer to directory pointed to by *path*. It does not change the current directory.

chroot() is often used to add extra security to special or public login accounts.

See Also

chroot, libc

Notes

The process that invokes **chroot()** must be running as the superuser **root**, and *path* must name a valid directory.

chsize() — System Call (libc)

Change the size of a file
int chsize(*fd, size***);**
int *fd***; long** *size***;**

The COHERENT system call **chsize()** changes the size of the file associated with the file descriptor *fd* to be exactly *size* bytes long. If *size* is larger than the file's initial size, then **chsize()** pads the file with the appropriate number of extra bytes. If *size* is smaller than the initial size, then *chsize()* frees all allocated disk blocks between *size* and the initial size. The maximum file size as set by **ulimit()** is in force for calls to **chsize()**.

With a successful call, **chsize()** returns 0; otherwise, it returns -1 and sets **errno** to an appropriate value.

See Also

libc, open(), ulimit()

Notes

When you use **chsize()** to shorten a file, COHERENT frees all disk blocks beyond the new end-of-file mark. However, it does not zero out the bytes beyond the new end-of-file in the last allocated disk block. If you wish to obliterate a file, simply using **chsize()** to reset its size to zero will *not* do the trick.

When you use **chsize()** to lengthen a file, the new bytes beyond the initial size are simply those bytes that were in the final disk block beyond the original end-of-file marker. All additional bytes beyond that point are zeroes. The file system will not actually allocate new disk blocks to accomodate the new file size, but rather will create one or more sparse blocks.

The term *sparse block* refers to the fact that in the COHERENT file system, a disk block that would be all zeroes need not take up a physical disk block. Rather, COHERENT marks the i-node to indicate that the block is all zeroes, but does not allocate a physical block. This saves space on the disk.

A *sparse file*, is a file that contains one or more sparse blocks. The file system handles sparse files correctly; however, the command **fsck** may return the error message

```
Possible File Size Error
```

for them.

If you lengthen a file with **chsize()**, you may create a sparse file, which may in turn cause **fsck** to complain.

ckermit — Command

Interactive inter-system communication and file transfer
ckermit [-abcdefghiklpqrstwx] [*file ...* **]**

ckermit implements the **kermit** communications protocol. It lets you communicate with other systems via modem or network, and to exchange files with other systems that have also implemented the **kermit** protocol. Unlike the **kermit** command also included with the COHERENT system, **ckermit** uses an interactive shell to remove some of the pain from the process of exchanging files. The name **ckermit** relects the fact that this command is written in the C language, and so has been ported to many different machines and operating systems.

You can run **ckermit** in either *interactive mode* or *command mode*. Simply typing the command

```
ckermit
```

invokes **ckermit** in interactive mode: **ckermit** displays a prompt, waits for your command, executes, then prompts you for its next command. Typing the command line plus one or more arguments invokes **ckermit** in command mode: **ckermit** then reads the arguments from the command line and executes them. After execution of the commands, **ckermit** returns to interactive mode.

ckermit's command-line options name either actions or settings. An action option tells **ckermit** to send a file, receive a file, or connect to a remote system. The command line may contain no more than one action option. A settings option changes one or more of the internal values that control how **ckermit** operates; for example, one setting option lets you set the baud rate of the serial port that **ckermit** will be using. A command line can contain any number of settings options.

Command-Line Options

ckermit recognizes the following command-line options:

-a *filename* Give an alternate name to a file being transferred. For example, the command

```
ckermit -s foo -a bar
```

transmits the file **foo** to a remote system, but tells the remote system that the file is named **bar**. Likewise, the command

```
ckermit -ra baz
```

stores the first incoming file under the name **baz**.

If more than one file arrives or is sent, only the first file is affected by the **-a** option.

-b *baudrate* Set the baud rate of the device to *baudrate*.

-c Connect to serial port, and pass all subsequent typing to that port To resume talking to your local system, type the escape character followed by the letter 'c'. The escape character is set by default to **<ctrl-\>**, although you can change it if you wish.

-d Debug mode — record debugging information in the file **debug.log** in the current directory.

-e *n* Set the length of the packet to *n* where *n* is a number between ten and about 1,000. Lengths of 95 or greater require that the implementation of **kermit** on the remote system support the long-packet extension to the **kermit** protocol.

-f Send a "finish" command to a remote server.

-g *file* Ask a remote system to send *file* or *files*. The file name must use the remote system's own syntax; you must quote all characters normally expanded by the COHERENT shell, e.g.:

```
ckermit -g x\*.\?
```

-h Help — display a brief synopsis of the command-line options.

-i The "image" option: specify that the file being transmitted or received is an eight-bit binary file, and therefore no conversion should be performed upon the data being received.

-k Passively receive file or files, copying them to standard output.

-l *device* Name the serial device to be used. For example

```
ckermit -l /dev/com2l
```

tells **ckermit** to use device **/dev/com2l**.

-n Like **-c,** but used after a protocol transaction has occurred. You can use both **-c** and **-n** in the same command.

-p *x* Set parity, where *x* is one of **e**, **o**, **m**, **s**, or **n** (respectively, even, odd, mark, space, or none). If parity is other than none, then **ckermit** uses the eighth-bit prefixing mechanism to transfer binary data, provided the impementation of **kermit** on the remote system agrees. The default parity is none.

-q Quiet — suppress screen update during file transfer; for example, this lets you transfer a file in the background.

-r Receive a file or files. Wait passively for files to arrive.

-s *file* Send the specified *file* or *files*. If *fn* is '-' then **ckermit** sends from standard input, which may come from a file:

```
ckermit -s - < foo.bar
```

or come from a parallel process:

```
ls -l | ckermit -s -
```

You cannot use this mechanism to send text typed from the keyboard. To send a file named '-', precede it with a path name, e.g.:

```
ckermit -s ./-
```

-t Specify half duplex, line turnaround with XON as the handshake character.

-w Write-Protect — avoid file-name collisions for incoming files.

-x Begin server operation. This option can be used in either local or remote mode.

If **ckermit** is in local mode, shows the progress of the file transfer. A dot is printed for every four data packets; other packets are shown by type (e.g., 'S' for Send-Init); 'T' is printed when there's a timeout; and '%' is printed for each retransmission.

During file transfer, you can type the following "interrupt" commands:

<ctrl-F> Interrupt the current file and go on to the next, if any.

<ctrl-B> Interrupt the entire batch of files and terminate the transaction.

<ctrl-R> Resend the current packet.

<ctrl-A> Display a status report for the current transaction.

These interrupt characters differ from the ones used in other implementations of **ckermit** to avoid conflict with the COHERENT shell's interrupt characters.

Interactive Operation

When you invoke **ckermit** in interactive mode, it displays the following prompt.

```
C-Kermit>
```

Type any valid **ckermit** command; the set of valid commands is described below. **ckermit** executes the command and then prompts you for another. The process continues until you tell it to quit.

Commands begin with a keyword, normally an English verb, such as **send**. You can abbreviate any keyword, as long as you type enough characters to distinguish it from all other keywords. Certain commonly used keywords (e.g., **send**, **receive**, **connect**) have special non-unique abbreviations (respectively, 's', 'r', and 'c').

Certain characters have special functions in interactive commands:

LEXICON

? Print a message that explains what is possible or expected at the current point within a command. Depending upon the context, the message may be a brief phrase, a menu of keywords, or a list of files.

<esc> Request completion of the current keyword or file name, or insertion of a default value. **ckermit** will beep if the requested operation fails. **<tab>** does the same thing.

<del> Delete the previous character from the command. **<backspace>** does the same thing.

<ctrl-W> Erase the rightmost word from the command line.

<ctrl-U> Erase the entire command.

<ctrl-R> Redisplay the current command.

<space> Delimit fields (keywords, filenames, numbers) within a command.

<return> Execute the command.

**** Insert any of the above characters into the command, literally. To enter a literal backslash, type two backslashes in a row (\\). Typing one backslash immediately **<return>** lets you continue the command on the next line.

ckermit recognizes the following interactive commands:

! *command* Execute a shell command. A space must follow the **!**.

% A comment. **ckermit** ignores everything that follows the **%**.

bye Terminate and log out a remote **kermit** server.

close Close a log file.

connect Connect to the remote system.

cwd *directory*
Change the working directory to *directory*.

dial Dial a telephone number.

directory Display a directory listing.

echo Display arguments literally. Useful in take-command files.

exit Exit from the program, closing any open logs.

finish Instruct a remote **kermit** server to exit, but not log out.

get Get files from a remote **kermit** server.

hangup Hang up the telephone.

help Display a help message for a given command.

log Open a log file — debugging, packet, session, transaction.

quit Same as **exit**.

receive Passively wait for files to arrive.

remote Issue file-management commands to a remote **kermit** server.

script Execute a login script with a remote system.

send *file* Send *file* to the remote **kermit** server.

server Begin server operation.

set Set various internal parameters.

show Display values of parameters, program version, etc.

space Display current disk space usage.

statistics Display statistics about most recent transaction.

take Execute commands from a file.

Interactive **ckermit** accepts commands from files as well as from the keyboard. Upon startup, **ckermit** looks for the file **.kermrc** first in directory **$HOME** and then in the current directory; if it finds the file, it executes all commands it finds therein. These commands must be in interactive format. Command files may be nested to any reasonable depth.

The set Command

As noted above, the **set** command lets you set the internal parameters by which **ckermit** operates. The **set** command recognizes the following arguments:

block-check
Level of packet error detection.

delay Time to wait before sending first packet.

duplex Specify which side echoes during connect mode.

escape-character
Character to prefix *escape commands* during connect mode.

file Set various file parameters.

flow-control
Communication line full-duplex flow control.

handshake Communication line half-duplex turnaround character.

line Communication-line device name.

modem-dialer
Type of modem-dialer on communication line.

parity Communication line character parity.

prompt Change the **ckermit** program's prompt.

receive Set various parameters for inbound packets.

retry Set the packet retransmission limit.

send Set various parameters for outbound packets.

speed Communication line speed.

Remote Commands

ckermit also has a suite of commands that are sent to the remote system for execution. They are as follows:

cwd Change remote working directory (also, **remote cd**).

delete Delete remote files.

directory Display a listing of remote file names.

help Request help from a remote server.

host Issue a command to the remote host in its own command language.

space Display current disk space usage on remote system.

type Display a remote file on your screen.

who Display the users logged in to the remote system, or get information about a user.

Files

.kermrc — **ckermit** initialization commands

See Also

commands, kermit, uucp

LEXICON

Notes

The **kermit** protocol was developed at the Columbia University Center for Computing Activities. **ckermit** is copyright © by the Trustees of Columbia University.

On some remote systems, the command **hangup** does not hang up the telephone properly. If this occurs, add the following macro to file **$HOME/.kermrc**:

```
define myhangup sleep 2,output +++,sleep 2,output ATH0\13
```

This create a macro named **myhangup**, which you can invoke to hang up the remote telephone. To test the proper load of the macro, type the following at the **ckermit** prompt:

```
show macro myhangup
```

It should show the command sequence. If it is intact, you can execute the new **hangup** command by typing **myhangup**.

Please note that **ckermit** is provided in binary form per the licensing terms set forth by its copyright holders. It is distributed as a service to COHERENT customers, as is. It is not supported by Mark Williams Company. *Caveat utilitor.*

clear — Command

Clear the screen
clear

The command **clear** reads the **termcap** description of your terminal and uses the information therein to clear your terminal's screen. The environmental variable **TERM** must define your terminal's type.

See Also

commands, TERM, termcap

clearerr() — STDIO Function (libc)

Present stream status
#include <stdio.h>
clearerr(*fp***) FILE** ***fp*;**

clearerr() resets the error flag of the argument *fp*. If an error condition is detected by the related macro **ferror**, **clearerr()** can be called to clear it.

Example

For an example of this function, see the entry for **ferror()**.

See Also

ferror(), libc
ANSI Standard, §7.9.10.1
POSIX Standard, §8.1

clist.h — Header File

Character-list structures
#include <sys/clist.h>

The header file **clist.h** holds definitions useful to functions that manipulate character lists. It defines the character-list structure **CLIST** and the character-queue structure **CQUEUE**.

See Also

header files

clock — Device Driver

Read the system clock
/dev/clock

The file **/dev/clock** lets you read and set your system's clock. It is a part of the driver **mem**, which manages memory; thus, it has major number 0 and minor number 5.

The real time clock occupies the first 14 bytes of nonvolatile RAM (**/dev/cmos**). The difference between

/dev/cmos and **/dev/clock** is that the latter device locks the circuit during a read, so that the clock will not be updated as it is being read.

/dev/clock limits access to a 14-byte data area. Attempts to read or write beyond this limit will fail. **/dev/clock** stores the system time in binary-coded decimal (BCD). For details on BCD, see the Lexicon entry for **float**.

The COHERENT command **ATclock** reads this device and writes to it.

See Also

ATclock, cmos, device drivers, float

clock() — Time Function (libc)

Get processor time
#include <time.h>
clock_t clock();

The function **clock()** calculates and returns the amount of processor time a program has taken to execute to the current point. Execution time is calculated from the time the program was invoked. This, in turn, is set as a point from the beginning of an era that is defined by the implementation. Under COHERENT, time is recorded as the number of milliseconds since January 1, 1970, 0h00m00s GMT.

The value **clock()** returns is of type **clock_t**, which is defined in header file. **time.h**. If **clock()** cannot determine execution time, it returns -1 cast to **clock_t**.

To calculate the execution time in seconds, divide the value returned by **clock()** by the value of the macro **CLK_TCK**, which is also defined in **time.h**.

Example

This example measures the number of times a **for** loop can run in one second on your system. This is approximate because **CLK_TCK** can be a real number, and because the program probably will not start at an exact tick boundary.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main()
{
      clock_t finish;
      long i;

      /* finish = about 1 second from now */
      finish = clock() + CLK_TCK;
      for(i = 0; finish > clock(); i++)
            ;

      printf("The for() loop ran %ld times in one second.\n", i);
      return(EXIT_SUCCESS);
}
```

See Also

difftime(), libc, mktime(), time.h
ANSI Standard, §.12.2.1

close() — System Call (libc)

Close a file
#include <unistd.h>
int close(*fd***) int** *fd***;**

close() closes the file identified by the file descriptor *fd*, which was returned by **creat()**, **dup()**, **open()**, or **pipe()**. **close()** also frees the associated file descriptor.

Because each program can have only a limited number of files open at any given time, programs that process many files should **close()** files whenever possible. The function **exit()** automatically calls **fclose()** for all open files; however, the system call **_exit()** does not.

Example

For an example of this function, see the entry for **open()**.

See Also

creat(), libc, open(), unistd.h
ANSI Standard, §4.9.3
POSIX Standard, §6.3.1

Diagnostics

close() returns -1 if an error occurs, such as its being handed a bad file descriptor; otherwise, it returns zero.

closedir() — General Function (libc)

Close a directory stream
#include <dirent.h>
int closedir(*dirp***)**
DIR ******dirp***;**

The COHERENT function **closedir()** is one of a set of COHERENT routines that manipulate directories in a device-independent manner. It closes the directory stream pointed to by *dirp*.

closedir() returns zero if no error occurs. If something goes wrong, it returns -1 and sets **errno** to an appropriate value.

Example

For an example of this system call, see the Lexicon entry for **opendir()**.

See Also

dirent.h, getdents(), libc, opendir(), readdir(), rewinddir(), seekdir(), telldir()
POSIX Standard, §5.1.2

Notes

The COHERENT implementation of the **dirent** routines was written by D. Gwynn.

clri — Command

Clear i-node
/etc/clri *filesystem inumber ...*

clri zeroes out each i-node with *inumber* on *filesystem*. *filesystem* is almost always a device-special file that corresponds to a disk device, e.g., **/dev/rat0a** or **/dev/rsd1c**. The raw device should be used. For example, the command

```
/etc/clri /dev/rat0a 8250
```

clears i-node 8250 on the file system on device **/dev/rat0a**, which is the first partition on your first AT hard disk.

The user must have read and write permission on the *filesystem.* If the file that *inumber* identifies is open, then **clri** probably will not work as you expect: the system maintains in core memory a copy of all active i-nodes, and the kernel will eventually write this copy to disk, thus undoing the action of **clri**. To ensure that this does not happen, unmount the file system before you running **clri**. If the i-node is for the root file system, reboot the system immediately after you run **clri**.

See Also

commands, dcheck, fsck, icheck, i-node, umount

cmos — Device Driver

Device for reading CMOS

The file **/dev/cmos** the entry via which you can read your system's CMOS. It is a part of the driver **mem**, which manages memory; thus, it has major number 0 and minor number 3.

The CMOS is a special, non-volatile area of random-access memory (RAM) that holds information about your system's configuration. The following gives the common meanings assigned to the various byte positions within the CMOS area:

Real-time clock:

0x00 Seconds
0x01 Alarm, seconds
0x02 Minutes
0x03 Alarm, minutes
0x04 Hours
0x05 Alarm, hours
0x06 Day of the week
0x07 Day of the month
0x08 Month
0x09 Year
0x0A Update in progress

Diagnostic power byte:

0x0E Bit 7 — Chip lost power
Bit 6 — Bad checksum
Bit 5 — Bad configuration byte
Bit 4 — Bad memory size
Bit 3 — Bad hard-disk byte
Bit 2 — Bad time of day

Restart-status byte:

0x0F Reloaded when restarting, e.g., returning from protected mode

Floppy-disk drive, drives A and B:

0x10 Bits 7-4 — Drive A:
0 = no drive
1 = 360-kilobyte drive
2 = 1.2-megabyte drive
3 = 720-kilobyte drive
4 = 1.44-megabyte drive
Bits 3-0 — Drive B:
0 = no drive
1 = 360-kilobyte drive
2 = 1.2-megabyte drive
3 = 720-kilobyte drive
4 = 1.44-megabyte drive

Floppy-disk drive, drives C and D:

0x11 Bits 7-4 — Drive C:
0 = no drive
1 = 360-kilobyte drive
2 = 1.2-megabyte drive
3 = 720-kilobyte drive
4 = 1.44-megabyte drive
Bits 3-0 — Drive D:
0 = no drive
1 = 360-kilobyte drive
2 = 1.2-megabyte drive
3 = 720-kilobyte drive
4 = 1.44-megabyte drive

Hard-disk drive:

0x12 Bits 7-4 — First hard-disk drive
0 = No drive
1-3 = Type 1-15
F = Use contents of byte 19
Bits 3-0 — Second hard-disk drive
0 = No drive
1-3 = Type 1-15
F = Use contents of byte 1A

Configuration of equipment:

0x014 Bits 7-6 — Floppy disks
00 = one floppy-disk drive
01 = two floppy-disk drives
10 = three floppy-disk drives
11 = four floppy-disk drives
Bits 5-4 — Type of display
00 = EGA/VGA
01 = CGA 40×25
10 = CGA 80×25
11 = monochrome display
Bit 1 — floating-point coprocessor installed
Bit 0 — Floppy-disk drive present

Memory:

0x15-0x16 Amount of memory below one megabyte
0x17-0x18 Amount of memory above one megabyte

Type of hard disk:

0x19 Type of first hard disk. Read only when bits 7-4 of byte 0x12 equal 0xF.
0x21 Type of second hard disk. Read only when bits 3-0 of byte 0x12 equal 0xF.

Miscellaneous:

0x2E-0x2F Checksum for bytes 0x10 through 0x2D
0x30-0x31 Indicate memory size above one megabyte
0x32 Century byte (BCD)
0x33 Flag for power-on information:
Bit 7 — Top 128 kilobytes of RAM is installed (shadow RAM is available)
Bit 6 — First boot after running set-up routine

/dev/cmos limits access to a 256-byte data area. Any attempt to read or write beyond this limit will fail.

See Also

ATclock, clock, device drivers, RAM

Notes

If you want to read or set the real time clock, then you should use **/dev/clock** instead of **/dev/cmos**.

Vendor-specific information, e.g., your system's memory configuration, is often kept in the CMOS area at locations beyond those documented above. Therefore, writing to undocumented regions of the CMOS area is extremely unwise: your computer could subsequently refuse to boot up properly. *Caveat utilitor.*

cmp — Command

Compare bytes of two files
cmp [-ls] *file1 file2* **[***skip1 skip2***]**

The command **cmp** compares two files byte by byte for equality. *file1* and *file2* name the files to compare; the file name '-' indicates the standard input.

If **cmp** finds two bytes that differ, it prints the number of the byte at which the discrepancy occurs, then exits. If it encounters EOF on one file but not on the other, it prints the message:

```
EOF on filen
```

cmp recognizes the following command-line options:

-l Note each differing byte by printing the positions and octal values of the bytes of each file.

-s Print nothing, but return the exit status.

By default, **cmp** begins at byte 1 of each file. The optional arguments *skip1* and *skip2* are integer values that tell **cmp** to skip that many bytes for the corresponding file before it begins the comparison. For example, the command

```
cmp FOO BAR 35 40
```

tells **cmp** to skip the first 35 bytes of **FOO** and the first 40 bytes of **BAR** before it begins to compare them.

See Also

commands, diff, sh, zcmp

Diagnostics

cmd returns zero for identical files, one for non-identical files, and two for errors, e.g., bad command or inaccessible file.

coff.h — Header File

Format for COFF objects
#include <coff.h>

coff.h describes the Common Object File Format (COFF), which is the object format used by COHERENT 386.

What Is COFF?

In brief, COFF is the UNIX System V standard for file formats. It defines the formats for relocatable object modules, for executable files, and for archives.

A COFF file is built around three sections, or *segments*:

text This holds executable machine code. It is write protected — the operating system is forbidden to overwrite it. (This is why operating systems that use COFF or similar formats are said to run in "protected mode.")

data This holds initialized data, that is, the data that the program finds when it begins execution. The program can read and write into this segment.

bss This segment holds unitialized data. It is simply a mass of space that is initialized to zeroes. It is contiguous with the **data** segment. The term **bss** from the old IBM mainframe days, and stands for "block started by symbol".

Not all segments have to be included in every COFF file. Further, some implementations of COFF define their own segments that manipulate special features of the operating system or hardware.

The following describes the structure of a COFF file. The areas within the file are described in the order in which they appear.

1. *file header*
This holds information set when the file was created, such as the date and time it was created, the number of segments in the file, a pointer to the symbol table, and status flags.

2. *optional header*
This gives information set at run-time, such as the address of the program entry point, and the size of the code and data segments.

3. *segment headers*
The next area holds a header for each segment in the file. Each header describes its segment's characteristics and contains pointers to the segment's contents, relocation information, line-number information, and other useful addresses.

4. *segment contents*
The next area holds the contents of the segments used in this file.

5. *relocation information*
The fifth area gives relocation information, one set of information for each segment in the file. The linker **ld** uses this information to generate the executable file at link time.

6. *line-number information*
This area holds debug information, one set of information for each segment. This area is optional.

7. *symbol table*
This area holds information used by both the linker and the debugger.

8. *string table*
This table holds very long names of variables.

Most of this information is irrelevant to the average user, or even the average developer of software. To the average user, COFF is "a machine that would go of itself"; you can run or compile programs without worrying what the linker puts where, or why. These details, however, can be very important if you are writing tools that manipulate the internals of files, such as archivers or debuggers. If you need detailed information on COFF and how to manipulate it, see *Understanding and Using COFF* (citation appears below).

For more information on how the COFF format affects COHERENT's language tools, see the Lexicon articles for **ar**, **as**, **cc**, **db**, and **ld**.

See Also

ar, as, cc, cdmp, coffnlist(), file formats, header files, ld
Gircys, G.R.: *Understanding and Using COFF*. Sebastopol, Calif., O'Reilly & Associates, Inc., 1988.

coffnlist() — General Function (libc)

Symbol table lookup, COFF format
#include <coff.h>
coffnlist(*fn***,** *nlp***,** *names***,** *count***)**
char **fn***fn***;**
SYMENT **nlp***nlp***;**
char **names***names***;**
int *count***;**

The function **coffnlist()** finds one or more names in the symbol table of a file in the COFF format.

You must arrange the names you seek into the form of a COFF symbol table. All long names (i.e., names longer than four characters) must be strung together like the COFF long-symbol-name section. Give each name an **n_type** of -1. After the call, any unfound names will still have this **n_type**, as a sign that it could not be found. Thus, you can use the same table to search several different COFF files.

fn points to the name of the file to be searched. *nlp* points to an array of type **SYMENT**. This structure is defined in header file **coff.h** as follows:

```
typedef     struct      syment      {
      union {
            char _n_name[SYMNMLEN]; /* Name */
            struct {
                  long _n_zeroes; /* If name[0-3] zero, */
                  long _n_offset; /* string table offset */
            } _n_n;
            char *_n_nptr[2];
      } _n;
      long        n_value;    /* Value */
      short       n_scnum;    /* Section number */
      unsigned short n_type;  /* Type */
      char        n_sclass;   /* Storage class */
      char        n_numaux;   /* Auxilliary entries */
#pragma align 2
} SYMENT;
#pragma align
```

count gives the number of symbols being sought. If there are long names, the displacement works from the *names* parameter.

Each item being sought must have 0xFFFF in its **n_type** field. This allows **coffnlist()** to be used on several files in order.

coffnlist() opens and reads the file pointed to by *fn*. It then scans the symbol table and tries to find a symbol with an **n_type** of 0xFFFF. Upon finding this entry, **coffnlist()** fills in the fields of the symbol entry.

coffnlist() returns zero if anything goes wrong, such as an inability to open the file *fn*. Otherwise, it returns one.

Example

The following example looks up three symbol names in the symbol table of file **tx.o**.

```
#include <stdio.h>
#include <coff.h>

main()
 {
      int i;
      static SYMENT sym[3];   /* the table of names to find */

      /* the long names section */
      static char long_names[] = "a_very_long_name";
      strcpy(sym[0].n_name, "x");   /* look up x */
      sym[0].n_type = -1;

      strcpy(sym[1].n_name, "y");   /* look up y */
      sym[1].n_type = -1;

      sym[2].n_zeroes = 0;    /* look up a_very_long_name */
      /* the long name table starts with a long giving its length
        * offsets are from the beginning of that long. Therefore
       * the n_offset of the first field is 4 not zero */
      sym[2].n_offset = sizeof(long);
      sym[2].n_type = -1;

      /* do lookups */
      if (!coffnlist("tx.o", sym, long_names, 3))
            exit(1);

      /* show off results */
      for (i = 0; i < 3; i++) {
            if (sym[i].n_type != -1)
                  printf("%s found at %x\n",
                        (sym[i].n_zeroes ? sym[i].n_name :
                              long_names + sym[i].n_offset - sizeof(long)),
                              sym[i].n_value);
      }
}
```

See Also

coff.h, libc, nlist()

coh_intro — Command

Tour the COHERENT file system
/etc/coh_intro [> *outfile* **]**

The command **coh_intro** walks you through the COHERENT file system. It gives you a brief introduction to each directory in the root file system, describes what it holds, and displays its contents.

This command is designed chiefly for a newcomer to COHERENT, to help teach her about the structure and operation of the COHERENT file system. An experience user may also wish to run **coh_intro** from time to time, in order to take a snapshot of her systems' current configuration.

See Also

commands

coherent.h — Header File

Miscellaneous useful definitions
#include <sys/coherent.h>

The header file **coherent.h** defines various useful types and objects. Among other things, it defines the structure **TIME**.

See Also

header files

COHERENT — Summary

Principles of the COHERENT System

This article describes COHERENT: its features, properties, and what sets it apart from other operating systems. It also gives tips on how to port an application to COHERENT, and describes how to un-install COHERENT from your system. For information on how COHERENT compares with MS-DOS, see the Lexicon article on **MS-DOS**.

What Is COHERENT?

COHERENT is a multiuser, multitasking operating system. *Multiuser* means that with COHERENT, more than one person can use your computer at any given time. *Multitasking* means that with COHERENT, any user can run more than one program at any given time. The design of COHERENT employs a few elegant concepts to give you a powerful and flexible system that is easy to use.

What is an Operating System?

An *operating system* is the master program that controls the operation of all other programs. It loads programs into memory, controls their execution, and controls a program's access to peripheral devices, such as printers, modems, and terminals.

Some operating systems permit only one user to use the computer at a time; and that user can run only one program at a time. However, you may well want your computer to support more than one user at a time, and run more than one program at a time. Sharing not only yields many economies (such as allowing a group of users to share one printer), but also allows the users to communicate with each other and so work together more efficiently.

Any multitasking operating system must be able to do the following tasks efficiently:

- Schedule computer time
- Control mass-storage devices (disks and tape drives)
- Organize disk-storage space
- Protect programs from conflict
- Protect stored information from destruction
- Ease cooperation among users

Today's operating systems also provide *tools*. These are programs that are bundled with the operating system, and that are designed to help you do your work more efficiently. For example, you need editors, compilers, debuggers, and assemblers to develop and test programs. Text formatters and spelling checkers help you write memoranda, manuals, or books. Command processors (also called *shells*) help you run the computer easily. Status checkers tell you what programs are being run, who is using the system, and how much space is left on your disk.

The combination of operating system and its tools transforms a boxful of wires and circuits into a useful machine.

COHERENT Documentation

This manual is designed to walk you through the COHERENT system. It consists of two parts: *tutorials* and *Lexicon*.

Each tutorial introduces a particular aspect of COHERENT. If you are a beginner, you should read the tutorials *Using the COHERENT System*, *Introducing*sh,*the*Bourne*Shell*, and *Introduction to MicroEMACS*. These will give you the basic information and basic skills you need to run COHERENT efficiently. A beginner who is interested in learning about the C language should look at the tutorial *The C Language*.

The tutorial *The make Programming Discipline* introduces the tool **make**. This tool is essential to building any complex tool under COHERENT. If you are going to be building tools under COHERENT, you should look at this tutorial.

The tutorial *UUCP, Remote Communications Utility* introduces UUCP. This bundle of programs lets your computer exchange mail and files with other computers, even if it is unattended. If you are all interested in networking with other computers (or plugging into the Internet), you should look at this tutorial.

The other tutorials introduce tools that are interest to advanced users.

The Lexicon fills the latter two thirds of this manual. It consists of more than 1,000 articles. The articles are printed in alphabetical order, to make it easy for you to find the one you want.

Most articles discuss a single aspect of the COHERENT system. Some articles, called "overview" articles, give a

broader discussion of an entire topic. Three overview articles are of particular interest:

Using COHERENT
: This article discusses the parts of COHERENT that are of interest to an ordinary user. It describes such matters as the commands available with COHERENT, and how a user can manage his own account.

Programming COHERENT
: This introduces the programming tools available under COHERENT; points to where you can find information about the COHERENT implementation of the C programming language; and points to where you can find information about the library routines and system calls that you can use in a program.

Administering COHERENT
: This article discusses how to administer COHERENT. It points to where you can find information on how to connect peripheral devices; manage mail and UUCP; change some of COHERENT's default behaviors; and modify and rebuild the COHERENT kernel. It also points to the articles that describe the files with which COHERENT manages itself.

If you cannot easily find an article that gives you the information you want, look in the index in the back of the manual. There is a good chance that you will find an entry there that points to the information you need. Also, you can use the command **apropos** to search the on-line version of the Lexicon for a key word that interests you. For details on this command, see its entry in the Lexicon.

How To Un-install COHERENT

To remove (or "un-install") COHERENT from your system, do the following:

1. Log in as the superuser **root**.
2. Invoke the COHERENT version of **fdisk**.
3. Choose the option to change all logical partitions. Don't change *any* parameters of any MS-DOS partitions.
4. Change *all* COHERENT partitions to type **Unused** with a size of 0, starting and ending at 0.
5. Exit **fdisk** and update the partition table.
6. Reboot the computer and run the MS-DOS **fdisk** utility to create a new MS-DOS partition table. Turn the unused space (formerly the COHERENT partitions) into an MS-DOS EXT partition. If you already have an MS-DOS EXT partition, change its parameters so that it incorporates the unused space.
7. Create one or more logical drives in the MS-DOS EXT partition.
8. Format the new logical drives using the MS-DOS **format** command.

Repeated tests with MS-DOS have shown that the above directions work. However, given the many flavors and releases of MS-DOS in circulation, Mark Williams Company cannot guarantee that the above steps will always work with MS-DOS. If they do not, consult your MS-DOS manual for creating a DOS partition table and file system on a new hard drive. If that information is not available, telephone Microsoft Technical Support at (206)454-2030.

Uninstalling the Mark Williams Bootstrap

The following describes how to remove the Mark Williams bootstrap program. You must do this if you are un-installing COHERENT from your system.

To remove the Mark Williams master boot program, you must overwrite the master boot-block on hard drive 0 with another boot program. Usually, this is the MS-DOS master boot program. Beginning with release 5.0, the MS-DOS version of **fdisk** has the switch **/mbr** that builds a new bootstrap program. All versions of the MS-DOS edition of **fdisk** writes a new master boot program if no valid signature appears at the end of the current contents of the master-boot block.

If you have MS-DOS version 5.0 or later, simply boot MS-DOS and run the command:

```
fdisk /mbr
```

If your version of MS-DOS predates release 5.0, you must modify the last two bytes of the master-boot block (to remove the magic "signature" that indicates a valid bootstrap program) then boot MS-DOS and run its version of **fdisk**.

Warning: See the note in the preceeding section about MS-DOS **fdisk** — back up your hard drive is backed up before you try this! There are several ways by which you can invalidate the signature at the end of the master-boot block. One way is to copy any sort of garbage into the master-boot block. You can (1) reformat cylinder 0 of your

hard drive — for example, using the **DIAGNOSTICS** menu of the AMI BIOS — or (2) use COHERENT to overwrite the block, e.g., with the command:

```
dd if=/coherent of=/dev/at0x count=1
```

The master-boot block is the first physical sector of the hard drive, i.e., cylinder 0, head 0, sector 1. (Note that numbering of sectors begins with one, not zero.) The MWC master bootstrap is part of the initial program load, and does not belong to any operating system because it runs before an operating system is loaded.

Please read the following carefully before you attempt erase the master-boot block:

Mark Williams Company can make no promises or guarantees concerning the behavior of any given version of the MS-DOS **fdisk**. Every version of the MS-DOS **fdisk** that we have tested does not recognize partitions allocated for other operating systems: MS-DOS cannot delete, or even display, such partitions. Certain configurations of empty partitions cause MS-DOS **fdisk** to hang.

Worst of all, don't expect *any* data on your hard drive to be available after MS-DOS **fdisk** rewrites an invalid master-boot block. Our experience is that MS-DOS **fdisk** erases all data in all partitions, even if previously existing MS-DOS partitions are re-allocated with identical cylinder ranges as at the time of their initial creation. *Caveat utilitor!*

cohtune — Command

Set a variable within a device driver
cohtune *driver tagfield* "*tagfield* **=** *value*"

The command **cohtune** sets the *tagfield* to *value* within the given device driver *driver*. You can then use the command **idmkcoh** to build a new kernel that incorporates your changes. When you boot the new kernel, your changes will have been made.

cohtune works by modifying the file **Space.c** for *driver*. Each device driver has such a file, that sets user-definable dimensions of its operation. When you invoke the command **idmkcoh** to build a new kernel, COHERENT automatically checks whether a **Space.c** module that have changed, compiles it, and links it into the newly built kernel. **idmkcoh** also recompiles every **Space.c** whenever you change a tunable variable in the kernel, just to ensure that all drivers are synchronized with changes in the kernel.

For example, the file **/etc/conf/hai/Space.c** gives the user-settable variable for the driver **hai**, which is COHERENT's host-adapter-independent SCSI driver. This file contains, among others, the variable **HAI_TAPE**. This variable is a bit-map; bit *n* is turned on if there is a SCSI tape device at SCSI ID *n*. If you have installed a SCSI tape as SCSI device 3, then type the following command:

```
cohtune hai "HAI_TAPE" "int HAI_TAPE = 0x08"
```

The value 0x08 turns on bit 3. As you can see, **cohtune** finds the line in **/etc/conf/hai/Space.c** that contains the string **HAI_TAPE** and is not commented out of the source, and replaces it with the line

```
int HAI_TAPE = 0x08
```

You can read a driver's **Space.c** to see how you can configure it. **Space.c** also gives some useful clues as to how the driver works and how it is currently configured.

You should *never* modify a **Space.c** by hand. If you do so, you run the risk of building a kernel that does not boot, or trashes your file system.

See Also

commands, device drivers, idenable, idmkcoh, idtune

Notes

cohtune cannot be used with STREAMS drivers.

Note that **cohtune** peforms no checks whatsoever on the content of the string with which you replace *tagfield* It should only be used by people familiar with C programming, because setting invalid values may cause errors that are difficult to diagnose. *Caveat utilitor.*

Because of the primitive nature of **cohtune**, we recommend that users not use it directly, but work instead through the configuration shell scripts supplied by the driver's developer (which typically live in directory **/etc/conf/***driver*) that can interactively generate the correct sequence of **cohtune** commands.

col — Command

Remove reverse and half-line motions
col [**-bdfx**][**-p***n*]

The command **col** reads the standard input and writes to the standard output. It removes reverse and half-line motions from the output of **nroff** for the benefit of output devices that cannot perform them. It maintains an image of the page in memory and performs these motions virtually so they do not appear on the output.

col understands four escape sequences: **<esc> 7** for reverse line feed, **<esc> 8** for half reverse line feed, **<esc> 9** for half forward line feed, and **<esc> B** for a forward line feed. It removes **<esc>** (ASCII 033) from the input stream if it is followed by any other character.

Eight control characters besides **<esc>** are interpreted by **col**. Newline, return, space, backspace, and tab carry their usual meaning. **VT** (013) is an alternate form of reverse line feed. The characters **SO** (017) and **SI** (016) signal the start and end of text in an alternate character set. **col** remembers the character set for each character and uses **SO** and **SI** to distinguish them on the output. **col** removes all other control characters from the input stream.

col recognizes the following options:

-b The output device cannot backspace. Only the last of a set of characters destined for a given position will appear.

-d Double-space the output. This doubles the length of a document but preserves relative vertical spacing. The **-f** option has precedence.

-f The output device can perform half-forward line feeds. Full lines appear single spaced with half lines between them. This is the only situation in which half forward line feeds appear in the output of **col** — reverse line motions never appear.

-x Suppress the default conversion of white space to tabs on output.

-p *n* Set the internal page buffer size to *n* full lines (default, 128).

If neither **-f** nor **-d** is chosen, **col** moves non-empty half lines to the next lower full line and pushes all later lines down one line. This can distort the appearance of the document.

See Also

ASCII, commands, nroff

Notes

Backing up past the start of a document or of the page buffer loses characters.

comm — Command

Print common lines
comm [-123] *file1 file2*

The command **comm** prints the lines unique to *file1* in the first column, the lines unique to *file2* in the second column, and the lines common to both in the third. Both *file1* and *file2* should be sorted in ASCII order. Any or all columns may be suppressed by indicating the column or columns to suppress in the optional flag. The file '**-**' means standard input.

See Also

cmp, commands, diff, sort, uniq

commands — Overview

The following lists the commands included with COHERENT. The command name is given on the left and a description on the right.

CD-ROM Commands

The following commands let you manipulate a CD-ROM device.

cdplayer . . . . . Play audio CDs
cdv . . . . . Interface to CD-ROM devices
cdview . . . . . Read a file from a CD-ROM

LEXICON

Communications

The following commands let you exchange information with other users and other systems.

ckermit . . . . . Interactive inter-system communication and file transfer
cu . . . . . UNIX-compatible interactive communications program
mail . . . . . Send/read electronic mail
mesg . . . . . Permit/deny messages from other users
msg . . . . . Send a brief message to other users
msgs . . . . . Read messages intended for all COHERENT users
uucico . . . . . Connect to a remote system
uucp . . . . . Copy a file to or from a remote system
wall . . . . . Send a message to all logged in users
write . . . . . Converse with another user

De-fragmentation Commands

The following commands give you information about the degree of fragmentation shown by a file system's free list. They can also rebuild a file system, to de-fragment it and so greatly the speed with which you can read and write it.

dpac . . . . . De-fragment a COHERENT file system
fmap . . . . . Measure fragmentation of the free list
qpac . . . . . Map the file system
spac . . . . . Sort a file system
upac . . . . . De-fragment a file system without sorting

Directory and File Handling

The following commands let you create, remove, and otherwise manipulate files and directories.

cat . . . . . Concatenate a file to the standard output
cd . . . . . Change directory
chgrp . . . . . Change the group owner of a file
chmod . . . . . Change the modes of a file
chmog . . . . . Change mode, ownership, and group of a file
chown . . . . . Change ownership of a file
cmp . . . . . Compare bytes of two files
compress . . . . . Compress a file
cp . . . . . Copy a file
cpdir . . . . . Copy directory hierarchy
dd . . . . . Convert the contents of a file
dos . . . . . Manipulate files on MS-DOS file systems
doscat . . . . . Concatenate a file on an MS-DOS file system
doscp . . . . . Copy files to/from an MS-DOS file system
doscpdir . . . . . Copy directories to/from an MS-DOS file system
dosdir . . . . . List the contents of an MS-DOS directory
dosdel . . . . . Delete a file from an MS-DOS file system
dosformat . . . . . Build an MS-DOS file system on a floppy disk
doslabel . . . . . Label an MS-DOS floppy disk
dosls . . . . . List files on an MS-DOS file system
dosmkdir . . . . . Create a directory in an MS-DOS file system
dosrm . . . . . Remove a file from an MS-DOS file system
dosrmdir . . . . . Remove a directory from an MS-DOS file system
fdisk . . . . . View/change hard-disk partitioning
file . . . . . Name a file's type
find . . . . . Search for files satisfying a pattern
gzip . . . . . GNU utility to compress files
gunzip . . . . . GNU utility to uncompress files
l . . . . . List directory's contents in long format
lc . . . . . List directory's contents in columnar format
lf . . . . . List directory's contents in columnar format
ln . . . . . Create a link to a file
lr . . . . . List subdirectorys' contents in columnar format
ls . . . . . List directory's contents
lx . . . . . List directory's contents in columnar format

mkdir . . . . . Create a directory
mv . . . . . Rename files or directories
mvdir . . . . . Rename a directory
pwd . . . . . Print the name of the current directory
qfind . . . . . Quickly find all files with a given name
rm . . . . . Remove files
rmdir . . . . . Remove directories
touch . . . . . Update modification time of a file
uncompress . . . . . Uncompress a file
unpack . . . . . GNU utility to uncompress files
unzip . . . . . Unzip a zipped archive
whereis . . . . . Locate source, binary, and manual files
which . . . . . Locate executable files
zcat . . . . . Concatenate a compressed file
zcmp . . . . . Compare compressed files
zforce . . . . . Force the suffix **.gz** onto every **gzip** file
znew . . . . . Recompress **.Z** files to **.gz** files

Editors

COHERENT includes a number of text editors, to suit a variety of tastes.

ed . . . . . Interactive line editor
elvis . . . . . Berkeley-style screen editor
emacs . . . . . COHERENT screen editor
ex . . . . . Berkeley-style line editor
me . . . . . COHERENT screen editor
sed . . . . . Stream editor
vi . . . . . Berkeley-style screen editor

Games

The following commands are just for fun.

almanac . . . . . Print an almanac entry for this date
banner . . . . . Print large sized letters
cal . . . . . Print a calendar
chase . . . . . Highly amusing video game
fortune . . . . . Print randomly selected, hopefully humorous, text
guess . . . . . Extraordinarily amusing guessing game
lines . . . . . Highly amusing board game
moo . . . . . Greatly amusing numeric guessing game
rubik . . . . . Play Rubik's cube
ttt . . . . . Three-dimensional tic-tac-toe

Kernel Tools

The following commands let you configure the COHERENT kernel, and build a new bootable kernel:

asypatch . . . . . Patch a kernel file for an asynchronous configuration
cohtune . . . . . Set a variable within a device driver
idbld . . . . . Reconfigure the COHERENT kernel
idenable . . . . . Enable or disable a device driver
idmkcoh . . . . . Build a new kernel
idtune . . . . . Set a tunable system value
patch . . . . . Patch a variable or flag within the kernel

Languages and Programming Tools

The COHERENT system comes with a number of languages, and tools for debugging and maintaining your programs.

as . . . . . Mark Williams assembler
asfix . . . . . Convert file to 80386 **as** form
awk . . . . . Report generation, pattern scanning, and processing language
cc . . . . . C-language compiler
cdmp . . . . . Dump COFF files into a readable form

LEXICON

conv . . . . . Numeric base converter
cpp . . . . . C preprocessor
db . . . . . Assembly-level symbolic debugger
ld . . . . . Link relocatable object files
lex. . . . . Lexical analyzer generator
m4 . . . . . Macro processor
make . . . . . Program building discipline
makedepend . . . . . Generate list of dependencies for a **makefile**
nm . . . . . Print a program's symbol table
od . . . . . Print an octal dump of a file
prof . . . . . Print execution profile of a C program
ref. . . . . Display a C function header
srcpath. . . . . Find source files
size . . . . . Print size of an object file
strip . . . . . Strip symbol tables from executable file
yacc. . . . . Parser generator

Libraries and Archives

The following commands help you create and read libraries and archives. These can be used as libraries (such as the libraries used when linking a C program), or to back up files.

ar . . . . . The object librarian/archiver
cpio . . . . . Archiving/backup utility
dump . . . . . File-system backup utility
dumpdate . . . . . Print dump dates
dumpdir . . . . . Print the directory of a dump
gnucpio . . . . . Archiving/backup utility
gtar . . . . . Archiving/backup utility
ranlib . . . . . Create index for object library
restor. . . . . Restore file system
tar. . . . . Archiving/backup utility

Mail

COHERENT comes with with a full-featured, UNIX-style mail facility based on the program **smail**. This is described in the overview article **mail**. The following commands perform mail-related work. Some are also listed in other sections of this article. Please note that the descriptions of **smail** and **rmail** are only for those users who wish to manipulate UUCP mailing on a low level; for most users, the descriptions under the command **mail** are more than sufficient.

checkerr . . . . . Check the mail system for errors
cvmail . . . . . Convert stored mail to System V format
getmap . . . . . De-archive Usenet map articles
lmail . . . . . Deliver local mail
mail. . . . . Send/read electronic mail
mailq . . . . . Display information about spooled mail
mkdbm. . . . . Build a data base for **smail**
mkfnames . . . . . Generate data base of user names
mkhpath. . . . . Build a **pathalias** data base from a **hosts** table
mkline . . . . . Fold mail data into one-line records
mkpath. . . . . Create a pathalias output file
mksort . . . . . Sort the standard input, allowing arbitrarily long lines
newaliases. . . . . Build the **aliases** data base from ASCII source
nptx. . . . . Generate permutations of users' full names
pathalias. . . . . Generate a set of paths among computers"
pathmerge. . . . . Merge sorted paths files
rmail . . . . . Receive mail
rsmtp. . . . . Run batched SMTP mail
runq. . . . . Periodically process the mail queue
savelog . . . . . Save a mail log
smail . . . . . Send mail
smtpd . . . . . SMTP daemon

For information on the configuration files used by the **smail** system, see the overview article **mail**, or the article **Administering COHERENT**.

Printing

The following commands help you print text. For commands that drive communications devices, e.g., modems, see the section on *Communications*, above.

cancel . . . . . Cancel a print job
chreq . . . . . Change priory, lifetime, or printer for a job
epson . . . . . Prepare a file for an Epson printer
fnkey . . . . . Set/print function keys for the console
hp . . . . . Prepare files for HP LaserJet-compatible printer
hpr . . . . . Send to LaserJet printer spooler
hpskip . . . . . Abort/restart current listing on LaserJet
lp . . . . . Spool a job for printing
lpadmin . . . . . Administer the **lp** print-spooler system
lpsched . . . . . Print jobs spooled with command **lp**
lpshut . . . . . Turn off the printer daemon
lpr . . . . . Send to line printer spooler
lpskip . . . . . Terminate/restart current line printer listing
lpstat . . . . . Give status of printer or job
pclfont . . . . . Prepare a PCL font for downloading via MLP
reprint . . . . . Reprint a spooled print job
route . . . . . Show or reset a user's default printer
stty . . . . . Set/print terminal modes

Shell Commands

COHERENT comes with two command interpreters, or *shells*: **ksh**, the Korn shell, and **sh**, the Bourne shell. The following commands are used either by the Korn shell, by the Bourne shell, or by both. Please note that commands used only by the Korn shell are marked by a dagger '†', whereas commands used only by the Bourne shell are marked by an asterisk '*'.

alias† . . . . . Set an alias
basename . . . . . Strip path information from a file name
bind† . . . . . Bind key sequence to editing command
break . . . . . Exit from shell construct
builtin† . . . . . Execute a command as a built-in command
case . . . . . Execute commands conditionally according to pattern
cd . . . . . Change directory
continue . . . . . Terminate current iteration of shell construct
dirname . . . . . Extract a directory name
dirs* . . . . . Print contents of directory stack
echo . . . . . Repeat an argument
eval . . . . . Evaluate arguments
exec . . . . . Execute command directly
exit . . . . . Exit from a shell
export . . . . . Add a shell variable to the environment
expr . . . . . Compute a command line expression
false . . . . . Unconditional failure
fc† . . . . . Edit and re-execute one or more previous commands
for . . . . . Execute commands for tokens in list
from . . . . . Generate list of numbers, for use in loop
getopts . . . . . Parse command-line options
hash† . . . . . Add a command to the shell's hash table
id . . . . . Print user and group IDs and names
if . . . . . Execute a command conditionally
jobs† . . . . . Print information about jobs
let . . . . . Evaluate an expression
nohup . . . . . Run a command while ignoring hangup signals
popd* . . . . . Pop an item from the directory stack
prep . . . . . Produce a word list
print† . . . . . Echo text onto the standard output

LEXICON

pushd* . . . . . Push an item onto the directory stack
read . . . . . Assign values to shell variables
readonly . . . . . Mark a shell variable as read only
set . . . . . Set shell option flags and positional parameters
shift . . . . . Shift positional parameters
sleep . . . . . Stop executing for a specified time
tee . . . . . Copy input to multiple output streams
test . . . . . Evaluate conditional expression
times . . . . . Print total user and system times
trap . . . . . Execute command on receipt of signal
true . . . . . Unconditional success
typeset† . . . . . Set/list variables and their attributes
umask . . . . . Set the file-creation mask
unalias† . . . . . Remove an alias
unset . . . . . Unset an environmental variable or shell function
until . . . . . Execute commands repeatedly
wait . . . . . Await completion of background process
whence† . . . . . List a command's type
while . . . . . Execute commands repeatedly
xargs . . . . . Execute a command with many arguments

String Processing

Some of the most useful commands are those that process strings. COHERENT has many commands that search for strings, manipulate strings, sort strings, and otherwise perform useful manipulations on strings.

c . . . . . Print multi-column output
cgrep . . . . . Pattern search for C programs
comm . . . . . Print common lines
cut . . . . . Select portions of each line of a file
detab . . . . . Replace tab characters with spaces
diff . . . . . Summarize differences between two files
diff3 . . . . . Summarize differences among three files
egrep . . . . . Extended pattern search
grep . . . . . Pattern search
head . . . . . Print the beginning of a file
join . . . . . Join two data bases
look . . . . . Find matching lines in a sorted file
more . . . . . Display text one screenful at a time
paste . . . . . Merge lines of files
rev . . . . . Print text backwards
scat . . . . . Print text files one screenful at a time
sort . . . . . Sort lines of text
split . . . . . Split a text file into smaller files
strings . . . . . Print all character strings from a file
tail . . . . . Print the end of a file
tr . . . . . Translate characters
tsort . . . . . Topological sort
uniq . . . . . Remove/count repeated lines in a sorted file
view . . . . . Berkeley-style text viewer
wc . . . . . Count words, lines, and characters in text files
zdiff . . . . . Compare two compressed files
zgrep . . . . . Search compressed files for a regular expression
zmore . . . . . Display compressed text one page at a time

System Accounting

The following commands help you to keep track of how your COHERENT system is working.

ac . . . . . Summarize login accounting information
accton . . . . . Enable/disable process accounting
df . . . . . Measure free space on disk
du . . . . . Summarize disk usage
hmon . . . . . Monitor the COHERENT System

ps . . . . . Print process status
sa . . . . . Print a summary of process accounting
quot. . . . . Summarize file-system usage
time. . . . . Time the execution of a command
times . . . . . Print total user and system times
uulog . . . . . Examine UUCP operations

System Maintenance

These commands help you to maintain your COHERENT system.

asymkdev . . . . . Create nodes for asynchronous devices
at . . . . . Execute commands at given time
bad . . . . . Maintain list of bad blocks
badscan . . . . . Examine a device for bad blocks
build . . . . . Install COHERENT onto a hard disk
check. . . . . Check file system
clri . . . . . Clear i-node
crontab. . . . . Copy a command file into the crontab directory
date. . . . . Print/set the date and time
dcheck . . . . . Check directory consistency
fdformat . . . . . Low-level format a floppy disk
fsck . . . . . Check and repair file systems interactively
icheck . . . . . i-node consistency check
mkfs . . . . . Make a new file system
mknod . . . . . Make a special file or named pipe
mount . . . . . Mount a file system
ncheck . . . . . Print file names corresponding to i-node
newgrp . . . . . Change to a new group
newusr . . . . . Add new user to COHERENT system
reboot . . . . . Reboot the COHERENT system
shutdown . . . . . Shut down the COHERENT system
sync. . . . . Flush system buffers
ttytype. . . . . Set default terminal types
umount . . . . . Unmount a file system
uuchk . . . . . Sanity-check the UUCP system

terminfo

COHERENT supports an implementation of **terminfo**, the terminal-description utility used under UNIX System V. (It also supports **termcap**, should you prefer to use that venerable, but still useful, system.) The following commands help support **terminfo**:

captoinfo . . . . . Convert termcap data to **terminfo** form
infocmp . . . . . De-compile a **terminfo** binary file
tic . . . . . Compile a **terminfo** description

Text Processors

These commands help you to create orderly, attractive printed text. For information on how to print the output of these commands, see the commands listed under *Device Handling*, above.

col. . . . . Remove reverse and half line motions
deroff. . . . . Remove text formatting control information
nroff . . . . . Text-formatting language
fmt . . . . . Adjust the length of lines in a file of text
fwtable . . . . . Build a font-width table from PCL or PostScript font
lcasep. . . . . Convert text to lower case
pr . . . . . Paginate and print files
prps. . . . . Paginate and print files on PostScript printers
PSfont . . . . . Cook an Adobe font into PostScript format
spell. . . . . Find spelling errors
troff. . . . . Extended text-formatting language
typo. . . . . Detect possible typographical and spelling errors

UUCP

The UUCP commands lets you form a network with other COHERENT or UNIX systems. Members of the network can grant each other permission to exchange mail and execute commands on each others' systems remotely and automatically, without having to be directed by a human being. The overview article **UUCP** describes the COHERENT UUCP facility in some detail. The following commands perform UUCP-related work; note that some of the commands listed here also are also listed in other sections of this article.

mwcbbs . . . . . Download files from the Mark Williams bulletin board
uuchk . . . . . Sanity-check the UUCP system
uucico . . . . . Connect to a remote system
uuconv . . . . . Convert UUCP configuration files into Taylor format
uucp . . . . . Copy a file to or from a remote system
uudecode . . . . . Decode a transmitted UUCP file
uuencode . . . . . Encode a UUCP file for tranmission
uuinstall . . . . . Configure UUCP control files
uumkdir . . . . . Create UUCP directories
uulog . . . . . Examine UUCP operations
uumvlog . . . . . Archive UUCP log files
uuname . . . . . Print names of recognized systems
uupick . . . . . Pick up a file uploaded from a remote system
uurmlock . . . . . Remove UUCP lock files
uusched . . . . . Call all systems that have jobs waiting for them
uuto . . . . . Send a file to a remote system
uutouch . . . . . Force polling of a remote site
uux . . . . . Execute a command on a remote system
uuxqt . . . . . Execute file as requested by remote system

Miscellaneous

The following commands do not fit neatly into any of the above categories. These include some of the more interesting and useful COHERENT commands, and are worth your attention.

apropos . . . . . Find manual pages on a given topic
ATclock . . . . . Read/set the AT realtime clock
bc . . . . . Interactive calculator with arbitrary precision
calendar . . . . . Electronic reminder service
chroot . . . . . Change root directory
clear . . . . . Clear your terminal's screen
coh_intro . . . . . Tour the COHERENT file system
crypt . . . . . Encrypt/decrypt text
dc . . . . . Desk calculator
disable . . . . . Disable a port
elvprsv . . . . . Preserve the modified version of a file after a crash
elvrec . . . . . Recover the modified version of a file after a crash
enable . . . . . Enable a port
env . . . . . Execute a command in an environment
factor . . . . . Factor a number
findmouse . . . . . Examine a port to see if a mouse is plugged into it
ftbad . . . . . Manipulate bad-block list on a floppy-tape cartridge
help . . . . . Print concise description of command
ideinfo . . . . . Display information about an IDE disk drive
install . . . . . Install a software update onto COHERENT
ipcrm . . . . . Remove an interprocess-communication memory item
ipcs . . . . . Display a snapshot of interprocess communications
kill . . . . . Signal a process
ksh . . . . . Invoke the Korn shell
login . . . . . Log in or change user name
makeboot . . . . . Make a bootable floppy disk
man . . . . . Display Lexicon entries
mklost+found . . . . . Make an enlarged **lost+found** directory
passwd . . . . . Set/change login password
phone . . . . . Print numbers and addresses from phone directory
script . . . . . Capture a terminal session into a file

sh . . . . . Invoke the Bourne shell
su . . . . . Substitute user id, become superuser
sum . . . . . Print checksum of a file
tape . . . . . Manipulate a tape device
tty . . . . . Print the user's terminal name
ttystat . . . . . Get terminal status
uname . . . . . Print information about the system
units . . . . . Convert units of measure
vsh . . . . . Invoke the COHERENT visual shell
who . . . . . Print who is logged in
yes . . . . . Print infinitely many responses

For more information on any of these commands, see its entry within the Lexicon.

See Also

Administering COHERENT, Programming COHERENT, Using COHERENT

compress — Command

Compress a file
compress [-dfvc] [-b*num* **] [** *file ...* **]**

compress compresses a file using the Lempel-Ziv algorithm. With text files and archives, it often can achieve 50% rate of compression.

If one or more *file*s are specified on the command, **compress** compresses them and appends the suffix **.Z** onto the end of each compressed file's name. If no *file* is specified on the command line, **compress** compresses text from the standard input and writes the compressed text to the standard output.

compress recognizes the following options:

-b The "bits" option. **compress** uses the compression level set via the *num* argument. Previous releases of **compress** would only allow values of *num* up to 12, with 12 being the default value if the **-b** option was not specified. The version of **compress** introduced with COHERENT version 3.1 handles values up to 16, with 12 being the default.

-c Send output to stdout.

-d Decompress rather than compress.

-f Force an output file to be generated even if no space is saved by compression.

-v Verbose mode: force **compress** to write statistics about its performance.

If you wish to ensure backwards compatibility with previous releases of COHERENT, do not use **compress** with a *num* value greater than 12.

See Also

commands, compression, gzip, ram, uncompress, zcat

compression — Technical Information

Programs used to compress text

Compression is the technique whereby a file is analyzed mathematically and made smaller. Compress is very useful in reducing the amount of disk space taken up by large files that you use infrequently.

The amount of compression will vary, depending upon the type of file being compressed, the compression algorithm used, and the level of compression requested. In general, files that show a great deal of repetition internally will compress more thoroughly than those that are largely random; thus, in general a text file will compress more thoroughly than will a digitized sound sample or image (although there are exceptions). The higher the level of compression you request, the more thoroughly the file will be compressed, but the longer the machine will have to work to achieve it. In most instances, raising the level of compression very high will save only a few bytes at a great cost in computer time.

You should note, too, that although compression algorithms try very hard not to lose information, it is possible that compressing some very complex files may result in a loss of information: that is, if you compress a file and de-compress it, the de-compressed file may be exactly the same as it was before you first compressed it. These programs will not affect most everyday varieties of data; but you should be aware of this fact.

COHERENT comes with the following tools for compressing and uncompressing files:

compress
This program compresses files uses the Lempel-Ziv algorithm. By default, it creates a file with the suffix **.Z**. It replaces the uncompressed original with its compressed analogue.

gtar This program creates tape archives. Its options **-z** and **-Z** invoke, respectively, the programs **gzip** and **compress** to compress the archive as it is being built, thus permitting you to build a compressed archive automatically.

gunzip This de-compresses files that had been compressed by the program **gzip**. By default, it works only with files that have the suffices **.z** or **.gz**. It replaces the compressed file with its uncompressed analogue.

gzip This program compresses files into the **zip** format. In general, it is faster and more thorough than **compress**, although it may not work as well on some files. It replaces the uncompressed original file with its compressed analogue.

uncompress
This uncompresses files that had been compressed by **compress**. It works with files that have the suffix **.Z**. It replaces the compressed file with its uncompressed analogue.

zcat This program de-compresses "on the fly" programs that had been compressed by **compress**, and writes the decompressed form to the standard output device. This is useful if you want to look at the contents of a compressed file but do not want to bother with de-compressing all of it.

Default Suffixes

Compressed files cannot be used in their compressed form; you must first uncompress them before you can use them. The key to uncompressing a compressed file is figuring out what program it was compressed with in the first place, so you can apply the correct de-compression tool.

If you have received a compressed file from a third-party source, you may have no idea what tool was used to compress the file; fortunately, however, most compression tools use standard suffices to "stamp" the files they compress. The following table gives commonly used suffices, plus examples of how to uncompress files that bear them:

Suffix	*Compression Program*	*Decompression Program*	*Example*
.Z	**compress**	**uncompress**	**uncompress foo.Z**
.tar.Z	**tar|compress**	**uncompress|tar**	**zcat foo.tar.Z | tar xvf -**
.z	**gzip**	**gunzip**	**gunzip foo.z**
.tar.z	**tar|gzip**	**gunzip|tar**	**gunzip foo.tar.z ; tar xvf foo**
.tgz	**gtar -cz**	**gtar -xz**	**gtar -xvzf foo.z**
.gtz	Same as **.tgz**		
.taz	Same as **.tgz**		

See Also

compress, gtar, gunzip, gzip, uncompress, Using COHERENT, zcat

con.h — Header File

Configure device drivers
#include <sys/con.h>

The header file **con.h** gives the configuration for each device driver included with the COHERENT system. Each driver is defined using the structure **CON**, which is declared in **<sys/con.h>**.

See Also

header files

config — System Administration

File that configures smail
/usr/lib/mail/config

File **/usr/lib/mail/config** holds instructions that configure the mailer-delivery program **smail**. You can modify this file to supplement, modify, or override **smail**'s default configuration.

Please note that this file is in no way related to file **/usr/lib/uucp/config**, which can be used to configure the Taylor UUCP system. For details on how to configure UUCP, see the Lexicon entry for **/usr/lib/mail/config**, which immediately follows this article in the Lexicon.

The rest of this article describes **config**, the attributes you can set within it, and how the setting of each attribute affects **smail**'s behavior.

Suite of Configuration Files

To begin, your machine can have two **smail** configuration files: a primary one, and a secondary one. Either can reset the values of any **smail** variable; for example, each can define names for the local host, define where files reside, or set the values for site-definable message-header fields. You are not obliged to use a configuration file: if **smail**'s default configuration suits you, then you can rename or move the primary configuration file so it will no longer be read. Likewise, if you have a primary configuration file, you are not obliged to have a secondary one.

smail reads the primary configuration file first, then the secondary configuration file. The values in the secondary configuration file can override those set in the primary file; the primary file, in turn, can redefine the name of the secondary configuration file. This gives you great flexibility to configure **smail** to suit your needs and preferences.

Format of a Configuration File

A configuration file consists of instructions; each instruction, in turn, sets an *attribute* to a value. Attributes come in three flavors: *string*, *numeric*, and *Boolean*. To set a variable to a string or numeric value, use the form:

variable **=** *value*

For example, the instructions

```
postmaster = tron@glotz.uucp
domains = wall.com
spool_mode = 0664
```

set the default address for the postmaster to **tron@glotz.uucp**, the attribute **domains** to **wall.com**, and the permissions for spool files to permit the file's owner and group to write into it.

Boolean attributes are either turned off or turned on. To turn on a Boolean attribute, use the notation:

+*boolean-attribute*

To turn it off, use the notation:

-*boolean-attribute*

You can also use the notation *-attribute* to set a numeric variable to zero and to un-set a value for a string variable. For example, the following instructions disable the use of an external transport file and tells **smail** that configuration files are not optional:

```
-transport_file
+require_configs
```

smail ignores blanks lines within a configuration file. If **smail** encounters a '#' character, it ignores that character plus all text to its right; thus, you can use this character to introduce a comment.

If a line begins with white space, **smail** assumes that it continues the previous line; in this way, you can extend an instruction over more than one line. For example, the following instructions set the **Received:** header field to use for messages to a multi-line value, and also set the name of a user that has few access capabilities:

```
# Use a verbose format for the Received: header field
received_field = "Received: by $primary_name
      with smail ($version_string)
      id <$message_id@$primary_name); $date"

nobody = unknown # comment: user "unknown" has few access capabilities
```

smail Attributes

The following names the attributes that you can set in a configuration file. Each attribute's name is followed by its type and its default setting in parentheses.

auth_domains (string, off)
: Name the domains for which your host is considered authoritative — i.e., the domains that your host knows how to access directly. The domain names must be separated by a single colon character ':'. Mail

addressed to any domain named in this list will not be forwarded to the smart host (described below).

auto_mkdir (Boolean, on)
If set, **smail** creates all directories required for spooling and logging if they do not exist. However, **smail** will never create required parent directories.

auto_mkdir_mode (integer, 0755)
When **smail** creates a directory, give it this permission mask. For details on what the numbers in a permission mask mean, see the Lexicon entry for **chmod**.

console (string, **/dev/console**)
Name the console device. This device is used as a last resort in attempting to write panic messages.

copying_file (string, **COPYING**)
The path name to file **COPYING**, which states your distribution rights and details the warranty information from the authors of **smail**. If this does not begin with '/', **smail** assumes that it is in the directory named by attribute **smail_lib_dir** (described below).

date_field (string, **Date: $spool_date**)
smail expands this string to form field **Date:** in a mail message's header, should the header not already contain such a field.

delivery_mode (string, **foreground**)
The default mode for delivering new mail. This can be one of the following values:

foreground
Immediate delivery via the process that received the message.
background
Immediate delivery via a child process. The process that received the message exits immediately.
queued
Do not attempt delivery until a later queue run.

director_file (string, directors)
This names the file that configures **smail**'s directors. If this does not begin with '/', **smail** assumes that it is in the directory named by variable **smail_lib_dir** (described below).

domains (string)
This sets the domain name that **smail** writes into the header of an outgoing mail message. It is computed at run time. If attribute **visible_name** is turned off, then **smail** sets it to the first name set by attribute **hostnames**. If **hostnames** is not set, then **smail** constructs the domain-name host names of the form *hostname.domain*. *hostname* is set in file **/etc/uucpname** *domain* is a name set by the attribute **domains**— **smail** uses each entry in **domains**, in order, to create the *hostnames* value.

For sites in the **UUCP** zone, **domains** often will merely be set to the string **uucp**. Finally, you can use the variable **$visible_name** within the string to which you set this attribute.

For compatibility with earlier versions of **smail**, this attribute can also be called **visible_domains**.

error_copy_postmaster (Boolean, off)
Send the postmaster a copy of every error message. Normally, **smail** sends the postmaster only the errors that appear to result from administrator mistakes. If you set this attribute, then **smail** also sends the postmaster the errors that are returned to the sender or that are mailed to owners of mailing lists.

fnlock_interval (number, 3)
Set the sleep interval between retries while attempting to lock mailbox files with a lockfile-based locking protocol. Under COHERENT, the function **sleep()** has a one-second granularity; therefore, you must this value to at least two.

fnlock_mode (number, 0666)
Create mailbox lock files.

fnlock_retries (number, five)
The number of times **smail** attempts to lock mailbox files using a file-based locking protocol.

from_field (string)
smail expands this string to form the fields **From:** and **Sender:** in a mail message's header. The expanded string must begin with **From:**, which may be replaced by other strings to form an actual header field. The default value is:

```
From: $sender${if def:sender_name: ($sender_name)}
```

grades (string)
Set the grade (or priority) characters that correspond to values of the **Precedence:** field in a mail message's header. The fields within the string are separated by ':'; precedence strings alternate with grade characters. Numbers have higher priority than upper-case letters, which in turn are higher than lower-case letters. Lower numbers are higher in priority than higher numbers, and the same goes for letters lower in the alphabet. Grades in the range 'a' through 'm' only return an error message and header to the sender when an error occurs. Grades in the range 'n' through 'z' return nothing to the sender should an error occur. The precedence names recognized by many BSD **sendmail** configurations are **special-delivery**, **first-class**, and **junk**. Others are useful mainly for getting mail out of the local machine or for communicating with other machines that run **smail** in a similar configuration. The grade character for a message is available in string expansions as the variable **$grade**. The default setting is:

```
special-delivery:9:air-mail:A:first-class:C:bulk:a:junk:n
```

hit_table_len (number, 241)
The length of the internal-address "hit" table. **smail** hashes addresses into this table to prevent multiple deliveries to the same address. Longer tables speed address hashing, at the price of a small increase in the amount of memory used. NB, this value may be ignored in the future.

host_lock_timeout (numeric, 30)
Set the time during which **smail** will attempt to lock a host's retry file; this file is used to guarantee exclusive delivery to that thost. If **smail** cannot lock the file within this time, then it leaves the message in the queue, to be delivered later.

A number with no suffix indicates seconds. Suffixes can be added to indicate a time multiplier: **m** indicates minutes, **h** indicates hours, and **d** indicates days.

hostnames (string)
hostname (string)
A colon-separated list of names for the local host. This list, together with the attributes **uucp_host** and **more_hostnames**, should represent all possible names for the local host. Note that **smail** does not recognize the name **hostname** as a name for the local host unless that name is also set by one of the other **hostname** variables. If your local host is in more than one domain or can gateway to more than one level of domains, then this attribute should represent those names. For a host in a registered domain in the UUCP zone, which is also in the maps distributed over USENET, **localhost.uucp** should also be in the list. The first value in **hostnames** is used internally as a special "primary name" for the local host.

Under COHERENT, this attribute is turned off by default. **smail** computes the value of **hostnames** by pairing the local host's name, as set in file **/etc/uucpname**, with every value set by attribute **domains**. **smail** re-computes the default value each time you run it.

lock_by_name (Boolean, on)
If this variable is turned on, locking of the input spool file is always based on lock files. Otherwise, an i-node—based locking mechanism may be used, such as the BSD function **flock()** or **lockf()** under System V or COHERENT. I-node—based locking is more efficient, if available. However, lock files can be easily created by shell scripts, which may be advantageous under some circumstances.

lock_mode (number, 0444)

log_mode (number, 0664)
The mode assigned to newly created mail-system log files.

logfile (string, **/usr/spool/smail/log/logfile**)
The file into which **smail** writes transaction messages and error messages. If this file does not exist, **smail** creates it with the mode set by variable **log_mode**.

max_hop_count (number, 20)
If the hop count for a message equals or exceeds this number, then any attempt at remote delivery results in an error message being returned to the sender. **smail** uses this mechanism to prevent infinite loops. To set the hop count for a specific message, use **smail**'s command-line option **-h**. Otherwise, **smail** computes it from the number of **Received:** fields in the message header.

max_load_ave (number)
For systems on which a load average can be computed, this attribute sets the maximum load average at which mail will be delivered. If the load average exceeds this number, **smail** saves incoming mail within the input spool directory for delivery later. Under COHERENT, this attribute is not set; therefore, **smail** does not compute the load average, and always attempts to deliver mail.

max_message_size (number, **100k**)
Set the maximum size of a message. **smail** truncates messages longer than this. (This is not yet implemented; at present, **smail** sets nolimit on the size of a message.)

message_buf_size (number, **100k**)
The size of the internal buffer that **smail** uses to read and write messages. The larger the value of this buffer, the fewer the number of calls to **read()** are required to read the message, because the entire message is always kept in memory. The default value is 100 kilobytes (**100k**).

message_id_field (string)
smail expands this attribute to form the field **Message-Id:** in a mail message's header. This will be used if such a field does not already exist in the header. The default value is:

```
Message-Id: <$message_id@$primary_name>
```

message_log_mode (number, 0644)
Each message has associated with it a unique file that contains a transaction log for that message. This number sets the permissions that **smail** gives this file when it creates it.

method_dir (string, **methods**)
If a method attribute for a router does not specify a path name that begins with '/', **smail** prefixes this directory onto the path to form the complete path for the method file. If this does not begin with '/', **smail** assumes that it is in the directory set by attribute **smail_lib_dir** (described below). See the description of the router file for more information on method files.

more_hostnames (string, off)
A colon-separated list of host names. These host names are in addition to any names that **smail** computed from the domains when forming the value of the variable **hostnames**. Thus, it is useful for specifying names that are not formed from the computed name for the host.

Attribute **more_hostnames** can also be called **gateway_names**, because it is often used to indicate the list of domains for which this machine is a gateway.

nobody (string, **nobody**)
The default user. This variables defines permissions to use when no other user is specified. Also, **smail** uses this user in some conditions when it is not certain whether a set of actions can be trusted, if performed under other, potentially more powerful users. This should reference a login identifier that has little power to do harm or access protected files.

paniclog (string, **/usr/spool/smail/log/paniclog**)
The name of the file onto which **smail** appends panic messages and other important error messages. If this file does not exist, **smail** creates it and assigns it the permissions specified by variable **log_mode**. **smail** records in this log all errors that require human intervention, such as configuration errors or directory-permission errors, that prevent mail spooling or delivery.

When a configuration error occurs, **smail** usually moves the mail into a special error directory under the input spool directory. This prevents **smail** from again attempting to delivery the message until the configuration error has been corrected.

Thus, you should regularly check both the panic log and the error directory, especially after you have changed a configuration. When the problem has been resolved, you can move the diverted messages back into the spool directory, and **smail** will again attempt to deliver them.

postmaster_address (string, **root**)
postmaster (string, **root**)
This attribute sets the default address of the postmaster. If the address **Postmaster** is not resolved by any of the configured directors, **smail** then uses this address.

qualify_file (string, **qualify**)
This variable names the file that contains the host-name qualification information. If this does not begin with '/', **smail** assumes that it is a subdirectory of the directory defined by the attribute **smail_lib_dir**.

queue_only (Boolean, off)
If this flag is set, then **smail** does not deliver incoming mail immediately. It only attempts delivery when it explicitly processes the input queue, such as when you invoke it with command-line option **-q**.

received_field (string)
smail expands this string to form the field **Received:** in a mail message's header. It inserts this field into the header if the "received" attribute is not explicitly turned off for a transport. The default value for **received_field** is:

```
received_field="Received: \
      ${if def:sender_host\
           {from $sender_host by $primary_name\
            ${if def:sender_proto: with $sender_proto}\
            \n\t(Smail$version #$compile_num) }\
      else {by $primary_name ${if def:sender_proto:with $sender_proto }\
            (Smail$version #$compile_num)\n\t}}\
      id $message_id; $spool_date"
```

require_configs (Boolean, off)
If this option is turned off or is not set, then **smail** does not require its configuration files to exist. This applies to the primary and secondary configuration files, and the director, router, and transport files (respectively, **/usr/lib/mail/directors**, **/usr/lib/mail/routers**, and **/usr/lib/mail/transports**). If one of these files does not exist, **smail** ignores it and instead uses its internally compiled configuration. If, however, you turn on this attribute, then if **smail** cannot find a configuration file whose file name is not null, it displays a panic message and exits.

To set a configuration file's name to null, turn off the attribute that names it. For example, to set the router file's name to null, use the attribute **-router**.

retry_file (string, **retry**)
This names the file that contains the retry-control information. If this name does not begin with '/', **smail** assumes that it is in directory named by variable **smail_lib_dir** (described below).

retry_duration (interval, **5d**)
This specifies the default period of time for which **smail** will attempt to deliver a message. If the message cannot be delivered within this period of time, **smail** assumes it is undeliverable, and sends a "bounce" message either to the sender or to the list's owner, should there be one. A number with no suffix indicates seconds. Suffices can be added to indicate a time multiplier: **m** indicates minutes, **h** indicates hours, and **d** indicates days. Under COHERENT, the default is five days.

retry_interval (interval, **10m**)
If **smail** cannot connect to a given host, it will wait at least this amount of time before it tries again. This applies to all messages routed to the host in question, to help process a queue efficiently.

return_path_field (string, **Return-Path: <$sender>**)
smail expands this string into field **Return-Path:** in the mail-message's header. It inserts this field into the header if attribute **return_path** is turned on for a given transport in file **/usr/lib/mail/transports**.

router_file (string, **routers**)
This attribute names the file that contains the router-configuration information. If this does not begin with '/', **smail** assumes that it is in the directory named by attribute **smail_lib_dir** (described below).

second_config_file (string, none)
This names the secondary configuration file. The section on configuration files, above, describes how this file relates to the primary configuration file. If this file's name does not begin with '/', **smail** assumes that it is in the directory named by attribute **smail_lib_dir** (described below).

This is primarily useful in networks whose machines share file systems. In particular, the attributes **smart_user**, **smart_path**, and **smart_transport** are set in the secondary configuration file.

sender_env_variable (string, not set)
This attribute names the environmental variable that, in turn, gives the name of the mail message's sender. Normally, the name of the sender is determined from her login identifier, or by checking calling process's real-user identifier. If **sender_env_variable** is set and the environmental variable it names exists, then **smail** uses that name by default. For example, if the line

```
sender_env_variable=BOGUS_NAME
```

appears in **/usr/lib/mail/config**, and if variable **BOGUS_NAME** is set in the user's environment, then **smail** uses that name to identify the sender, instead of the name for that user that appears in file **/etc/passwd**.

smail (string, **/bin/smail**)
This attribute names the **smail** binary. **smail** uses this to re-**exec** itself when a major configuration change has been detected, or to **exec smail** when delivering error messages. If this name does not begin with '/', **smail** assumes that this binary is kept in the directory named by attribute **smail_lib_dir**.

smail_lib_dir (string, **/usr/lib/mail**)
This attribute gives the full path name of the directory in which **smail** by default seeks its configuration files.

smail_util_dir (string, **/usr/lib/mail**)
This attribute gives the full path name of the directory that holds **smail**'s utilities, in particular the utilities **mkaliases** and **mkdbm**.

smart_path (string, not set)
This attribute defines the value that the router **smarthost** uses by default for its **path** attribute. It gives the path to a machine whose routing data base is more complete than the one on your local host. By default, this is not set; however, if you using UUCP to receive mail service from another system, you must set this variable to the name of that system. For details, see the Lexicon entry for **routers**.

smart_transport (string, not set)
This attribute defines the value that the **smarthost** router driver uses by default for its attribute **transport**. For details, see the Lexicon entry for **routers**.

smart_user (string, not set)
This attribute defines the value that the **smarthost** router driver uses by default for its attribute **smart_user**. For details, see the Lexicon entry for **routers**.

smtp_accept_max (number, 20)
This attribute sets the maximum number of SMTP connections that **smail** will process at any one time. This is for use with SMTP daemons started with **smail**'s command-line option **-bd**, or through the command **smtpd**. If **smail** receives a a connection request when this number of SMTP-connection children have already been forked, **smail** shuts down the connection with SMTP message 421. If this attribute is set to zero, then the number of SMTP connections is unlimited.

smtp_accept_queue (number, 5)
If this number of SMTP connection processes is exceeded, then **smail** accepts additional connections but queues their messages for later processing. When the number of current connection processes drops below this number, **smail** resumes the immediate processing of mail (if attribute **delivery_mode** is set to **foreground** or **background**.) If **delivery_mode** is set to zero, then **smail** will always process mail immediately, regardless of the number SMTP connections that it is handling. Note that the value of **smtp_accept_queue** should be less than the value of **smtp_accept_max**. Setting **smtp_accept_max** to zero prevents **smtp_accept_queue** from working correctly in all cases.

smtp_banner (string)
smail expands this string to the SMTP startup banner. **smail**'s SMTP server writes this banner when it accepts a connection request. Each line of this message is automatically preceded by identification code "220"; newlines are correctly changed into a carriage-return newline sequence. The default value for **smtp_banner** is:

```
$primary_name Smail$version #$compile_num ready at $date
```

smtp_debug (Boolean, on)
This Boolean variable controls the meaning of the **DEBUG** command when receiving SMTP commands. If this variable is on, then the **DEBUG** command (with an optional debugging level) sets debugging to the specified level, or to level 1 if no level was specified. **smail** writes the debugging output to the SMTP connection.

smtp_receive_command_timeout (interval, **5m**)
This attribute sets the time that **smail**'s SMTP daemon waits for a **receiver** command after it displays its prompt. If the daemon does not receive the command within this interval, it closes down the connection

and exits. The default is **5m**, that is, five minutes.

smtp_receive_message_timeout(interval, 2h)
This attribute sets the time that **smail** SMTP daemon waits for a message after it has displayed its prompt:

```
354 Enter mail
```

If it does not receive the entire message within this interval, it removes the message, closes the connection, and exits. The default is **2h**, that is, two hours.

spool_dirs (string, **/usr/spool/smail**)
This sets the directory or directories into which **smail** spools incoming mail. If it names more than one directory, the directories must be separated by a colon ':'. If **smail** cannot write a message to the first directory (say, due to permission problems, file-system-full errors, etc.), it tries to write the message into the other directories, one after another, until it either succeeds in writing the message or runs out of directories to try. Each spool directory is expected to have the following writable subdirectories:

input	The actual spool files
lock	Temporary lock files
msglog	Temporary per-message transaction logs and audit trails
error	Messages failing from problems requiring human intervention

spool_grade (character, **C**)
This attribute gives the default grade for mail messages. It can be overridden by a **Precedence:** field in a message's header. **smail** uses the grade to sort messages in the input spool directory. The grade is also available in string expansions as the variable **$grade**. See the description of the attribute **grades**, above, for more information.

spool_mode (number, **0440**)
This attribute sets the permissions **smail** gives to spooled files.

transport_file (string, **transports**)
This attribute names the file that holds the transport-configuration information. If the directory does not begin with '/', **smail** assumes it is in the directory named by the attribute **smail_lib_dir** (described above).

trusted_users (string, off)
This names the users who are trusted to specify a sender for a message. Users who are not in this list cannot specify a **Sender:** field in a mail header; if they do, **smail** removes it. If a trusted user specifies a **From:** header field, then **smail** also creates a **Sender:** field that names the real user who submitted the message.

In general, this attribute should name every user under whom remote mail is received and sent to **smail**. If this list is turned off, using the form **-trusted**, then every user is trusted.

NB, **smail** uses the real user identifier to verify a trusted user. However, the program **uucico** runs under the real user identifier of the user who invoked it — and any user can invoke **uucico**. **smail** cannot distinguish this case from any other, and thus will do the "wrong thing" in this instance. Under COHERENT, this attribute is turned off by default to avoid this problem.

trusted_groups (string, off)
This attribute names the user groups that are trusted to specify a sender of a message. **smail** checks a user's effective group identifier to ensure that he really is a member of a trusted group. Thus, were **smail** a **setgid** program, then this string would be of no value and should be turned off. However, if **smail** is not set gid (as it is not under COHERENT), then programs that invoke **smail** under a specific effective gid, not a specific real uid, can be detected and can be properly treated as trusted.

uucp_name (string)
This attribute gives the name of your local host. It is computed at run-time. This name is available in string expansions as the variable **$uucp_name smail** also uses it in the "remote from *hostname*" suffix to "From" lines for mail being delivered to remote machines, when the **from** attribute is turned on for a transport.

visible_domains (string)
This is a synonym for attribute **domains**.

See Also

Administering COHERENT, directors, mail [overview], routers, smail, transports

LEXICON

Notes

config — System Administration

File that configures UUCP
/usr/lib/uucp/config

The file **/usr/lib/uucp/config** performs overall configuration of the Taylor UUCP system. By setting commands within this file, you can override the default settings that are compiled into the COHERENT edition of UUCP.

Please note that this file is in no way related to file **/usr/lib/mail/config**, which configures **smail**, the mail-delivery program. For details on how to configure **smail**, see the Lexicon entry for **/usr/lib/mail/config**, which immediately precedes this article in the Lexicon.

Please note also that COHERENT does *not* include an edition of this file with its release of Taylor UUCP. That is because the default behaviors for COHERENT are already compiled into UUCP. However, you can create this file if you wish, and use it to change or override the default behaviors built into Taylor UUCP. This lets you customize UUCP to suit your needs and preferences, without having to modify or recompile the UUCP sources.

The rest of this article describes the commands that you can embed within **config**, should you wish to change the defaults for UUCP on your COHERENT system.

Miscellaneous Commands

The following **config** commands perform miscellaneous actions:

hdb-files true | false
If true, use HoneyDanBer configuration files instead of Taylor configuration files. COHERENT by default uses Taylor configuration files.

lockdir *directory*
Write lock files into *directory*. Under COHERENT, these files are written into **/usr/lib/uucp**.

max-uuxqts *number*
Set to *number* the maximum number of **uuxqt** processes that can run at any given time. The default is zero, which means that there is no limit.

nodename *name*
hostname *name*
uuname *name*
These commands are synonyms. Each tells UUCP to use *name* as the name of your system. Under COHERENT, your system's name is set in file **/etc/uuname**, and is returned by the system call **uname()**.

pubdir *directory*
Use *directory* as the publically accessible directory. Under COHERENT, the default public directory is **/usr/spool/uucppublic**.

run-uuxqt *string* | *number*
Specify when **uucico** should invoke **uuxqt**. If its argument is a number, **uucico** invokes **uuxqt** after it has received *number* execution files. If it is not a number, it must be one of the following strings:

once Invoke **uuxqt** once at the end of execution.
percall Invoke **uuxqt** once per call.
never Never invoke **uuxqt**.

Under COHERENT, the default is **once**.

spool *directory*
Use *directory* as the spool directory. Under COHERENT, the default spool directory is **/usr/spool/uucp**.

timetable *period time_string*
Define a time table to be used by default with subsequent **time** instructions. *period* is the period of day to which the time table applies. *time_string* is a standard time string that applies to that time of day. Taylor UUCP defines the following time tables by default:

```
timetable Evening Wk1705-0755,Sa,Su
timetable Night Wk2305-0755,Sa,Su2305-1655
timetable NonPeak Wk1805-0655,Sa,Su
```

unknown *string ...*
Let unknown systems log into your system. An "unknown," is one that is not described in **/usr/lib/uucp/sys**. Each *string* is applied to the unknown system, just as if it were named in **sys**. The COHERENT configuration of Taylor UUCP does not permit unknown systems to log in.

v2-files true|false
If true, use **V2**-style configuration files. COHERENT by default uses Taylor configuration files.

Configuration File Names

The following commands instruct Taylor UUCP to use configuration files other than the default ones:

callfile *file ...*
When dialing out, read the system name and password that your system passes to the remote system from each *file*. Taylor UUCP reads these files should the password or system name in a given system's description be set to '*'. Each line within a call file consists of three fields: the name of the remote system, the name by which your system identifies itself to the remote system, and the password. This mechanism permits you to make file **/usr/lib/uucp/sys** publically readable, while keeping the system names and passwords confidential.

COHERENT's default implementation of Taylor UUCP does not use call files, but you can set them up easily enough. Note that if you do so, pay careful attention to the permissions that you give each *file*.

dialcode *file ...*
Read dial codes from each *file*. "Dial codes" permits UUCP to interpret telephone numbers so they can be used through different telephone systems or area codes. COHERENT by default does not name or configure any dial-code files.

dialfile *file ...*
Read dialer-configuration information from every *file* instead of from the default file, **/usr/lib/uucp/dial**.

passwdfile *file ...*
Tell **uucico** to read system passwords from each *file*. This applies only to systems that are logging into your system, and only when **uucico** is managing the login process instead of the standard COHERENT programs. Each line in a *file* consists of two fields: the login name used by the remote system, and its password. **uucico** reads each file until it finds a password for the system that is attempting to log in.

Note that the COHERENT configuration of Taylor UUCP does not support encrypted passwords.

portfile *file ...*
Read port-configuration information from every *file* instead of from the default file, **/usr/lib/uucp/port**.

sysfile *file ...*
Read system-configuration information from every *file* instead of from the default file, **/usr/lib/uucp/sys**.

Log Files

The following commands let you change the log files that Taylor UUCP uses by default:

debugfile *file*
Write debugging information into *file* instead into the default file. Because COHERENT's port of Taylor UUCP uses HoneyDanBer logging instead of Taylor logging, **uucico** ignores this command. Under COHERENT, Taylor UUCP writes debugging information into **/usr/spool/uucp/.Admin/audit.local**.

logfile *file*
Write logging data into *file*. COHERENT's port of Taylor UUCP uses HoneyDanBer logging by default, which means that each system has its own log file within directory **/usr/spool/uucp/.Log**.

statfile *file*
Write statistics information into *file* instead of into the default file, **/usr/spool/uucp/.Admin/xferstats**.

Levels of Debugging

The COHERENT port of Taylor UUCP has debugging compiled into it. As noted above, under COHERENT Taylor UUCP writes its debugging information into file **/usr/spool/uucp/.Admin/audit.local**. You can place the command **debug** into file **config** to set the level of debugging to use by default.

LEXICON

Please What the Taylor documentation calls a *level* of debugging really records information about a given *activity*. For example, the command **debug chat** tells Taylor UUCP to record information about all actions taken while executing a chat script — not just the problems that occur while a chat script is being executed.

The command **debug** recognizes the following commands:

abnormal
Log abnormal situations.

chat Log chat-script activities.

handshake
Log activities during handshaking with the remote system.

uucp-proto
Log activities that involve the UUCP session protocol.

proto Log activities that involve individual link protocols.

port Log activities that involve the communications port.

config Log activities that occur while reading the configuration files.

spooldir
Log activities in the spool directory.

execute
Log whenever a program is executed.

incoming
Log all incoming data.

outgoing
Log all outgoing data.

all Log all of the above.

You can name more than one activity with the **debug** command. If you have more than one activity, the items in the list of activities must be separated by a comma instead of white space; for example, command

```
debug chat,handshake
```

tells UUCP to log activities that occur during execution of the chat script and handshaking.

A form of the **debug** command lets you invoke activities by number from the above list. note that the order is significant: **abnormal** is activity number zero, and **all** activity 11. For example, command

```
debug 3
```

tells UUCP to log activities zero through three — that is, **abnormal** through **uucp-proto**.

Note, too, that the **debug** command in this file can be overridden by using command-line option **-x** with any UUCP command.

See Also

Administering COHERENT, dial, port, sys, UUCP

connect() — Sockets Function (libsocket)

Connect to a socket
#include <sys/types.h>
#include <sys/socket.h>
int connect(*socket***,** *name***,** *namelen***)**
int *socket***,** *namelen***; struct sockaddr** **name***;*

The function **connect()** establishes a connection for a socket.

socket is a file identifier that describes a socket possessed by the current process. It must have been returned by a call to **socket()**. If it is of type **SOCK_DGRAM**, **connect()** specifies the peer with which the socket is to be connected; this address is that to which datagrams are to be sent, and the only address from which datagrams are to be received. If, however, it is of type **SOCK_STREAM**, **connect()** attempts to connect it with another socket. The

other socket is identified by *name*, which points to the full path name of the file to which the other socket is bound. This connection must have been established by a call to function **bind()**. *namelen* gives the length, in bytes, of the file name to which *name* points.

As a rule, a socket of type **SOCK_STREAM** can successfully connect only once; however, those of type **SOCK_DGRAM** sockets can call **connect()** multiple times to change their association. Datagram sockets can dissolve the association by connecting to an invalid address, such as a null address.

If the connection or binding succeeds, **connect()** returns zero. If an error occurs, it returns -1 and sets **errno** to an appropriate value. The following lists the errors that can occur, by the value to which **connect()** sets **errno**:

EBADF *socket* is somehow invalid.

ENOTSOCK
socket references a file, not a socket.

EADDRNOTAVAIL
The address is not available on this machine.

EAFNOSUPPORT
Addresses in the specified address family cannot be used with *socket*.

EISCONN
socket is already connected to an address or socket.

ETIMEDOUT
connect() timed out without establishing a connection.

ECONNREFUSED
The attempt to connect was forcefully rejected.

ENETUNREACH
The network is not reachable from this host.

EADDRINUSE
The address is already in use.

EFAULT
name gives an illegal address.

EINPROGRESS
socket is non-blocking yet the connection cannot be completed immediately.

EALREADY
The socket is non-blocking and a previous call to **connect()** has not yet been completed.

Example

For an example of this function, see the Lexicon entry for **libsocket**.

See Also

accept(), getsockname(), libsocket, select(), socket()

console — Device Driver

Console device driver

/dev/console is the device driver for the console of a COHERENT system. It is currently assigned major device number 2 and minor device number 0.

/dev/console interprets escape sequences in console output to control output on the console monitor. These escape sequences include the sequences from ANSI 3.4-1977 and ANSI X3.64-1979 that deal with terminal control. Thus, they are similar to those used by the DEC VT-100 and VT-220 terminals.

Escape Sequences

In addition to the ASCII control characters BEL, BS, CR, FF, HT, LF, and VT, **/dev/console** recognizes a number of special sequences, each of which is introduced by the ASCII character ESC. You can type these on the keyboard, or write them in a file and invoke them by **cat**ing the file to the standard output.

The following gives the escape sequences that **/dev/console** recognizes. The text in parentheses gives the ANSI

mnemonic for this escape sequence. Note that in this table, **ESC** represents the ASCII character ESC (i.e., 0x1B). **CSI** stands for Control Sequence Introducer, which here consists of the character ESC followed by the character '[' (0x5B). Note, too, that this table inserts spaces between characters. This is simply for the sake of legibility; at present, no escape sequence can contain a literal space character.

ESC = Enter alternate keypad mode. This escape sequence is non-standard and is slated for removal; you should avoid embedding it in scripts or programs.

ESC > Exit alternate keypad mode. This escape sequence is non-standard and is slated for removal; you should avoid embedding it in scripts or programs.

ESC *n* Print the special graphics character *n*.

ESC 7 Save the current cursor position. This escape sequence is non-standard and is slated for removal; you should avoid embedding it in scripts or programs.

ESC 8 Restore the previously saved cursor position. This escape sequence is non-standard and is slated for removal; you should avoid embedding it in scripts or programs.

ESC D (IND, Index)
Move the cursor down one line without changing the column position. This command moves the scrolling region text up and inserts blank lines if required. Although this escape sequence now moves the cursor down, it may not do so in the future when COHERENT supports writing systems other than left-to-right, top-to-bottom. Furthermore, this control sequence has been marked for removal from future international standards. This escape sequence has been slated for removal; you should avoid embedding it in scripts or programs.

ESC E (NEL, Next Line)
Move the cursor to the first column of the next line. This command move the scrolling region down and inserts blank line if required.

ESC M (RI, Reverse Index)
Move the cursor up one line without changing column position. As with IND, the direction of motion depends on the writing system currently in use.

CSI *n* **@** (ICH, Insert Character)
Insert *n* characters at the current position (default, one).

CSI *n* **A** (CUU, Cursor Up)
Move the cursor up *n* rows (default, one). Stop at top of page.

CSI *n* **B** (CUD, Cursor Down)
Move the cursor down *n* rows (default, one). Stop at bottom edge of scrolling region.

CSI *n* **C** (CUF, Cursor Forward)
Move the cursor *n* columns forward (default, one). Stop at right bottom corner of scrolling region.

CSI *n* **D** (CUB, Cursor Backwards)
Move the cursor *n* columns backwards (default, one).

CSI *n* **E** (CNL, Cursor Next Line)
Move the cursor *n* rows down (default, one). Move scrolling region up and insert a blank line if required.

CSI *n* **F** (CPL, Cursor Preceding Line)
Move the cursor *n* rows up (default, one). Move the scrolling-region text down and insert a blank line if required.

CSI *n* **G** (CHA, Cursor Character Absolute)
Move the cursor to column *n* of the current line.

CSI *n* **;** *m* **H** (CUP, Cursor Position)
Move the cursor to column *m* of row *n*. Position is relative to the scrolling region.

CSI *n* **I** (CHT, Cursor Horizontal Tabulation)
Move the cursor *n* tabulation stops forward (default, one).

CSI *c* **J** (ED, Erase in Display)
Erase display, where *c* is one of the following characters:

- **0** Erase from cursor to end of screen.
- **1** Erase from beginning of screen to cursor.
- **2** Erase the entire screen.

CSI *c* **K** (EL, Erase in Line)
Erase line, where *c* is one of the following characters:

- **0** Erase from cursor to end of line.
- **1** Erase from beginning of line to cursor.

2 Erase entire line.

CSI *n* **L** (IL, Insert Line)
Insert *n* blanks lines (default, one).
CSI *n* **M** (DL, Delete Line)
Delete *n* lines (default, one).
CSI *c* **O** (EA, Erase in Area)
Erase scrolling region, where *c* is one of the following characters:

0 Erase from cursor to end of scrolling region.
1 Erase from beginning of scrolling region to cursor.
2 Erase entire scrolling region. Reposition cursor at top left corner of scrolling region.

CSI *n* **P** (DC, Delete Character)
Delete *n* characters at the current position (default, one).
CSI *n* **S** (SU, Scroll Up)
Scroll the characters in the scrolling region up by *n* lines (default, one). The bottom of the scrolling region is cleared to blanks.
CSI *n* **T** (SD, Scroll Down)
Scroll the characters in the scrolling region down *n* lines (default, one). The top line of the scrolling region is cleared to blanks.
CSI *n* **X** (ECH, Erase Character)
Erase *n* characters at the current position (default, one).
CSI *n* **Z** (CBT, Cursor Backward Tabulation)
Move the cursor backwards by *n* tabulation stops (default, one).
CSI *n* **‘** (HPA, Horizontal Position Absolute)
Move the cursor to column *n* of the current line.
CSI *n* **a** (HPR, Horizontal Position Relative)
Move the cursor forward (i.e., to the right) *n* columns in the current line.
CSI *n* **d** (VPA, Vertical Position Absolute)
Move the cursor to row *n* of the display.
CSI *n* **e** (VPR, Vertical Postition Relative)
Move the cursor down *n* rows.
CSI *n* **;** *m* **f** (HVP, Horizontal and Vertical Position)
Move the cursor to column *m* of row *n*.
CSI *s1* **;** ... *sN* **m** (SGR, Select Graphic Rendition)
Select graphics rendition on the terminal. This command takes one or more colon-separated parameters *s1* through *sN*, each of which is one of the following strings:

0 All attributes off.
1 Bold intensity.
4 Underlining on. On color terminals, underlining rendered as white characters on a red background, in compliance with UNIX practices.
5 Blink on.
7 Reverse video.
10 Select primary font (see notes, below).
11 Select first alternative font (see notes, below).
12 Select second alternative font (see notes, below).
30 Black foreground.
31 Red foreground.
32 Green foreground.
33 Brown foreground.
34 Blue foreground.
35 Magenta foreground.
36 Cyan foreground.
37 White foreground.
40 Black background.
41 Red background.
42 Green background.
43 Brown background.
44 Blue background.

45 Magenta background.
46 Cyan background.
47 White background.

For example, the following command sets the foreground color to cyan and the background to black:

```
echo '\033[36;40m'
```

The following codes are not standard, and are slated for modification. Do not embed these codes in scripts or programs:

In the default font (font 0), **/dev/console** ignores control characters other than BEL, BS, CR, ESC, FF, HT, LF and VT and prints all other ASCII characters.

In font 1, **/dev/console** prints all characters (including control characters), except ESC.

In font 2, **/dev/console** prints every character except ESC with the high bit toggled. This provides access to the IBM graphics character set using ordinary ASCII characters.

CSI *n* **;** *m* **r** Make rows *n* through *m* of the display into the scrolling region. This is not a standard control sequence. It implements functionality included in standard sequences, and will be removed from a future console driver that implements the standard sequence.

CSI *c* **v** Select cursor rendition, where *c* is one of the following characters:

0 Cursor visible.
1 Cursor invisible.

This is not a standard sequence. It implements functionality not provided by any standard sequence. Developers are cautioned that there is no truly portable equivalent (although on many systems positioning the cursor off the screen has the same effect).

CSI ? 4 h (SM, Set Mode)
Enable smooth scrolling. This eliminates "snow" from the screen, but slows down the speed at which the console scrolls. The mode selected by the private-use parameter **?4** is not a standard mode.

Note that the term "smooth" is somewhat misleading; it means that the driver waits for vertical retrace before it updates video memory. The reason for waiting for retrace was that the old CGA tubes were poorly designed — the CRT logic and the main CPU were allowed simultaneous access to the video memory, with the result that direct-memory screen writes often produced static (snow). Having code wait for vertical retrace obviates the problem, but it also slows down the screen.

CSI ? 4 l (RM, Reset Mode)
Disable smooth scrolling. This is the default. The mode selected by the private-use parameter **?4** is not a standard mode.

CSI ? 7 h (SM, Set Mode)
Enable wraparound. Typing past column 80 moves the cursor to the first column of the next line, scrolling if necessary. The mode selected by the private-use parameter **?7** is not a standard mode, but is mandated by iBCS2.

CSI ? 7 l (RM, Reset Mode)
Disable wraparound. The cursor will not move past column 80. This is useful if the screen is being used as a block-mode interface. The mode selected by the private-use parameter **?7** is not a standard mode, but is mandaetd by iBCS2.

CSI ? 8 h (SM, Set Mode)
Erase in the current foreground color.

CSI ? 8 l (RM, Reset Mode)
Erase in the original foreground color, even if the current mode is reverse video.

CSI ? 25 h (SM, Set Mode)
Enable line 25.

CSI ? 25 l (RM, Reset Mode)
Disable line 25.

CSI > 13 h (SM, Set Mode)
Enable the screen saver. This is not standard.

CSI > 13 l (RM, Reset Mode)
Disable the screen saver. This is not standard.

ESC ' (DMI, Disable Manual Input)
Disable manual input. Terminal "beeps" (outputs **<ctrl-G>**) when you press a key on the keyboard. Interrupt and quit signals are still passed to the terminal process. Input may be renabled via **ESC c** (power up reset) or **ESC b** (enable manual input).

ESC b (EMI, Enable Manual Input)
Enable keyboard input that has been disabled by **ESC '**.

ESC c (RIS, Reset to Initial State)
Reset to power-up configuration

ESC t Enter keypad-shifted mode. This is a non-standard sequence that conflicts with explicit provisions of the relevant standards. It will be removed from future versions of the console driver in favor of a sequence that does not conflict.

ESC u Exit keypad-shifted mode. This is a non-standard sequence that conflicts with explicit provisions of the relevant standards. It will be removed from future versions of the console driver in favor of a sequence that does not conflict.

Numeric Keypad

The following describes the sequences sent by the numeric keypad.

The keypad sends the following escape sequences:

Key 0 Send **CSI L**.
Key 1 Send **CSI F**.
Key 2 Send **CSI B**.
Key 3 Send **CSI G**.
Key 4 Send **CSI D**.
Key 5 Send **ESC 7**.
Key 6 Send **CSI C**.
Key 7 Send **CSI H**.
Key 8 Send **CSI A**.
Key 9 Send **CSI I**.
Key . Send ASCII DEL.

When the **<shift>** key is pressed or the **<num-lock>** key is set, the keypad sends the literal characters '0' through '9' and '.'. If the **<num-lock>** key is set, pressing **<shift>** restores the escape sequences shown above.

The escape sequence **ESC =** sets the alternate-keypad mode. In this mode, the keypad sends the following escape sequences when the **<num-lock>** key is not set:

Key 0 Send **ESC ? p**.
Key 1 Send **ESC ? q**.
Key 2 Send **ESC ? r**.
Key 3 Send **ESC ? s**.
Key 4 Send **ESC ? t**.
Key 5 Send **ESC ? u**.
Key 6 Send **ESC ? v**.
Key 7 Send **ESC ? w**.
Key 8 Send **ESC ? x**.
Key 9 Send **ESC ? y**.
Key . Send **Esc ? n**.

The escape sequence **ESC >** resets this mode.

Other Special Keys

The following gives the escape sequences sent by the keyboard's special keys:

<home> Send "cursor home" (**CSI H**).
<up> Send "cursor up" (**CSI A**).
<pg up> Send **CSI I**.
<left> Send "cursor left" (**CSI D**).
<right> Send "cursor right" (**CSI C**).
<end> Send **CSI F**. Note that this escape sequence does not do what users normally expect: to send cursor to bottom left of screen, send the escape sequence **CSI 24 H**.

<down> Send "cursor down" (**CSI B**).
<pg dn> Move cursor to previous page (**CSI G**).
<ins> Send **CSI L**. Note that this escape sequence does not do what users normally expect.
<del> Send ASCII DEL.
F1 Send **CSI M**.
F2 Send **CSI N**.
F3 Send **CSI O**.
F4 Send **CSI P**.
F5 Send **CSI Q**.
F6 Send **CSI R**.
F7 Send **CSI S**.
F8 Send **CSI T**.
F9 Send **CSI U**.
F10 Send **CSI V**.
<shift>F1 Send **CSI Y**.
<shift>F2 Send **CSI Z**.
<shift>F3 Send **CSI a**.
<shift>F4 Send **CSI b**.
<shift>F5 Send **CSI c**.
<shift>F6 Send **CSI d**.
<shift>F7 Send **CSI e**.
<shift>F8 Send **CSI f**.
<shift>F9 Send **CSI g**.
<shift>F10 Send **CSI h**.
<ctrl>F1 Send **CSI k**.
<ctrl>F2 Send **CSI l**.
<ctrl>F3 Send **CSI m**.
<ctrl>F4 Send **CSI n**.
<ctrl>F5 Send **CSI o**.
<ctrl>F6 Send **CSI p**.
<ctrl>F7 Send **CSI q**.
<ctrl>F8 Send **CSI r**.
<ctrl>F9 Send **CSI s**.
<ctrl>F10 Send **CSI t**.
<ctrl><shift>F1
Send **CSI w**.
<ctrl><shift>F2
Send **CSI x**.
<ctrl><shift>F3
Send **CSI y**.
<ctrl><shift>F4
Send **CSI z**.
<ctrl><shift>F5
CSI @.
<ctrl><shift>F6
CSI [.
<ctrl><shift>F7
**CSI **.
<ctrl><shift>F8
CSI].
<ctrl><shift>F9
CSI ^.
<ctrl><shift>F10
CSI _.
<alt>F1 Send **CSI 1 y**.
<alt>F2 Send **CSI 2 y**.
<alt>F3 Send **CSI 3 y**.
<alt>F4 Send **CSI 4 y**.
<alt>F5 Send **CSI 5 y**.
<alt>F6 Send **CSI 6 y**.

<alt>F7	Send **CSI 7 y**.
<alt>F8	Send **CSI 8 y**.
<alt>F9	Send **CSI 9 y**.
<alt>F10	Send **CSI 0 y**.
<esc>	Send ASCII ESC (0x1B).
<tab>	Send ASCII HT.
<ctrl>	When combined with 'A' through '_', send the corresponding ASCII control character; when combined with the (¢) key, send ASCII LF; when combined with the key **<backspace>**, send ASCII DEL ; when combined with **<alt>** and **<del>**, issue system reset.
<shift>	Change alphabetic keys from lower case to upper case. If the **<caps-lock>** is set, shift from upper case to lower case.
<alt>	When combined with **<ctrl>** and **<del>**, issue a system reset.
<backspace>	Send ASCII BS; when combined with **<ctrl>**, send ASCII DEL.
<return>	Send ASCII CR; when combined with **<ctrl>**, send ASCII LF.
*	Send ASCII '*'.
<caps-lock>	Toggle "caps lock" mode.
<num-lock>	Toggle the interpretation of the numeric keypad, as described above.
<scroll-lock>	Send **<ctrl-S>** and toggle the Scroll Lock LED.
-	Send '-'.
+	Send '+'.

Altering Console Configuration

To change the hardware configuration of your console (i.e., to switch from a monochrome to a color console, or modify your keyboard or configuration of virtual consoles), log in as the superuser **root** and type the following commands:

```
cd /etc/conf
console/mkdev
bin/idmkcoh -o /kernel_name
```

where *kernel_name* is what you wish to name the newly built kernel. When you reboot, invoke *kernel_name* in the usual manner and your new configuration will have been implemented.

The following tunable kernel variables affect the behavior of the console driver:

CON_BEEP_SPEC
: This tunable kernel parameter lets you toggle whether the console can beep. If you set it to zero, the console will not beep, no matter what. By default, this is set to one, which enables beeping.

SEP_SHIFT
: This tunable kernel variable permits each virtual-console session to have its own settings for the keyboard's shift keys. When this variable is set to one, you can have **<CAPS LOCK>** turned on in one screen and **<NUM LOCK>** in another, and the driver correctly remembers the proper shift state when you switch sessions. The default for this variable is zero — that is, the keyboard uses the same settings for the shift keys in every virtual-console session.

See Also

Administering COHERENT, ASCII, device drivers, virtual consoles

Notes

Under COHERENT release 4.2, the codes sent by the keys **F1** through **F10**, **<pg up>**, **<pg dn>**, **<ins>**, **<del>**, and **<end>** have changed from those sent under previous releases. This was done so that COHERENT can more closely conform to the standard expected by many third-party packages. If this presents a problem, you can use the COHERENT command **fnkey** to change the codes sent by the function.

If you are using the keyboard driver **vtnkb**, you can remap the keyboard and (within limits) change the codes sent by some keys. For details, see the Lexicon entry **vtnkb**.

Beginning with COHERENT release 4.2, the console uses a 25-line screen, rather than the 24-lines used in previous releases. This is to support the numerous third-party packages that assume a 25-line display. A variant form of the **termcap** and **terminfo** entries for **ansipc** returns the screen to 24 lines, should you need that feature.

Please note that as of this writing (March 1994), the sequences **CSI** *n m*; **do not work, where** *n* **is between 50 and 57. This is being worked repaired.**

const — C Keyword

Qualify an identifier as not modifiable

The type qualifier **const** marks an object as being unmodifiable. An object declared as being **const** cannot be used on the left side of an assignment (an *lvalue*), or have its value modified in any way. Because of these restrictions, an implementation may place objects declared to be **const** into a read-only region of storage.

See Also

C keywords, volatile
ANSI Standard §6.5.3

const.h — Header File

Declare machine-dependent constants
#include <sys/const.h>

The header file **const.h** declares most machine-dependent constants. These are constants that change among the various machines for which the COHERENT system is available; an example is the clock speed of the processor.

See Also

header files, times()

Notes

This header file is obsolete and will be dropped from a future release of COHERENT. Its use is strongly discouraged.

continue — Command

Terminate current iteration of shell construct
continue [*n*]

The command **continue** helps to control the flow of commands given to the shell. When it is used without an argument, **continue** terminates the execution of the current iteration of the innermost **for**, **until**, or **while** shell construct; that is, it acts like a branch to the enclosing **done**, after which loop execution may continue or terminate. If an argument is given, **continue** terminates the current iteration of the *n*th enclosing **for**, **until**, or **while** loop.

The shell executes **continue** directly.

See Also

break, commands, for, ksh, sh, until, while

continue — C Keyword

Force next iteration of a loop

continue forces the next iteration of a **for**, **while**, or **do** loop. For example,

```
while ((foo = getchar()) != EOF) {
      if ((foo < 'a') || (foo > 'z'))
            continue;
      ...         /* do something */
}
```

forces the **while** loop to throw away everything except lower-case alphabetic characters.

See Also

C keywords, for, while
ANSI Standard, §6.6.6.2

controls — System Administration

Data base for the lp print spooler
/usr/spool/mlp/controls

The file **/usr/spool/mlp/controls** is the data base for the print spooler **lp**. The superuser **root** can modify this file, either with a text editor or (to a more limited extent) with the command **lpadmin**.

The format of **controls** is simple. Every blank line is ignored. All text after the pound sign '#' is also ignored; you

can use this feature to embed comments in the file. The rest of the file consists of commands, each of which has the format *command*=*arguments*.

The following describes the commands that you can embed in **controls**:

default=*printer*
This command sets the default printer, that is, the printer on which jobs are printed when the user does not specify a printer on the **lp** command line.

docopies=*status*
This command controls how **lp** prints multiple copies. If it is set to **on**, then multiple copies are generated by invoking the printer's control script for each time; if it is set to **off**, then multiple copies are printed by telling the control script to do it. The difference in the two methods is that the former gives you more accurate information about the status of the job. If you wish to print many copies and you want to monitor the job's progress, then set **docopies** to **on**.

feed=*status*
The command **localfeed** tells **lp** whether to insert a formfeed character between printing jobs sent to printers other than local printers. Setting it to **on** tells **lp** to output a formfeed character; setting it to **off** (or deleting it) tells it not to do so.

localfeed=*status*
The command **localfeed** tells **lp** whether to insert a formfeed character between printing jobs sent to a local printer. (A *local* printer is one plugged into the auxiliary port of a terminal.) Setting it to **on** tells **lp** to output a formfeed character; setting it to **off** (or deleting it) tells it not to do so.

logroll=*hours*
This command sets the time, in hours, at which the log file **log** is renamed **log.o** and a fresh log file is begun. This is done so the log file does not grow without bounds. The default value is 168 hours (one week).

longlife=*hours*
Set, in hours, the "life-expectancy" of a file with a lifetime of **L**. The default is three days (72 hours).

printer=*name*,*device*,*script*
This command defines a printer. **lp** accesses a printer by its name; it cannot access a printer unless you name it in a **printer** command. *name* names the printer. You can name a printer anything you like, so long as it is one word. *device* names the device into which it is plugged. *script* names the file in directory **/usr/spool/mlp/backend** that tells how to massage the text being passed to the printer. You can write or modify each script in that directory, and name each script whatever you like. Note that one physical printer can have multiple names, each using a different script; and one script can be shared by multiple physical printers.

The command

```
printer = linenlq, /dev/lpt2, pannlq
```

names a printer **linelq** that is plugged into port **/dev/lpt2**, and whose input is filtered through the contents of script **/usr/spool/mlp/backend/pannlq**.

The command

```
printer = linepr, /dev/lpt2, linepr
```

names a printer **linepr** that is plugged into **/dev/lpt2**, and whose input is filtered through the contents of script **/usr/spool/mlp/backend/linepr**.

Note that these examples both name the same physical device. They differ in the scripts they use to massage their input; this will be described in detail below.

Finally, a **printer** can direct its output to any device, serial or parallel, even **/dev/null**. For example:

```
printer=disk,/dev/null,disk
```

As will be shown below, the script **disk** writes its output into a temporary file, so you can examine it without wasting a piece of paper. The format of a printer-control script is described below.

You do not have to include a printer-control script in a **printer** command; if you do not include one, the printer daemon **lpsched** uses the command **cat** by default.

shortlife=*hours*
Set, in days, the "life-expectancy" of a file with a lifetime of **S**. The default is 48 hours (two days).

templife=*hours*
Set, in minutes, the "life-expectancy" of a file with a lifetime of **T**. The default is two hours.

Printer Control Scripts

A printer-control script massages the text being handed to a given printer. The printer daemon **lpsched** redirects the output of the script (and therefore, of every command within the script) to the device named on the appropriate **printer** command named in the file **/usr/spool/mlp/controls**.

For example, consider the command

```
printer = linenlq, /dev/lpt2, pannlq
```

This command names a printer **linenlq**, declares that it is plugged into port **/dev/lpq2**, and requests that **lpsched** massage input to the printer through script **/usr/spool/mlp/backend/pannlq**. When **lpsched** processes a request that is directed to printer **linelq**, it pipes the text of the job into script **pannlq**, and redirects the output of **pannlq** to device **/dev/lpt2**.

It is important to remember that a printer-control script is not restricted to a few commands that the spooler understands. Each is a true shell script that can use any or all COHERENT commands to process text. The limits of what a script can do are set only by your imagination.

Consider the following examples. In the discussion, above, of the command **printer**, two scripts were mentioned: **pannlq** and **linepr**. Both send their output to the same physical printer, but they process the input text in different ways. The following gives the contents of **linepr**:

```
# filter the input through pr
pr

# throw a page at the end
echo "\f\c"
```

This script filters its input through the COHERENT command **pr**, which paginates the text and puts a header on it. It then echoes a formfeed character, to force the printer to throw a blank page at the end of the job. As in other shell scripts, a pound sign '#' introduces a comment and blank lines are ignored.

The following gives the contents of script **pannlq**:

```
# turn on near-letter-quality printing
echo "\021\033n"

pr

# turn off near-letter-quality printing
echo "\021\033P"
```

This script resembles the first, except that it includes commands to echo the magic strings that turn on and turn off near-letter-quality printing on this printer. This is one small example of the flexibility you can employ in devising a script

As with other shell scripts, you can modify the behavior of a printer-control script by setting environmental variables. For example, consider the following variation on the script **linepr**:

```
if [ $HEADER ]; then
      pr -h "$HEADER"
else
      pr
fi

# throw a page at the end
echo "\f\c"
```

If you have exported the environmental variable **HEADER**, then this script prints it at the top of each page; otherwise, it prints the default header. You can use the same technique to do other work, such as force the printing of a banner page.

The **lp** spooler reserves for its own use the environmental variables **MLP_COPIES**, **MLP_FORMLEN**, **MLP_LIFE**, **MLP_PRIORITY**, **MLP_SPOOL**. Your scripts can also use these variables. For more information on what each does, see its entry in the Lexicon.

When **lpsched** uses a printer-control script, it passes it three arguments: respectively, the sequence number of the print job (which identifies the job uniquely); the name of the user; and the number of copies being printed. You can use this information to control the printing of output; for example, consider the following:

```
for i in ‘from 1 to $3‘
do
      pr -h "User $2 - Copy $i of $3"
done
echo "\f\c"
```

Note, too, that just as each physical printer can be accessed in different ways via different scripts, so too the same script can be used by multiple physical printers. If you had multiple Panasonic printers plugged into your system, you could use the above script with each of them to massage their input appropriately.

One last example. As noted above, the output of a printer-control script can be directed to any device, not just a port. (It can also be redirected to non-existent ports, so be careful when you enter your **print** commands.) You can use this feature to redirect formatted text into files or other interesting places. Consider the following **printer** command:

```
printer=disk,/dev/null,disk
```

This creates a “printer” named **disk**. The text filtered through file **disk** is redirected to **/dev/null**. The contents of script **disk** show what this device is up to:

```
tee /tmp/D$$
```

This script uses the COHERENT command **tee** to redirect its input both to the standard output (which in the case of printer **disk** is thrown away) and into a file in directory **tmp**. You can use this command to save input for further examination later.

This discussion just scratches the surface of what you can do with the **lp** print spooler and its control scripts. For more information, see the Lexicon entries for **printer** and **lp**.

See Also

Administering COHERENT, lp, lpadmin, MLP_COPIES, MLP_FORMLEN, MLP_LIFE, MLP_PRIORITY, MLP_SPOOL, printer

conv — Command

Numeric base converter
conv [*number*]

conv converts *number* to hexadecimal, decimal, octal, binary, and ASCII characters, and prints the results on the standard output. If no *number* is given, **conv** reads one number per line from the standard input until you type the end-of-file character **<ctrl-D>**.

number may be in hexadecimal, decimal, octal, binary, or character format, as shown below. Each example represents the decimal number 97.

Base	*Representation*
hexadecimal	**0x61**
hexadecimal	**#61**
decimal	**97**
octal	**0141**
binary	**$1100001**
character	**‘a’**

conv represents an ASCII control character in its output by preceding the character by a carat ‘^’. For example, it prints **<ctrl-X>** as ‘^X’. **conv** prints “bad digit” if anything is wrong with the input.

See Also

bc, commands, conv, dd, od, units

Notes

conv represents the input *number* internally as a **long** integer. If *number* does not fit in a **long**, **conv** silently truncates it.

core — System Administration

Format of a core-dump file
#include <sys/core.h>

When a process terminates abnormally because a it encounters an unrecoverable error or receives an asynchronous signal from another process, COHERENT tries to write a copy of its image in memory into a file called **core**. You can examine this file with the debugger **db** and other tools to try to determine what went wrong.

The structure **ch_info** appears at the head of a **core** file. The header file **core.h** defines it as follows:

```
struct  ch_info {
      unsigned short ch_magic;
      unsigned int ch_info_len;
};
```

Field **ch_magic** is always set to the constant **CORE_MAGIC**. This "magic" value signifies to COHERENT that this is a core file. Field **ch_info_len** gives a count of information bytes in the core file, including the **ch_info** structure itself.

If the value of **ch_info_len** exceeds the size of the **ch_info** structure, this indicates that data follow the **ch_info** structure. These data follow the **ch_info** structure, and are in the form of a **core_proc** structure. Header file **<sys/core.h>** defines this structure as follows:

```
struct core_proc {
      gregset_t cp_registers;
      int cp_signal_number;
      struct _fpstate cp_floating_point;
      dregset_t cp_debug_registers;
};
```

This substitutes for a dump of the **u** area, whose information is reserved for the kernel alone.

This is followed by an image of each process segment. The data for each segment consists of the following: a header, which is a structure of type **core_seg**; **cs_pathlen** bytes of data that give the path name of the file from the segment data originated; and **cs_dumped** bytes of core-image data.

core.h defines strucutre **core_seg** as follows:

```
struct core_seg {
       size_t cs_pathlen;                      /* length of path name */
       off_t cs_dumped;                        /* dumped size in bytes */
       caddr_t cs_base;                        /* virtual base address */
       off_t cs_size;                          /* full size in bytes */
       unsigned long cs_reserved[8];
};
```

The order of the segments is the text segment first (if it is present — usually it is omitted), followed by the data segment, and then the stack segment. The contents of the text segment can usually be identified from the program being debugged. The patchable kernel variable **DUMP_TEXT** allows the COHERENT kernel to dump text segments as well as data and stack segments.

Patchable kernel variable **DUMP_LIM** sets the maximum size of a segment within a **core** file. The system uses this limit to keep keep core files from getting out of hand.

See Also

Administering COHERENT, core.h, signal(), wait()

Diagnostics

COHERENT will not write **core** if that file already exists as a non-ordinary file or if there is more than one link to it. The 0200 bit in the status returned to the parent process by **wait()** indicates a successful dump.

For a list of signals that automatically trigger a core dump, see the Lexicon entry for **signal()**.

core.h — Header File

Declare structure of a core file
#include <sys/core.h>

The header file **core.h** includes the structures and constants from which the system builds a **core** file. For more information on **core** files, see the Lexicon entry for **core**.

See Also

core, header files

cos() — Mathematics Function (libm)

Calculate cosine
#include <math.h>
double cos(*radian***) double** *radian***;**

cos() calculates the cosine of its argument *radian*, which must be in radian measure.

Example

For an example of this function, see the entry for **sin()**.

See Also

acos(), cosh(), libm
ANSI Standard, §7.5.2.5
POSIX Standard, §8.1

cosh() — Mathematics Function (libm)

Calculate hyperbolic cosine
#include <math.h>
double cosh(*radian***) double** *radian***;**

cosh() calculates the hyperbolic cosine of *radian*, which is in radian measure.

Example

The following example uses **cosh()** to compute the height and time to impact of a falling object. Assume that an object is acted on both by gravity and by air resistance propotional to v^2, where v is its velocity. When p is the proptionality constant for the resistance of air, the object's height after t seconds is given by the formula

```
y = y0 -1/p*ln(cosh(t*sqrt(p*g)))
```

and its time to reach the ground is given by the formula:

```
t = 1/sqrt(p*g)*log(exp(p*y0)+sqrt(exp(2*p*y0)-1))
```

Assuming that

```
g = 32 ft/s^2
```

the example computes an object's height after t seconds and the total time in seconds that it will take to reach the ground. It was written by Sanjay Lal (sanjayl@tor.comm.mot.com):

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

main ()
{
      float height, init_height, resistance, time_to_hit, g;
      int i;
      char buffer[50];

      g = 32.0;

      printf("Enter initial height, in feet: ");
      fflush(stdout);
      init_height = atof(gets(buffer));
```

```
      resistance = 0.0;
      while (resistance > 0.005 || resistance < 0.001) {
            printf("Enter air resistance (0.001 to 0.005): ");
            fflush(stdout);
            resistance = atof(gets(buffer));
      }

      time_to_hit = 1.0/sqrt(resistance*g) *
            log(exp(resistance*init_height) +
            sqrt(exp(2*resistance*init_height)-1));

      printf("Initial height: %1.0f\n", init_height);
      printf("Air resistance: %1.3f\n", resistance);
      printf("Time for object to hit the ground: %1.3f seconds\n",
            time_to_hit);

      /* countdown to impact */
      for (i = 2; ; i++) {
            height = init_height -
                  (1.0/resistance*log(cosh(sqrt(resistance*g)*(double)i)));

            if (height < 0) {
                  printf("BOOM!\n");
                  exit(EXIT_SUCCESS);
            } else
                  printf("Height after %i seconds: %1.3f feet\n", i, height );
      }
}
```

See Also

libm, cos()
ANSI Standard, §7.5.3.1
POSIX Standard, §8.1

Diagnostics

When overflow occurs, **cosh()** returns a huge value that has the same sign as the actual result.

cp — Command

Copy a file
cp [-d] *oldname newname*
cp [-d] *file1 ... fileN directory*

cp copies files. In its first form, **cp** copies the contents of *oldname* to *newname*, which it creates if necessary. If *newname* is a directory, **cp** copies *oldname* to a file of the same name in directory *newfile*.

In its second form, **cp** copies each *file*, from *file1* through *fileN*, into *directory*.

With the **-d** option, **cp** preserves the date (modification time) of the source file or files on the target file or files. By default, target files get the current time.

A file cannot be copied to itself.

See Also

commands, cpdir, ksh, mv, sh, wildcards

Notes

If you use **cp** to copy a file into another existing file, the newly copied file takes on the permissions of the file into which the text was copied. For example, consider the files **foo** and **bar**, whose permissions are as follows:

```
-rw-r--r--  1 fred       user            40 Tue Apr 14 18:19 bar
-rw-r-----  1 fred       user          1816 Tue Apr 14 20:53 foo
```

If you use **cp** to copy **foo** into **bar**, then typing **ls -l** shows the following:

```
-rw-r--r--  1 fred       user          1816 Tue Apr 14 21:37 bar
-rw-r-----  1 fred       user          1816 Tue Apr 14 20:53 foo
```

bar now has exactly the same contents as **foo** but retains its old set of permissions.

cpdir — Command

Copy directory hierarchy
cpdir [*option ...* **]** *dir1 dir2*

cpdir copies source directory hierarchy *dir1* to target hierarchy *dir2,* which is created if necessary. Either hierarchy may straddle device boundaries.

cpdir preserves as much as possible of the source structure. Files under *dir1* go to identically named files under *dir2.* Links between source files are preserved as links between corresponding target files. Preserved source file attributes include mode, subject to the user's file creation mask. If the user is not the superuser, **cpdir** cannot preserve the owner, group, and sticky bits in the mode, and the invoking user owns all new files; under the superuser it preserves these as well. In addition, the superuser may "copy" special nodes and pipe nodes; **cpdir** copies only the facility, not the contents. It also preserves real major and minor device numbers of special nodes.

If the target file corresponding to a source file exists and is not a directory, **cpdir** unlinks it before copying. This differs from the action of **cp**.

cpdir recognizes the following options:

-a Give a verbose account on one line of the files copied.

-d Preserve the last-modified date instead of using the present date.

-e Print error message and continue execution after an error. The default action is to exit on any error.

-r [*n***]** Descend no more than *n* levels in the source hierarchy. Contents of *dir1* are at level 1. If missing, *n* defaults to 1.

-s *name*
Suppress the copy of file *name*, which should be the pathname of the file relative to *dir1.*

-t Test only, make no changes. With this option, **cpdir** prints a report of all errors (**-e** is implied), all unlinked target files, and other useful information, including a summary of all external links into the target hierarchy that would have been broken had the unlinking actions been executed.

-u Update regular files. Copy the source only if it was created or altered more recently than the target file, or if the target does not exist.

-v Print a verbose account of its activities. **cp** prints a file-by-file account of its actions, in addition to the information listed under **-t.**

See Also

commands, cp, link(), umask(), unlink()

cpio — Command

Archiving/backup utility

cpio is a standard utility that writes archives of files to disk or tape. Under COHERENT, **cpio** is a link to the command **gnucpio**. For details, see the Lexicon entry for that command.

See Also

commands, gnucpio

cpp — Command

C preprocessor
/lib/cpp [*option…***] [***file…***]**

The command **cpp** calls the C preprocessor to perform C preprocessing. It performs the operations described in section 3.8 of the ANSI Standard; these include file inclusion, conditional code selection, constant definition, and macro definition. See the entry on **C preprocessor** for a full description of C's preprocessing language.

Normally, **cpp** is used to preprocess C programs, but it can be used as a simple macro processor for other types of files as well. For example, the X utility **imake** uses **cpp** to help build makefiles.

cpp reads each input *file*, processes directives, and writes its product on **stdout**. If the option **-E** is not used, **cpp**

also writes into its output statements of the form **#line***n filename*, so the parser can connect its error messages and debugger output with the original line numbers in your source files.

Options

cpp recognizes the following options:

-C Do not suppress comments. Normally, **cpp** strips all comments from C code before it invokes the parsing phase, **cc0**.

-D*VARIABLE***[=***value***]**
Define *VARIABLE* for the preprocessor at compilation time. If *value* is not defined, *VARIABLE* is set to one. For example, the command

```
cc -DLIMIT=20 foo.c
```

tells the preprocessor to define the variable **LIMIT** to be 20. The C preprocessor acts as though the directive **#define LIMIT 20** were included in all source code.

-E Strip all line-number information from the source code. This option is used to preprocess assembly-language files or other sources, and should not be used with the other compiler phases.

-I*directory*
C allows two types of **#include** directives in a C program, i.e., **#include "file.h"** and **#include <file.h>**. The **-I** option tells **cpp** to search a specific directory for the files you have named in your **#include** directives, in addition to the directories that it searches by default. You can have more than one **-I** option on your **cpp** command line.

-o *file* Write output into *file*. If this option is missing, **cpp** writes its output onto **stdout**, which may be redirected.

-P Strip all file and line-number information from the C code. This is identical to the **-E** option, defined above.

-Q Suppress all messages.

-U*VARIABLE*
Undefine *VARIABLE*, as if an **#undef** directive were included in the source program. This is used to undefine the variables that **cpp** defines by default.

-V Print verbose messages.

-VCPLUS
Suppress C++-style online comments.

cpp reads the environmental variables **CPPHEAD** and **CPPTAIL** and appends their contents to, respectively, the beginning and the end of the command **cpp**.

See Also

C preprocessor, cc, commands

Diagnostics

The following gives the error messages returned by **cpp**. The messages are in alphabetical order. Each is marked as to whether it is a *fatal*, *error*, or *warning* condition. A fatal message usually indicates a condition that caused the compiler to terminate execution. Fatal errors from the later phases of compilation often cannot be fixed, and may indicate problems in the compiler or assembler. An error message points to a condition in the source code that the compiler cannot resolve. This almost always occurs when the program does something illegal, e.g., has unbalanced braces.

string argument mismatch *(error)*
The argument *string* does not match the type declared in the function's prototype. Either the function prototype or the argument should be changed.

#assert failure *(error)*
The condition being tested in a **#assert** statement has failed.

at beginning of macro *(error)*
Macro replacement lists may contain tokens that are separated by **##**, but **##** cannot appear at the beginning or the end of the list. The tokens on either side of the **##** are pasted together into one token.

at end of macro *(error)*
Macro replacement lists may contain tokens that are separated by **##**, but **##** cannot appear at the beginning or the end of the list. The tokens on either side of the **##** are pasted together into one token.

string: cannot create *(fatal)*
The preprocessor **cpp** cannot create the output file *string* that it was asked to create. This often is due to a problem with the output device; check and make sure that it is not full and that it is working correctly.

string: cannot open *(fatal)*
The compiler cannot open the file *string* of source code that it was asked to read. **cpp** may not have been told the correct directory in which this file is to be found; check that the file is located correctly, and that the **-I** options, if any, are correct.

cannot open include file *string (fatal)*
The program asked for file *string*, which was not found in the same directory as the source file, nor in the default **include** directory specified by the environmental variable **INCDIR**, nor in any of the directories named in **-I** options given to the **cc** command.

conditional stack overflow *(fatal)*
A series of **#if** expressions is nested so deeply that it overflowed the allotted stack space. You should simplify this code.

#define argument mismatch *(warning)*
The definition of an argument in a **#define** instruction does not match its subsequent use. One or the other should be changed.

#elif used without #if or #ifdef *(error)*
An **#elif** instruction must be preceded by an **#if**, **#ifdef**, or **#ifndef** control line.

#elif used after #else *(error)*
An **#elif** instruction cannot be preceded by an **#else** instruction.

#else used without #if or #ifdef *(error)*
An **#else** control line must be preceded by an **#if**, **#ifdef**, or **#ifndef** control line.

#endif used without #if or #ifdef *(error)*
An **#endif** instruction must be preceded by an **#if**, **#ifdef**, or **#ifndef** instruction.

EOF in comment *(fatal)*
Your source file appears to end in mid-comment. The file of source code may have been truncated, or you failed to close a comment; make sure that each open-comment symbol '/*' is balanced with a close-comment symbol '*/'.

EOF in macro *string* invocation *(error)*
Your source file appears to end in a macro call. The source file may be been truncated.

EOF in midline *(warning)*
Check to see that your source file has not been truncated accidentally.

EOF in string *(error)*
Your file appears to end in the middle of a quoted string literal. Check to see that your source file has not been truncated accidentally.

#error: *string (fatal)*
An **#error** control line has been expanded, printing the remaining tokens on the line and terminating the program.

error in #define syntax *(error)*
The syntax of a **#define** statement is incorrect. See the Lexicon entry for **#define** for more information.

error in #include syntax *(error)*
An **#include** directive must be followed by a string enclosed by either quotation marks (" ") or angle brackets (<>). Anything else is illegal.

identifier *string* has too many arguments *(error)*
Too many actual parameters have been provided.

LEXICON

illegal control line *(error)*
A '#' is followed by a word that the compiler does not recognize.

illegal cpp character (*n* decimal) *(error)*
The character noted cannot be processed by **cpp**. It may be a control character or a non-ASCII character.

illegal use of defined *(error)*
The construction **defined(***token***)** or **defined** *token* is legal only in **#if**, **#elif**, or **#assert** expressions.

string in #if *(error)*
A syntax error occurred in a **#if** declaration. *string* describes the error in detail.

include stack overflow *(fatal)*
A set of **#include** statements is nested so deeply that the allotted stack space cannot hold them. Examines the files for a loop. You should try to fold some of the header files into one, instead of having them call each other.

macro body too long *(fatal)*
The size of the macro in question exceeds the limit designed into the preprocessor. Try to shorten or split the macro.

macro expansion buffer overflow in *string (fatal)*
The COHERENT C compiler uses a static buffer space to expand preprocessor macros. In some extreme cases, a macro will exhaust this space, thus causing the C compiler to exit with this message. Try to shorten the macro, or break it up. See the Lexicon entry for **cpp** for suggestions on how to use an alternative C preprocessor to expand huge macros.

macro *string* redefined *(error)*
The program redefined the macro *string*.

macro *string* requires arguments *(error)*
The macro calls for arguments that the program has not supplied.

macros nested *number* deep, loop likely *(error)*
Macros call each other *number* times; you may have inadvertently created an infinite loop. Try to simplify the program.

missing #endif *(error)*
An **#if**, **#ifdef**, or **#ifndef** instruction was not closed with an **#endif** instruction.

missing output file *(fatal)*
The preprocessor **cpp** found a **-o** option that was not followed by a file name for the output file.

multiple #else's *(error)*
An **#if**, **#ifdef**, or **#ifndef** expression can be followed by no more than one **#else** expression.

nested comment *(warning)*
The comment introducer sequence '/*' has been detected within a comment. Comments do not nest.

new line in *string* literal *(error)*
A newline character appears in the middle of a string. If you wish to embed a newline within a string, use the character constant '\n'. If you wish to continue the string on a new line, insert a backslash '\' before the new line.

newline in macro argument *(warning)*
A macro argument contains a newline character. This may create trouble when the program is run.

out of space *(fatal)*
The compiler ran out of space while attempting to compile the program. To remove this error, examine your source and break up any functions that are extraordinarily large.

parameter must follow # *(error)*
Macro replacement lists may contain **#** followed by a macro parameter name. The macro argument is converted to a string literal.

preprocessor assertion failure *(warning)*
A **#assert** directive that was tested by the preprocessor **cpp** was found to be false.

string redefined *(error)*
: **cpp** macros should not be redefined. You should check to see that you are not **#include**ing two different versions of a file somehow, or attempting to use the same macro name for two different purposes.

too many arguments in a macro *(fatal)*
: The program uses more than the allowed ten arguments with a macro.

too many directories in include list *(fatal)*
: The program uses more than the allowed ten **#include** directories.

string: unknown option *(fatal)*
: The preprocessor **cpp** does not recognize the option *string*. Try re-typing the **cc** command line.

Notes

The COHERENT C compiler uses a static buffer space to expand preprocessor macros. Some programs that make especially intensive use of the C processor's macro facility may die during compilation with the message

```
macro expansion buffer overflow
```

This means that the program has exhaused the compiler's ability to process macros. You may wish to use an alternative preprocessor, such as the one that comes with **gcc**, as described below.

The COHERENT C compiler combines the preprocessor **cpp** with the parser **cc0**. The file **/lib/cpp** is simply a link to the C compiler **/lib/cc0**. Thus, there is no way to specify an alternative version of the preprocessor through the **cc** command. You can get around, this however, by linking the alternative preprocessor to a file named **cc0** in a directory other than **/lib**, then calling the alternative version via **cc**. For example, to have **gcc** preprocess program **hugemacro.c**, do the following. First, type the following commands to link the **gcc** preprocess to a file named **cc0**:

```
su root
cd /usr/local/lib/gcc-lib/i386-coh/2.3.2
ln cpp cc0
```

Then, to preprocess and compile **hugemacro.c**, type the following:

```
cc -t0 -B/usr/local/lib/gcc-lib/i386-coh/2.3.2 -E hugemacro.c > tmp.c
cc tmp.c
rm tmp.c
```

You may wish to embed the above into your **makefile**, or write it into a shell script.

CPPHEAD — Environmental Variable

Append options to beginning of cpp command line
export CPPHEAD=*options*

The COHERENT C preprocessor **cpp** reads the environmental variables **CPPHEAD** and **CPPTAIL** before it begins its work. You can set these variables to hold the default options that you want the preprocessor always to use.

cpp appends the options in **CPPHEAD** to the beginning of its command line.

See Also

cpp, CPPTAIL, environmental variables

CPPTAIL — Environmental Variable

Append options to end of cpp command line
export CPPTAIL=*options*

The COHERENT C preprocessor **cpp** reads the environmental variables **CPPHEAD** and **CPPTAIL** before it begins its work. You can set these variables to hold the default options that you want the preprocessor always to use.

cpp appends the options in **CPPTAIL** to the end of its command line.

See Also

cpp, CPPHEAD, environmental variables

creat() — System Call (libc)

Create/truncate a file
#include <fcntl.h>
int creat(*file***,** *mode***)**
char **file***; **int** *mode*;*

creat() creates a new *file* or truncates an existing *file*. It returns a file descriptor that identifies *file* for subsequent system calls. If *file* already exists, its contents are erased. In this case, **creat()** ignores the specified *mode*; the mode of the *file* remains unchanged. If *file* did not exist previously, **creat()** uses the *mode* argument to determine the mode of the new *file*. For a full definition of file modes, see **chmod()** or the header file **stat.h**. **creat()** masks the *mode* argument with the current **umask**, so it is common practice to create files with the maximal mode desirable.

Example

For an example of how to use this routine, see the entry for **open()**.

See Also

chmod(), fcntl.h, fopen(), libc, open(), stat.h, stdio.h
ANSI Standard, §4.9.3
POSIX Standard, §5.3.2

Diagnostics

If the call is successful, **creat()** returns a file descriptor. It returns -1 if it could not create the file, typically because of insufficient system resources or protection violations.

cron — System Administration

Execute commands periodically
/etc/cron&

cron is a daemon that executes commands at preset times.

Once each minute **cron** searches for commands to execute. **cron** first looks for file **/usr/lib/crontab**. If it exists, then **cron** reads it for commands to execute. If **/usr/lib/crontab** does not exist, however, **cron** searches **/usr/spool/cron/crontabs** for command files. Each user can have her own command file in that directory. See the Lexicon entry for **crontab** for information how to write and load a command file.

For each entry in each command file, **cron** compares the current time with the scheduled execution time and executes the command if the times match. When it finishes the search, **cron** sleeps until the next minute. Because it never exits, **cron** should be executed only once (customarily by **/etc/rc**).

cron is designed for commands that must be executed regularly. Temporal commands that need to be executed only once should be handled with the command **at**.

Permissions

cron performs some interesting manipulations with permissions. This is necessary to allow **cron** to run a wide variety of programs untended without creating loopholes in the system's security. Occasionally, this can create difficulties for users who do not grasp what **cron** does or why. The following describes how **cron** manipulates permissions on the programs you ask it to run.

To begin, when **cron** executes a user's **crontab** file, it changes the effective user ID to the ID of that user whose **crontab** file is being executed, **cd**'s to the user's **HOME** directory. When, however, **cron** runs an entry from a **/usr/lib/crontab**, it uses the user ID and group ID of user **daemon**. This prevents security holes involving entries in a **crontable** file.

For example, the following **crontab** entry contains redirection:

```
* * * * *   echo hello world >/dev/console 2>&1
```

If **cron** finds this entry in **/usr/lib/crontab**, it tries to execute the command as user **daemon**. The command will not execute it if user **daemon** lacks permission to write to **/dev/console**. Note that using **/usr/lib/crontab** is *not* recommended.

If however, it finds the entry in user **henry**'s **crontab** file, it tries to execute the command under the effective user ID of **henry**. The command will fail if **henry** lacks permission to write to **/dev/console**, and will succeed if he does.

When the shell executes a command in the background, it reads its standard input from **/dev/null** (unless redirected) and writes its standard output to the controlling tty. If **cron** is invoked with **/etc/cron&** from **/etc/rc**, there is no controlling tty, so the standard output goes to **/dev/null**. Thus,

```
* * * * *   echo hello world
```

typically writes **hello world** to **/dev/null**.

When a user logs in, **/bin/login** grabs the tty and runs **chown** and **chmod** on it. It is owned by the user with default permissions 700. If the user who has logged in on the console types the command

```
chmod /dev/console a+w
```

to allow all users to write to it, then the **crontab** entry

```
* * * * *   echo hello world >/dev/console 2>/tmp/cron.err
```

will indeed echo to the console every minute.

cron should be executed only once, at boot time. It uses **/usr/lib/cron/FIFO** as a lock file to prevent the execution more than one **cron** daemon.

If mail options are enabled, which is the default, **cron** sends mail to the owner of a **crontab** about all commands in that file that failed.

To allow **cron** to remove lock file **/usr/lib/cron**, do *not* send signal **KILL** to **cron**. Instead, use signal **TERM**. **cron** ignores signals **INT**, **HUP**, and **PIPE**. **cron** uses the signal **ALRM** internally.

Files and Directories

/usr/lib/cron/FIFO
Lock file (named pipe). Created by **cron**; removed by **cron/rc**.

/usr/lib/cron/cron.allow
List of allowed users. Permissions: **600 root root**.

/usr/lib/cron/cron.deny
List of denied users. Permissions: **600 root root**.

/usr/lib/crontab
Global **crontab** file, used by previous COHERENT **cron** mechanism.

/usr/spool/cron
Spool directory parent. Permissions: **700 root root**.

/usr/spool/cron/crontabs
Main **cron** directory. It holds each user's command file. Permissions: **700 root root**.

See Also

Administering COHERENT, commands, crontab

Notes

cron does not presently write into the log file. The size of the *hostname* + *domain* must not exceed 1,000 characters.

cron looks for **/usr/lib/crontab** to remain compatible with the COHERENT 286 version of **cron**. If, however, you continue to keep all **cron** commands in file **/usr/lib/crontab**, it will not be possible to run **setuid cron** tasks for logins that have a password. It is strongly recommended that you do *not* use **/usr/lib/crontab**, and instead create individual **crontab** files.

crontab — Command

Copy a command file into the crontab directory
/usr/bin/crontab [-l] [-r] [-f *filename***] [-m[ed]] [-u***user***]**

The command **crontab** copies a command file into directory **/usr/spool/cron/crontabs**. This directory holds the command files for all users. This mechanism permits each user to have her own file of commands to be executed periodically. If the file name is '-', then **crontab** reads the standard input.

crontab recognizes the following options.

-f *filename*
Replace your crontab file with *filename*.

-l List your crontab file.

-m[ed] Enable/disable the sending of mail to a user about any command in her crontab file that fails.

-r Remove your crontab file.

-u *user* Specify *user*. Only the superuser **root** can specify any user other than herself.

Allowing and Denying Access

The files **/usr/lib/cron/cron.allow** and **/usr/lib/cron/cron.deny** let the system administrator govern which users can use the **crontab** command:

- If **cron.allow** exists, then **crontab** checks its contents; if a given user is identified therein, then she can use **crontab**. Obviously, if **cron.allow** exists but is empty, then nobody can use **crontab**.
- If **cron.allow** does not exist, then **crontab** checks the contents of **cron.deny**. If a given user is identified therein, then she cannot use **crontab**; otherwise, she can. If **cron.allow** does not exist and **cron.deny** exists but is empty, then everyone can use **crontab**.
- If neither file exists, then everyone can use **crontab**.

Format of a crontab File

A **crontab** command file consists of lines separated by newlines. Each line consists of six fields separated by white space (tabs or blanks). The first five fields describe the scheduled execution time of the command. Respectively, they represent the minute (0-59), hour (0-23), day of the month (1-31), month of the year (1-12), and day of the week (0-6, 0 indicates Sunday). Each field can contain a single integer in the appropriate range, a pair of integers separated by a hyphen '-' (meaning all integers between the two, inclusive), an asterisk '*' (meaning all legal values), or a comma-separated list of the above forms. The remainder of the line gives the command to be executed at the given time.

For example, the **crontab** entry

```
29 * * 7 0 msg henry Succotash!
```

means that every hour on the half-hour during each Sunday in July, **cron** will invoke the command **msg**, and the user named **henry** will have the message

```
daemon: Succotash!
```

written on his terminal's screen (if he is logged in).

crond recognizes three special characters and escape sequences in a **crontab** file. If a command contains the percent character '%', **crond** executes only the portion up to the first '%' as a command, then passes the remainder to the command as its standard input. **crond** translates any percent characters '%' in the remainder to newlines. To prevent the special interpretation of a '%', precede it with a backslash, '\%'. Finally, **crond** removes the sequence **<newline>** from the text before it passes the text to the shell **sh**; this can be used to make an entry in the **crontab** more legible.

You must pay special attention to permissions when you write a **crontab** command file. For information on how the **crontab** daemon **crond** manipulates permissions, see the entry for **crond** in the Lexicon.

Directories and Files

/usr/spool/cron/crontabs
Main **cron** directory. It holds each user's command file. Permissions: **700 root root**.

/usr/lib/cron/FIFO
Lock file (named pipe). Created by **cron**; removed by **crond/rc**.

/usr/lib/cron/cron.allow
List of allowed users. Permissions: **600 root root**.

/usr/lib/cron/cron.deny
List of denied users. Permissions: **600 root root**.

/usr/lib/crontab
Global **crontab** file, used by previous COHERENT **cron** mechanism. **/usr/spool/cron** Spool directory parent. Permissions: **700 root root**.

/usr/spool/cron/crontabs
Spool directory. Permissions: **700 root root**.

See Also

commands cron

Notes

COHERENT **crontab** is superset of the command of the same name included with UNIX System V release 3 (SVR3). The main differences are as follows:

- COHERENT **crontab** prints the usage when no options have been chosen, whereas SVR3 **crontab** reads stdin and can just remove the user's crontab file.
- SVR3 **crontab** does not include option **-f** *file_name*.
- SVR3 **crontab** does not include option **-u** *user*. Under SVR3 **crontab**, you must **su** to another user (e.g., **uucp**) before you can maintain her **crontab** file.

crypt — Command

Encrypt/decrypt text
crypt [*password***]**

The command **crypt** encrypts data. It emulates a rotor-encryption machine, such as the Enigma or Hagelin C-48 cipher machines. Unlike these machines, **crypt** uses only one rotor, with a 256-character alphabet and a keying sequence of period 2^32.

crypt reads text from standard input and writes the encrypted text to standard output. *password* is used to construct the model of the machine and to start the keying sequence. If no *password* is given, **crypt** prompts for a password on the terminal and disables echo while it is being typed in. The *password* may be up to ten characters long, but must not be empty; all characters past the first ten are ignored. All characters are legal, although it may not be possible to input certain characters from the terminal.

crypt uses the same *password* for both encryption and decryption. For example, the commands

```
crypt COHERENT <file1 >file2
crypt COHERENT <file2 >file3
```

leave *file3* identical to *file1*.

Encrypted files produced by **ed** with its **-x** option may be read by **crypt**, and vice versa, as **ed** uses **crypt** to perform its encryption.

Security of a cryptosystem depends on several factors:

1. Brute-force attempts to break the system should be infeasible. Passwords should be at least five characters long; although the construction of the machine model from the password takes a substantial fraction of a second, it is still plausible that encrypted files could be read by a brute-force search of a portion of the password space (say, all passwords less than four characters long).
2. Cryptanalysis of the basic encryption scheme should be very hard. Analysis of rotor machines is understood, but it is difficult and in most cases probably not worth the trouble.
3. Passwords must be kept secret. **crypt** erases *password* as soon as it can, to avoid the possibility that it could appear in the output of **ps**.
4. Privileged access to the system must be guarded. Under COHERENT, the security of **crypt** can be no better than the security governing access to superuser status, because the superuser can do practically anything. This is probably **crypt**'s most vulnerable point.

Files

/dev/tty — Typed passwords

See Also

commands, passwd, security, shadow

crypt() — General Function (libc)

Encryption using rotor algorithm
char *__crypt__**(***key, extra***); char** **key***, ***extra***;**

crypt() implements a version of rotor encryption. It produces encrypted passwords that are verified by comparing the encrypted clear text against an original encryption.

key is an ASCII string that contains the user's password. *extra* is a "salt" string of two additional characters that are stored in the password file with the encrypted password. Each character must come from an alphabet of 64 symbols, which consists of the upper-case and lower-case letters, digits, the period '.', and the slash '/'.

crypt() returns a string built from the 64-character alphabet described above; the first two characters returned are the *extra* argument, and the rest contain the encrypted password.

See Also

libc

ct — Device Driver

Controlling terminal driver

Most processes that the COHERENT kernel executes are associated with a *controlling terminal*. (The only exceptions are daemon processes that are started by the process **init**.) This terminal directs I/O to the physical device through which the user who invoked the process is accessing COHERENT. Usually, this is a serial port or the console, but it could also be a socket (in the case of a **telnet** or **ftp** session), or some other device.

The driver **ct** lets a program access the controlling terminal automatically. It is accessed through the device **/dev/tty**. Thus, when a program invokes the system calls **open()**, **close()**, **ioctl()**, **read()**, or **write()** on **/dev/tty**, driver **ct** directs those calls automatically to the appropriate driver for the controlling terminal. This spares applications from having to know the details of the controlling device — all it has to do is manipulate **/dev/tty** and let **ct** take care of the details.

Files

/dev/tty

See Also

device drivers, init

Diagnostics

When a call finds no valid controlling terminal for a process, it returns a value of -1 and sets **errno** to **ENXIO.**

ctags — Command

Generate tags and refs files for vi editor
ctags [**-r**] *files...*

ctags generates the files **tags** and **refs** from a group of C-source files. **tags** is used by the **elvis** editor's **:tag** command, **<ctrl-]>** command, and **-t** option. **refs** is used by the command **ref**.

Each C-source file is scanned for **#define** statements and global function definitions. The name of the macro or function becomes the name of a tag. For each tag, a line is added to **tags**, which contains the following:

- the name of the tag
- a tab character
- the name of the file containing the tag
- a tab character
- a way to find the particular line within the file

refs is used by the command **ref**, which can be invoked via **elvis**'s **K** command. When **ctags** finds a global function definition, it copies the function header into **refs**. The first line is flush against the right margin, but the

argument definitions are indented. The command **ref** can search **refs** much faster than it could search all C-source files. The file-names list will typically include the names of all C-source files in the current directory, in the following format:

```
ctags -r *.[ch]
```

The **-r** to **ctags** tells it to generate both **tags** and **refs**. Without **-r**, it generates only **tags**.

See Also

commands, elvis, ref

Notes

This version of **ctags** does not parse ANSI source code very well. It has trouble recognizing the ANSI function definitions.

Please note that this program is offered as a service to COHERENT users, but is not supported by Mark Williams Company. *Caveat utilitor.*

ctermid() — General Function (libc)

Name the terminal device that controls the current process
#include <stdio.h>
char ***ctermid (***path_name***)**
char **path_name***;*

The general function **ctermid()** returns the full path name of the terminal device that controls the current process. It does for the controlling terminal what the function **ttyname()** does for a general file descriptor.

path_name points to a block of memory into which **ctermid()** can write the name of the contolling terminal. It must point to at least **L_ctermid** bytes of available memory. If *path_name* is NULL, **ctermid()** writes the name into a statically allocated buffer that may be overwritten by subsequent calls to **ctermid()**.

If all goes well, **ctermid()** returns the address where it wrote the name of the controlling terminal. If an error occurs — for example, it could not discover the name of the controlling terminal — it returns an empty string.

See Also

libc
POSIX Standard 1003.1, §4.7.1

Notes

In almost every instance, **ctermid()** returns the string "/dev/tty". Under COHERENT, the name of the controlling terminal for the current process is **/dev/tty**. Because some operating systems do not follow this common practice, POSIX Standard provides **ctermid()** as a portable means of getting the controlling terminal's name.

ctime() — Time Function (libc)

Convert system time to an ASCII string
#include <time.h>
#include <sys/types.h>
char ***ctime(***timep***)**
time_t ***timep***;**

ctime() converts the system's internal time into a string that can be read by humans. It takes a pointer to the internal time type **time_t**, which is defined in the header file **<sys/types.h>**, and returns a fixed-length string of the form:

```
Thu Mar  7 11:12:14 1989\n
```

ctime() is implemented as a call to **localtime()** followed by a call to **asctime()**.

Example

For another example of this function, see the entry for **asctime()**.

```
#include <time.h>
#include <sys/types.h>

main()
{
      time_t t;

      time(&t);
      printf("%s\n", ctime(&t));
}
```

See Also

libc, time [overview], time.h
ANSI Standard, §7.12.3.2
POSIX Standard, §8.1

Notes

ctime() returns a pointer to a statically allocated data area that is overwritten by successive calls.

ctype.h — Header File

Header file for data tests
#include <ctype.h>

ctype.h declares and defines the following routines, which can check and transform character types:

_tolower()	Convert an upper-case character to lower case
_toupper()	Convert a lower-case character to upper case
isalnum()	Test if alphanumeric character
isalpha()	Test if alphabetic character
isascii()	Test if ASCII character
iscntrl()	Test if a control character
isdigit()	Test if a numeric digit
isgraph()	Test if a graphics character
islower()	Test if lower-case character
isprint()	Test if printable character
ispunct()	Test if punctuation mark
isspace()	Test if a tab, space, or return
isupper()	Test if upper-case character
isxdigit()	Test if hexadecimal numeral
toascii()	Convert a character to ASCII
tolower()	Convert an upper-case character to lower case
toupper()	Convert a lower-case character to upper case

Example

The following example demonstrates **isalnum()**, **isalpha()**, **isascii()**, **iscntrl()**, **isdigit()**, **islower()**, **isprint()**, **ispunct()**, and **isspace()**. It prints information about the type of characters it contains.

```
#include <ctype.h>
#include <stdio.h>

main()
{
      FILE *fp;
      char fname[20];
      int ch;
      int alnum = 0;
      int alpha = 0;
      int allow = 0;
      int control = 0;
      int printable = 0;
      int punctuation = 0;
      int space = 0;
```

```
        printf("Enter name of text file to examine: ");
        fflush(stdout);
        gets(fname);

        if ((fp = fopen(fname, "r")) != NULL) {
              while ((ch = fgetc(fp)) != EOF) {

                    if (isascii(ch)) {
                          if (isalnum(ch))
                                alnum++;
                          if (isalpha(ch))
                                alpha++;
                          if (islower(ch))
                                allow++;
                          if (iscntrl(ch))
                                control++;
                          if (isprint(ch))
                                printable++;
                          if (ispunct(ch))
                                punctuation++;
                          if (isspace(ch))
                                      space++;

                    } else {
                          printf("%s is not ASCII.\n",
                                      fname);
                          exit(1);
                    }
              }

              printf("%s has the following:\n", fname);
              printf("%d alphanumeric characters\n", alnum);
              printf("%d alphabetic characters\n", alpha);
              printf("%d alphabetic lower-case characters\n",
                    allow);
              printf("%d control characters\n", control);
              printf("%d printable characters\n", printable);
              printf("%d punctuation marks\n", punctuation);
              printf("%d white space characters\n", space);
              exit(0);

        } else  {
              printf("Cannot open \"%s\".\n", fname);
              exit(2);
        }
}
```

See Also

header files, libc
ANSI Standard, §7.3

Notes

The argument for a **ctype** function or macro should be an **int** that is representable as an **unsigned char** or EOF — i.e., [-1, 0, ..., 255], as described in the ANSI standard §4.3.

The functions **_tolower()**, **_toupper()**, **isascii()**, and **toascii()** are not part of the ANSI standard. Programs that use them may not be portable to all implementations of C.

cu — Command

UNIX-compatible communications utility
cu [options] [*system***] [***phone***] [dir]**

The command **cu** implements a version of the communications utility used under UNIX System V. (Its name is an acronym for "call UNIX".) With it, you can interactively telephone other systems, upload files, download files, and perform other communications tasks. Unlike the program **ckermit**, which is also included with COHERENT, **cu** uses the information stored in UUCP data-base files **dial**, **port**, and **sys** to automate the dialing of a remote system.

To tell **cu** to dial a given system, just use that system's name on the **cu** command line. **cu** then reads from files

dial, **port**, and **sys** the information on how to dial the system you have named; then uses that information to open the port, set up the modem, and dial the system, up to the point where you see a login prompt on the remote system. For example, to dial system **mwcbbs**, use the command:

```
cu mwcbbs
```

Instead of dialing a remote system, you may wish to talk directly to a modem — for example, to reset its registers; or you may wish to log into a local system that is directly connected to your system via a serial port. To talk directly to a device, use the option **-p** followed by the name of the port into which the device is plugged, plus the command **dir**. This command tells **cu** that you wish to talk to the port directly. (Ports are named in the file **port**; for details, see its entry in the Lexicon.) For example, to talk directly a modem that is on a port named **MWCBBS**, use the command:

```
cu -p MWCBBS dir
```

To have **cu** dial a specific telephone number over a specific port, again use the option **-p** option to name the port, followed by the telephone number to call. For example, the command

```
cu -p MWCBBS 17085590412
```

connects to the modem on port **MWCBBS** and dial the telephone number 1-708-559-0412.

cu assumes that a string that begins with an alphabetic character names a system. To call a system whose name begins with numeral, use the command-line option **-z**, described below.

cu Commands

You can give commands to **cu** while you converse with the remote system. Each command begins with an escape character, which by default is the tilde '~'. **cu** recognizes the escape character only when it appears at the beginning of a line. After you type the escape character, **cu** replies with the name of your system, to show that it is ready to receive your command. If you do not see **cu**'s reply within a second or two, something has gone wrong.

To send to the remote system an escape character at the beginning of a line, enter it twice; for example, typing

```
~~
```

sends a single '~' to the remote system. All commands are either a single character or a word that begins with '%'.

cu recognizes the following commands:

~. Terminate the conversation.

~! *command*
Run *command* in a shell on your local system. If no *command* is given, start up a shell.

~$ *command*
Run *command* on your local system, and redirect to the remote system what *command* writes to the standard output.

~| *command*
Run *command* on your local system, and pipe into *command* what the remote system sends to your system.

~+ *command*
Combine the commands **~$** and **~|**. You can use this command to invoke alternative file-transfer utilities, e.g., **rz** and **sz**.

~#
~%break
Send a break signal.

~c *directory*
~%cd *directory*
cd to *directory* on your local system.

~> *file* Send *file* to the remote system. This command just dumps the file over the communication line, and performs no error checking. It assumes that the remote system is expecting it. You should first open a file on the remote system such as through the command

```
cat > filename
```

before you invoke this feature of **cu**.

~< Receive a file from the remote system. **cu** prompts you to name the file into which it will write what it receives from the remote system, then prompts you for the command to execute on the remote system to begin the file transfer (often, just **cat** *filename*). **cu** reads data from the remote system and writes them into into the file you named on your system until it detects the variable **eofread**.

~p *herefile farfile*
~%put *herefile farfile*
Copy (or *put*) file *herefile* on your system into file *farfile* on the remote system.

~t *farfile herefile*
~%take *farfile herefile*
Take file *farfile* from the remote system, and write it into file *herefile* on your system. This runs the appropriate commands on the remote system.

~s *variable* **[***value***]**
Set the **cu** *variable* to *value*. If no value is not given, set *variable* to **true**. **cu**'s variables are described below.

~! *variable*
Set the **cu** *variable* to **false**. **cu**'s variables are described below.

~%nostop
Turn off **XON/XOFF** flow control.

~%stop
Turn on **XON/XOFF** flow control.

~v List all **cu** variables and their values. **cu**'s variables are described below.

~? Help: list all **cu** commands.

cu Variables

The following variables are build into **cu** to control its default behaviors:

binary This variable indicates whether to pass binary information untouched when it transfers a file. If this variable is false, **cu** converts newline characters to carriage returns. If set to true, then **cu** passes binary data through untouched. The default is **false**.

binary-prefix
This variable gives the string that prefaces a binary character in a file transfer. This variable applies only if the variable **binary** variable is true. The default is **<ctrl-Z>**.

delay If this variable is true, **cu** delays for one second after it recognizes the escape character. The default is true.

echo-check
If **true**, **cu** checks file transfers by examining what the remote system echoes. This is not a robust method of checking the integrity of a transferred file, but it is the best that **cu** offers. The default is **false**.

echonl The character that **cu** looks for after it sends each line in a file. The default is the carriage return.

eofread This sets the string that **cu** looks for after it receives a file retrieved with the command **~<**. The default is **$**, which is intended to be a typical shell prompt.

eofwrite The string that **cu** writes after it sends a file with the command **~>**. The default is **<ctrl-D>**.

eol This variable gives the characters that **cu** recognizes as completing a line of input. **cu** recognizes the escape character only when it occurs immediately *after* one of the **eol** characters. **cu** recognizes the following **eol** characters by default: **<ctrl-C>**, **<ctrl-D>**, **<ctrl-O>**, **<ctrl-Q>**, **<ctrl-R>**, **<ctrl-S>**, and **<ctrl-U>**.

escape The escape character. By default, this is the tilde '~'.

kill This tells **cu** the character to use to delete a line if the echo-check fails. The default is **<ctrl-U>**.

resend The number of times to resend a line if the echo-check continues to fail. The default is ten.

timeout This variable sets the time, in seconds, that **cu** waits for a character either when it does echo-checking or when it looks for the **echonl** character. The default is **30**.

verbose Print accumulated information during a file transfer. The default is **true**.

To list the values of the variables, use the command **~v**. To modify a variable, use the commands **~s** or **~!**. For example, to turn off the one-second pause after sending an escape character, use the command:

```
~! delay
```

To change the escape character from '~' to '\', use the command:

```
~s escape \
```

Options

cu recognizes the following command-line options:

-a *port* The same as the option **-p**, described below.

-c *number* Dial *number*. You must use this option if the telephone number begins with a letter.

-d Enter debugging mode. This is equivalent to **-x all**.

-e Use even parity.

-*N* Equivalent to the command **-s** *N*, where *N* is an integer.

-h Half-duplex mode: echo locally all characters sent to the remote system.

-I *file* Use *file* instead of the configuration file.

-l *device* The device on which to dial out. Use this option to dial out on ports that are not list in the file **port**. You must have write permission on *device*.

-n Prompt for the telephone number to use.

-o Use odd parity. If you use both **-e** and **-o** on the command line, no parity is used. If neither is specified, **cu** uses the default parity of the line.

-p *port* The port to use. If you do not use this option, **cu** uses the default port for the system being contacted, as set in file **/usr/lib/uucp/sys**.

-s *speed* Set the baud rate to *speed*.

-t Map every carriage return character to the pair carriage/linefeed. Use this option when transferring files to an MS-DOS system.

-z *system* Call *system*. You must use this option if the name of the remote system begins with a numeral.

-x *activity* Log a given *activity*. These logs can help you debug problems with **cu**. **cu** recognizes the following activities:

abnormal
chat
handshake
port
config
incoming
outgoing

One **-x** option can name multiple activities, with the activities separated by commas. A **cu** command line can contain multiple **-x** options.

You can also use this option with a number, which turns on that many activities from the foregoing list, in the order in which the appear in this list. For example, the option **-x 2** is equivalent to the option **-x abnormal,chat** The option **-x all** logs on all activities.

See Also

ckermit, commands, dial, port, sys, UUCP

Notes

Unlike **ckermit**, the file-transfer facility in **cu** is primitive and performs no error checking. If you wish primarily to transfer files, you should consider using **ckermit** instead of **cu**. As noted above, the command **~+** plugs the standard input and standard output of two commands into each other; with this feature, you can use the other file-transfer utilities (e.g., **rz** and **sz**) to transfer files under **cu**.

cu requires that the device **/dev/console** appear last in file **/etc/ttys**. If this is not so, **cu** refuses to disable the enabled port or dial out. For details on this file, see the Lexicon entry for **ttys**.

cu was ported to COHERENT from the Taylor UUCP package, written by Ian Taylor (ian@airs.com).

curses.h — Header File

Define functions and macros in curses library
#include <curses.h>

curses.h defines the macros and declares the functions that comprise the **curses** library.

See Also

header files, libcurses, termcap, terminfo

cut — Command

Select portions of each line of its input
cut -c*list* **[***file ...***]**
cut -f*list* **[-s] [-d** *char***] [***file ...***]**

cut "cuts" one or pieces out of each line in its input, and writes the piece or pieces to the standard output. *list* specifies the pieces to cut out of each line. **cut** reads its input from *file*; if no *file* is named on its command line, **cut** reads the standard input.

A "piece" of an input line can be defined either as one or more characters from fixed positions in the line; or as one or more fields. The option **-c** selects characters from fixed positions; you would use this option if you were cutting up a file each of whose lines was of a fixed length. The option **-f** selects fields. A field does not have to have a fixed length, but its end must be marked by some special character; by default, a white-space character marks the end of a field. Option **-d** lets you specify the "magic character" that marks the end of a field. Option **-s** tells **cut** to throw away every line that does not contain the field-delimiter character. By default, **cut** will pass through unmodified every line that does not contain the field delimiter.

Options **-c** and **-f** are each followed by a *list*, which describes the pieces that you want from each input line. A piece is defined as follows:

N A piece consists of a single column or field. For example, the command

```
cut -f2 /etc/ttytype
```

selects field 2 from file **/etc/ttytype**.

N-N The range of columns or fields. For example, command

```
cut -c4-12 /etc/ttytype
```

selects columns 4 through 12, inclusive, from file **/etc/ttytype**.

-N Select every column or field from the beginning of the line through *N*. For example, command

```
cut -d\| -f-3
```

reads the first three fields from the standard input.

N- Select every column or field from *N* through the end of the line. For example, the command

```
cut -c15-
```

selects every character from character 15 through the end of the line.

If *list* defines more than one piece, the definitions of the pieces must be separated by commas. For example, the command

```
cut -c3-5,7-9
```

cuts columns three through five and seven through nine from the standard input, and writes them onto the standard output.

cut returns zero on success, one if an error occurred.

Examples

The following cuts column 4 through the end of the line from file **/etc/ttys**, and writes the cut piece onto the standard output. In effect, it throws away the first three columns of every line in that file:

```
cut -c4- /etc/ttys
```

You would use this command to display every serial-port device name that that file contains.

The next command selects fields one and six from file **/etc/passwd**. (Field one in this file gives a user's login identifier; and field six gives her home directory.) Note that fields in this file are delimited by a colon ':'.

```
cut -d: -f1,6 /etc/passwd
```

The final example cuts the first field from the input. It also explicitly sets the field delimiter to the space character. You would use this command to clip any trailing white space from data read from the standard input:

```
cut -f1 -d' '
```

See Also

awk, commands, paste, sed

Notes

cvmail — Command

Convert mail from COHERENT 3.X format to SV format
cvmail [-m *filename***] [***filename***]**

The command **cvmail** converts to System V format existing COHERENT 3.X mailboxes and files used to store messages saved by COHERENT's 3.X mail utility.

To convert a default mailbox (i.e., a mailbox in directory **/usr/spool/mail**), invoke **cvmail** with its **-m** option, followed by the name of the user whose mailbox is to be converted. For example, to convert the mailbox belonging to user **bob**, type:

```
cvmail -m bob
```

If you have saved mail messages into a file, invoke **cvmail** with the name of the file to convert. For example, if you have stored mail messages in file **msg.save**, you can convert this file by typing:

```
cvmail msg.save
```

See Also

commands, mail

Notes

If you invoke **cvmail** without any arguments, it prompts you for the name of a file to convert. The file is not assumed to be a mailbox in directory **/usr/spool/mail**.

CWD — Environmental Variable

Current working directory

The Korn shell uses the environmental variable **CWD** to hold the current working directory.

See Also

environmental variables, ksh

d_passwd — System Administration

Give passwords for devices
/etc/d_passwd

The COHERENT system lets you force classes of users who log in through particular devices to enter an extra password. This helps you protect your system against people who may be try to break into your system via modem.

When a user attempts to log in, the command **login** check file **/etc/dialups** (should it exist) to see if this device is protected by an extra password. If this file names the device, **login** looks in file **/etc/d_passwd** to see if that user's shell is associated with an extra password. If that is so, then **login** prompts the user for that password, in addition to his usual password as set in file **/etc/passwd** or **/etc/shadow**.

Each entry in **/etc/d_passwd** has the following format:

shell:*password*:*comment*

If field *shell* is empty, then **login** applies this password to all users who are using shells not named elsewhere within **d_passwd**.

The following gives an example of **d_passwd**:

```
/usr/lib/uucp/uucico::UUCP logins don't have extra password
/bin/sh:encrypted password:normal user with interactive shell
/usr/bin/ksh:encrypted password:normal user with interactive shell
```

To recreate the function of the account **remacc** (which **login** no longer recognizes as special), set **/etc/dialups** to name your dial-up ports, and set **d_passwd** to the following:

```
:encrypted_password:people/accounts dialing in
```

The following gives the contents of **d_passwd** from a typical COHERENT system:

```
:.03qn7EtBd.gi:Default dialup password
/usr/lib/uucp/uucico:.03qn7EtBd.gi:Dialup password for UUCP
/bin/sh:.03qn7EtBd.gi:Normal dialup extra password
/usr/bin/ksh:.03qn7EtBd.gi:Normal dialup extra password
```

The gibberish between the first and second ':' characters are the encrypted passwords. Note that this user has given the same password to each shell upon dialing up. This probably is a mistake.

See Also

Administering COHERENT, dialups, login

daemon — Definition

A *daemon* is a program that runs continually on your computer. It waits quietly for some condition to occur; then it awakens and performs some action (such as redirecting the file to a printer).

For example, the daemon **/etc/cron** wakes up every minute and checks every **cron** file. If a file contains a command to be executed at this time, then **cron** executes it.

As a general rule, anything that does not interact directly with users can be classified as a daemon. Daemons do not generally generate output to a user's terminal.

Any time you have a resource, like a printer or data base, to which access should be controlled, you can use a daemon.

For a list of daemons available under the COHERENT system, see the Lexicon entry for **Administering COHERENT**.

See Also

Using COHERENT

Notes

The function **bedaemon()**, which is included in **libmisc**, makes a program a daemon. See the article on **libmisc** for details.

A daemon may be killed accidentally, or through an error condition. When that occurs, a user may summon the daemon from the misty deep, but it will not come. The superuser **root** can reinvoke a daemon like any other program.

data formats — Definition

Mark Williams Company has written C compilers for a number of different computers. Each has a unique architecture and defines data formats in its own way.

The following table gives the sizes, in **char**s, of the data types as they are defined by various microprocessors.

Type	*i80386*	*i8086 SMALL*	*i8086 LARGE*	*Z8001*	*Z8002*	*68000*	*PDP11*	*VAX*
char	1	1	1	1	1	1	1	1
double	8	8	8	8	8	8	8	8
float	4	4	4	4	4	4	4	4
int	4	2	2	2	2	2	2	4
long	4	4	4	4	4	4	4	4
pointer	4	2	4	4	2	4	2	4
short	2	2	2	2	2	2	2	2

COHERENT places some alignment restrictions on data, which conform to all restrictions set by the microprocessor. Byte ordering is set by the microprocessor; see the Lexicon entry on **byte ordering** for more information.

Please note that Intel processor documentation and the Intel Binary Compatibility Standard (iBCS2) use the term *word* differently. The following table defines how they differ:

Bytes	*1*	*2*	*4*	*8*
Bits	*8*	*16*	*32*	*64*
Intel	**byte**	**word**	**dword**	**qword**
iBCS2	**byte**	**halfword**	**word**	**doubleword**

See Also

byte ordering, C language, data types, double, float, float.h, Programming COHERENT

Notes

COHERENT 286 supports Intel SMALL model only. COHERENT 386 supports the i80386 data format.

data types — Definition

COHERENT's implementation of C recognizes the following data types:

char
double
float
int
long
long float
long int
short
short int
signed char
signed int
signed long
signed long int
signed short
signed short int

unsigned int
unsigned long
unsigned long int
unsigned char
unsigned short
unsigned short int

The following types are synonymous:

char	**signed char**		
short	**short int**	**signed short**	**signed short int**
unsigned short	**unsigned short int**		
int	**signed int**		
long	**signed long**	**long int**	**signed lont int**
unsigned long	**unsigned long int**		
long float	**double**	**long double**	

The ANSI Standard states that **int**, **short**, and **long** are signed by default. It also sates that the implementation determines whether a **char** is signed or unsigned by default; but it does state that a printable character must be positive. COHERENT uses signed **char**s by default; therefore, if you wish to use a character value greater than 0x7F, you must explicitly declare the character to have type **unsigned char**. If you use this type in an arithmetic expression, COHERENT's C compiler automatically casts it to **unsigned int**.

Finally, COHERENT's header files define these commonly used data types:

```
<acct.h>    typedef unsigned short comp_t;
<fcntl.h>   typedef struct flock flock_t;
<signal.h>  typedef long sig_atomic_t;
<stddef.h>  typedef int ptrdiff_t;
<stdio.h>   typedef long fpos_t;
<stdlib.h>
            typedef struct { int quot; int rem; } div_t;
            typedef struct { long quot; long rem; } ldiv_t;
<sys/acct.h>
            typedef unsigned short comp_t;
<sys/clist.h>
            typedef unsigned int cmap_t;
<sys/confinfo.h>
            typedef ddi_init_t init_t;
            typedef ddi_start_t start_t;
            typedef ddi_exit_t exit_t;
            typedef ddi_halt_t halt_t;
<sys/fd.h>  typedef unsigned fd_t;
<sys/ksynch.h>
            typedef struct lock_info lkinfo_t;
            typedef struct sleep_lock sleep_t;
            typedef struct readwrite_lock rwlock_t;
<sys/mmu.h>
            typedef long  cseg_t;
<sys/mzioctl.h>
            typedef long mzattr_t;
<sys/poll.h>
            typedef struct event event_t;
<sys/resource.h>
            typedef unsigned long rlim_t;
<sys/scsiwork.h>
            typedef struct scsi_work scsi_work_t;
            typedef struct scsi_cmd scsi_cmd_t;
<sys/seg.h>
            typedef long  cseg_t;
<sys/signal.h>
            typedef n_sigset_t sigset_t;
            typedef o_sigset_t sigset_t;
```

<sys/stream.h>
typedef struct free_rtn frtn_t;
<sys/types.h>
typedef char *vaddr_t;
typedef unsigned short minor_t;
typedef unsigned short major_t;
<sys/uio.h>
typedef struct uio uio_t;

See Also

C language, char, data formats, double, float, int, long, pointer, Programming COHERENT, short, unsigned

date — Command

Print/set the date and time
date [-s] [-u] [[*yymmdd***]***hhmm***[.***ss***]]**

date sets or prints prints the date and time of day.

If invoked without an argument, **date** prints the current date and time. It looks for the environmental variable **TIMEZONE**, which specifies local time zone and daylight saving time information. For details on the format of this variable, see the Lexicon entries for **TIMEZONE** and **ctime()**.

If invoked with a numeric argument (that is, one that consists of just digits, with no prefix), **date** interprets that argument as giving the current date and time, and uses it to set the current system time. The string must have the format *yymmddhhmm***[***ss***]**; the fields must be defined as follows:

yy	Year (00-99)
mm	Month (01-12)
dd	Day (01-31)
hh	Hour (00-23)
mm	Minute (00-59)
ss	Seconds (00-59)

For example, typing

```
date 940612141233
```

sets the date to June 12, 1994, and the time to 2:12:33 P.M. At least *hh* and *mm* must be specified — the rest are optional. **date** will complain and refuse to change the time should you attempt to set an impossible date or time, e.g., the date to February 30 or the time to 25 o'clock.

Note that the COHERENT command **ATclock** returns the date and time as recorded by your computer's internal clock. To reset the time as COHERENT understands it to the time as your computer understands it, use the command:

```
date `/etc/ATclock`
```

If you use option **-s** on **date**'s command line, **date** does *not* convert to daylight savings time when it sets the time.

If you use option **-u** on **date**'s command line, **date** sets and prints the date and time in Greenwich Mean Time (GMT) rather than in your local time.

See Also

ATclock, commands, ctime(), printf(), time, TIMEZONE

Notes

Only the superuser **root** can change the system's date or time.

The COHERENT version of the **date** command differs from the UNIX version in that the last two fields of its output are reversed. For example, the UNIX output of **date** reads

```
Sun Jan 13 12:02:09 CST 1991
```

where the COHERENT output reads:

```
Sun Jan 13 12:02:09 1991 CST
```

This may be important when importing UNIX shell commands into COHERENT.

db — Command

Assembler-level symbolic debugger
db [-a *symfile***] [-cdefort] [[***mapfile***]** *program***]**

db is an interactive, symbolic debugger. It allows you to run object files and executable programs under trace control (see the Lexicon entry for **ptrace**), run programs with embedded breakpoints, and dump and patch files in a variety of forms. You can use it to debug assembly-language programs that have been assembled by **as**, the Mark Williams assembler, and programs that have been compiled with the Mark Williams C compiler **cc**.

What is db?

db is a symbolic debugger, which means that it works with the symbol tables that the compiler builds into the object files it generates. Because **db** works on the level of assembly language, you need a working knowledge of i80386 assembly language and microprocessor architecture.

Invoking db

To invoke **db**, type its name, plus the options you want (if any) and the name of the files with which you will be working. *mapfile* is an object file that supplies a symbol table. *program* is the executable program to be debugged. If both names are given, the options default to **-c**. If only one name is given, it is the *program*; in this case the options default to **-o**. If both names are omitted, *mapfile* defaults to **l.out** or **a.out**, and *program* defaults to **core**. If possible, **db** accesses *program* with write permission.

db recognizes the following command-line options:

-a *symfile*
Read *symfile* for the list of symbols within the executable, instead of the executable's symbol table. This lets you copy an executable's symbol table in *symfile*, then strip that executable.

-c *program* is a core file produced by a user core dump. **db** checks the name of the command that invoked the process that produced the core, against the name of the *mapfile,* if given. Pure segments are read from the *mapfile*.

-d *program* is a system dump. If the command line names no files, *mapfile* defaults to **/COHERENT** and *program* defaults to **/dev/dump**.

-e The next argument is an object file; **db** executes it as a child process and passes it the rest of the command line. This permits the shell to expand wildcard characters that you place in the **db** command line, without spoiling the syntax of the **db** command.

-f Map *program* as a straight array of bytes (file).

-k Map program as a kernel process; *mapfile* defaults to **/coherent**, and *program* defaults to **/dev/kmem**.

-o *program* is an object file. If *mapfile* is given, it is another object file that provides the symbol table.

-p *prompt*
Change the command prompt from **db:** to *prompt*.

-r Only read the file, even if you have write permission for it. Use this to give a file additional protection.

-s Do not load symbol table.

-t Perform input and output for **db** via **/dev/tty**. This permits you to debug a process whose standard input or output has been redirected.

Commands and Addresses

db executes commands that you give it from the standard input. **db** displays the prompt

```
db:
```

when it is ready to receive a command. To change its prompt, use the **-p** option, described above. A command usually consists of an *address*, which tells **db** where in the program to execute the command; and then the command name and its options, if any.

An address is represented by an *expression*, which can be built out of one or more of the following elements:

- The '.', which represents the current address. When you enter an address, **db** sets the current address to that location. To advance the current address, type the **<Enter>** key.
- The name of a register. **db** recognizes the names of all registers on the 80386 microprocessor and the 80387 numeric co-processor. You can preceed a register name with a '%', but this is not required. If your program contains function **eax()**, the identifier **eax** identifies the function and **%eax** the register. If your program does not define **eax**, then either **eax** or **%eax** means the register. For example, the commands

```
sin:b
:e
:s
%st0?N
```

  sets a breakpoint at routine **sin()**, executes to it, single steps into it, and then prints the contents of the NDP stacktop **%st0**, which one step into **sin()** contains the argument.

  Typing the name of a register displays its contents. **db** displays register contents and stack traceback in hexadecimal values, regardless of the current default radix.
- The symbols **d**, **i**, and **u**, which represent location 0 in, respectively, the data space, the instruction space, and the u-area.
- The names of global symbols and symbolic addresses can be used in place of the addresses where they occur. This is useful when setting a breakpoint at the beginning of a subroutine.
- An integer constant, which can be used in the same manner as a global symbol. The default is hexadecimal; a leading **0** indicates octal and **0x** indicates hexadecimal.
- You can use the following binary operators:

  + Addition
  - Subtraction
  ***** Multiplication
  / Integer division

  All arithmetic is done in **long**s.
- You can use the following unary operators:

  ~ Complementation
  - Negation
  ***** Indirection

  All operators are supported with their normal level of precedence. You can use parentheses '()' for binding.

Every symbol refers to a segment: the data segment, the instruction segment, or the u-area. This segment, in turn, dictates the format in which **db** displays by default what it finds at that address. The format used by an expression is that of its leftmost operand. The symbols **d**, **i**, and **u** name specific segments in the absence of other symbols.

Displaying Information

To display information about *program*, use an expression of the form

[*address***][,***count***]?[***format***]**

This displays *format* for *count* iterations, starting at *address*. The symbol '.' represents the *address*, which defaults to the current display address if omitted. *count* defaults to one. The *format* string consists of one or more of the following characters:

^	Reset display address to '.'
+	Increment display address
-	Decrement display address
b	Byte
c	**char**; control and non-**char**s escaped
C	Like 'c' except '\0' not displayed
d	Decimal
f	**float**
F	**double**
i	Machine instruction, disassembled
l	**long**
n	Output '\n'
N	NDP (80387) register
O	octal
p	Symbolic address
s	String terminated by '\0', with escapes
S	String terminated by '\0', no escapes
u	**unsigned**
w	word
x	Hexadecimal
Y	time (as in i-node, etc.)

The format characters **d**, **o**, **u**, and **x**, specify a numeric base. Each of these can be followed by **b**, **l**, or **w**, which specify a datum size, to describe a single datum for display. A format item may also be preceded by a count that specifies how many times the item is to be applied. *format* defaults to the previously set format for the segment (initially **o** for data and u-area, and **i** for instructions). Except where otherwise noted, **db** increments the display address by the size of the datum displayed after each format item.

Execution Commands

In the following commands, *address* defaults to the address where execution stopped, unless otherwise specified; *count* and *expr* default to one. *commands* is an arbitrary string of **db** commands, terminated by a newline. A newline may be included by preceding it with a backslash '\'.

[*address***]=**
Print *address* (offset) in hexadecimal. *address* defaults to '.'.

[*address***]=***value***[,***value***[,***value***]...]**
Patch *value* into the program, beginning at point *address*. The address defaults to '.'. You can list up to ten *value*s. The command **=** assigns values to sequential locations in the traced process. **db** determines the size of the assigned value from the last display format used. You can set and display the registers of the traced process, just like any other address in the traced process.

? Print the last error message.

[*address***][,***n***]?[***ft***]**
Display formatted information. *ft* indicates the format, which must be one of **bcCdfFilnNopsSuvwxY**. For details, see the command **:hf**, below

*address***?**
Print *address*.

!*command*
Pass *command* to a shell for execution.

[*address***] :a**
Print *address* symbolically. *address* defaults to '.'.

[*address***]:b[***commands***]**
Set a breakpoint at *address*. Execute *commands* when the breakpoint is encountered. *commands* defaults to **i+.:a\ni+.?i\n:x\n** — that is, print the breakpoint address, disassemble the instruction at the breakpoint address, and read more commands from the console.

:br [*commands***]**
Set breakpoint at return from current routine, and execute *commands*. The default *commands* are the same as for **:b**, above.

LEXICON

[*address***] :c**
Continue execution from *address*.

[*address***] :d[r][s]**
Delete the breakpoint previously set at *address*. If the optional **r** or **s** is specified, delete return or single-step breakpoint. *address* defaults to '.'.

[*address***]:e[***commandline***]**
Begin traced execution of the object file at *address* (default, entry point). **db** parses *commandline* and passes it to the traced process. **argv[0]** must be typed directly after **:e** if supplied. For example,

```
:eprogname foo bar baz
```

sets **argv[0]** to **progname**, **argv[1]** to **foo**, **argv[2]** to **bar**, and **argv[3]** to **baz**. Quotation marks, apostrophes, and redirection are parsed as by the shell, but special characters '?*[]' and shell punctuation '(){}|;' are not. For complete shell command line parsing use the **-e** option, above.

Note that you must use the **:e** command to start the program execution prior to using the single-step, trace-back, or display-register commands. For example, the following COHERENT command sequence sets a breakpoint at **main()**, begins execution, and single-steps ten times through the program after having reached the breakpoint:

```
main:b
:e
,10:s
```

:f Print type of fault that caused a core dump or stopped the traced process.

:h Print help information.

:hf Print help information about display formats. **db** recognizes the following display formats:

b	Byte.
c	**char**; control, and non-**char**s printed as escape sequences.
C	**char**; control and non-**char**s print as '.'.
d	Decimal.
f	**float**.
F	**double**.
i	Disassembled machine instruction.
l	**long**.
n	Output '\n'.
N	NDP (80387) floating-point register (ten bytes).
o	Octal.
p	Symbolic address.
s	String (NUL-terminated) with escape sequences.
S	String (NUL-terminated).
u	**unsigned**.
v	File system l3-block address (three bytes).
w	Word.
x	Hexadecimal.
Y	Time.

Options **d**, **o**, **u**, and **x** specify numeric bases (decimal, octal, unsigned decimal, hexadecimal). Each may be followed by **b**, **w**, or **l** to indicate a datum size (respectively, byte, word, or long).

:m Display segmentation map.

:p Display all breakpoints.

[*expr***] :q**
If *expr* is nonzero, quit the current level of command input (see **:x**). *expr* defaults to one. End-of-file is equivalent to **:q**.

:r Display the contents of all registers on the microprocessor.

:rN Display the contents of all registers on the microprocessor and on the numeric co-processor. If your system does not possess a numeric co-processor, it displays the contents of the pseudo-registers used by COHERENT's emulator.

[*address***][,***count***]:s[c][***commands***]**
Single-step execution starting at *address*, for *count* steps, executing *commands* at each step. *commands* defaults to **i+.?i**.

After a single-step command, **<Enter>** is equivalent to **.,1:s[c]**. The option **c** tells **db** to turn off single-stepping at a subroutine call and turn it on again upon return.

[*depth***] :t**
Print a call traceback to *depth* levels. If *depth* is zero (default), unwind the whole stack.

[*expr***] :x**
If *expr* is nonzero, read and execute commands from the standard input up to end of file or to receiving the command **:q**. *expr* defaults to one.

Note that the **:c**, **:s**, **:t**, and **:r** commands cannot be executed before a program is started. If you are debugging the program **hello**, do the following first:

```
db hello
main:b
:e
```

This invokes the debugger for **hello** and advances it to **main**. Now you can use the full set of commands.

Examples

The first example uses **db** to examine a program named **myprog**, which has core-dumped. To debug it, use the command

```
db myprog core
```

You could then issue the following commands to see where the problem lay:

:f This command displays the fault that caused the core dump.

:r This displays the contents of registers at the point where the program core dumped.

:t This command traces back the stack. With this command, you can see how your program arrived at the point where it core dumped. You can use this to find the point in your code where the program "jumps the rails"; often, this is all the information you need to fix the fault.

i1? This prints the value of global variable **i1** in your program at the time of the core dump.

:q Quit **db**. At this point, you should have a good idea of what went wrong with your program.

For another example, consider the following program, named **segv.c**:

```
main()
{
      register char *cp;

      cp = &main;
      *cp = 1000;
}
```

Compile this program with the command **cc segv.c**. To run it, type **segv**; as you can see, it crashes with a segmentation violation, producing a core-dump file named **core**. Now, you can use **db** to find out why the program core dumped.

To invoke the debugger, type:

```
db segv core
```

Now, type the **db** command:

```
:f
```

This tells **db** to print the type of fault that caused the program to dump core. **db** replies:

```
segmentation violation
```

Now, type:

LEXICON

```
*%eip?
```

db replies:

```
000000E9     movb (%ebx), $0xE8
```

Here, **db** gives you the value of the instruction pointer register **%eip** when the segmentation violation occurred and disassembles the instruction at that location. The offending instruction is trying to store indirectly through register **%ebx**. Type:

```
:t
```

db prints a traceback of the call stack:

```
7FFFFD24  000000E9  main(1, 0x7FFFFD38, 0x7FFFFD40)
```

This shows the program was in **main()** and not in any other function. Type:

```
:r
```

db prints contents of the machine registers:

```
%cs =000B       %eip=000000E9   %ss =0013       %fw =00011246
%ds =0013       %es =0013       %fs =0000       %gs =0000
%eax=00000001   %ebx=000000D4   %ecx=00000013   %edx=7FFFFD40
%esp=7FFFFD1C   %ebp=7FFFFD24   %edi=004090F4   %esi=00400D24
```

This shows that register **%ebx** has the value 0xD4 at the time of the core dump. Print the contents of **%ebx** symbolically:

```
%ebx?p
```

db replies:

```
00000020     main
```

The program is trying to store into the address of **main**. This causes a segmentation violation because COHERENT does not allow programs to write on code. Finally, type

```
:q
```

to exit from **db**.

In the last example, suppose you want to print the current address, the instruction at the current address, and the contents of global variable **j** when you hit function **fn** while running **db**. Type:

```
db cmd
main:b
:e
fn:b.:a\
.?i\
j?\
:x
```

The backslash '\' at the end of a line "escapes" a newline — that is, it tells **db** to ignore the newline, and concatenate the contents of the next line onto those of the present line. Thus, the **fn** command line (four physical lines with escaped newlines) forms a single **db** command that says the following:

.:a Print the current position as an address.
.?i Print the contents of the current position as an instruction.
j? print the contents of **j**.
:x Read more **db** input from the console.

The **:x** is necessary if you want to keep debugging interactively after **db** executes the breakpoint command list!

See Also

commands, coff.h, core, l.out.h, od, ptrace()

dbm.h — Header File

Header file for DBM routines
#include <dbm.h>

Header file **<dbm.h>** declares the functions used to manipulate DBM data bases:

dbmclose() . . . . . Close a DBM data base
dbminit() . . . . . Open a DBM data base
delete() . . . . . Delete a record from a DBM data base
fetch() . . . . . Fetch a record from a DBM data base
firstkey() . . . . . Retrieve the first record from a DBM data base
nextkey() . . . . . Retrieve the next record from a DBM data base
store() . . . . . Write a record into a DBM data base

It also defines the structure **datum**, which holds a data element, either a key or its associated data set:

```
typedef struct {
      char *dptr;
      int dsize;
} datum;
```

See Also

gdbm.h, header files, libgdbm, ndbm.h

Notes

Please note that function **dbmclose()** is non-standard. A program that uses it cannot be recompiled on an orthodox UNIX system.

For a statement of copyright and permissions on this header file, see the Lexicon entry for **libgdbm**.

dbm_clearerr() — NDBM Function (libgdbm)

Clear an error condition on an NDBM data base
#include <ndbm.h>
dbm_clearerr (*database***)**
DBM ******file***;**

Macro **dbm_clearerr()** clears an error that had been set on *database*.

See Also

Notes

As of this writing, this macro in fact does nothing.

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

dbm_close() — NDBM Function (libgdbm)

Close an NDBM data base
#include <ndbm.h>
void dbm_close (*database***)**
DBM ******database***;**

Function **dbm_close()** closes the NDBM data base to which *database* points. *database* must first have been opened by a call to **dbm_open()**.

See Also

Notes

This function is a wrapper for function **gdbm_close()**. It is included for compatibility with existing code.

If you have called **dbm_fetch()** to select data from *database*, you must use or copy the returned information before you call **dbm_close()**. If you do not, **dbm_close()** may corrupt the data in the **datum** that **dbm_fetch()** has returned.

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

dbm_delete() — NDBM Function (libgdbm)

Delete records from an NDBM data base
#include <ndbm.h>
int dbm_delete (*database***,** *key***)**
DBM ******database***;**
datum *key***;**

Database function **dbm_delete()** deletes the record with *key* from the data base to which *database* points. *database* must have been opened by a call to **dbm_open()**.

If all goes well, **dbm_delete()** returns zero. It returns -1 if *database* did not contain a record with *key*, or if *database* were opened into read-only mode.

See Also

Notes

This function is a wrapper for function **gdbm_delete()**. It is included for compatibility with existing code.

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

dbm_dirfno() — NDBM Function (libgdbm)

Return the file descriptor for an NDBM .dir file
#include <ndbm.h>
int dbm_dirfno (*database***)**
DBM ******database***;**

A NDBM data base consists of two files. One, with the suffix **.dir**, holds the index for the data base; the other, with the suffix **.pag**, holds the data themselves.

Function **dbm_dirfno()** returns the file descriptor for the **.dir** file associated with the data base to which *database* points. *database* must have been returned by a call to **dbm_open()**.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

dbm_error() — NDBM Function (libgdbm)

Check a NDBM data base for an error
#include <ndbm.h>
int dbm_error (*database***)**
DBM ******database***;**

Macro **dbm_error()** checks *database* for an error condition, should a call to another data-base function fail.

See Also

Notes

Under the GDBM package, this macro in fact does nothing. It is included simply for compatibility with existing software.

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

dbm_fetch() — NDBM Function (libgdbm)

Fetch a record from an NDBM data base
#include <ndbm.h>
datum dbm_fetch (*database***,** *key***)**
DBM ******file***;**
datum *key***;**

Function **dbm_fetch()** retrieves from *database* the record with the given *key*. *database* must first have been opened through a call to function **dbm_open()**.

dbm_fetch() returns the address of the record it has retrieved. It returns NULL either if something went wrong (e.g., it could not read *database*), or if *database* does not contain a record with *key*.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

dbm_firstkey() — NDBM Function (libgdbm)

Retrieve the first key from an NDBM data base
#include <ndbm.h>
datum dbm_firstkey (*database***)**
DBM **database***;*

Function **dbm_firstkey()** retrieves the record with the first key in *database*. *database* must first have been opened through a call to function **dbm_open()**.

dbm_firstkey() returns the address of the record it has retrieved. It returns NULL either if something went wrong (e.g., it could not read *database*), or if *database* is empty.

You can use **dbm_firstkey()** with function **dbm_nextkey()** to walk through *database*. For example:

```
for (key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db))
```

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

dbm_nextdbm() — NDBM Function (libgdbm)

Retrieve the next key from an NDBM data base
#include <ndbm.h>
datum dbm_nextkey (*database***)**
DBM **database***;*

Function **dbm_nextkey()** retrieves the next key from the data base to which *database* points. *database* must first have been opened via a call to function **dbm_open()**, and had the first key retrieved from it via a call to function **dbm_firstkey()**.

dbm_nextkey() returns the address of the record it has retrieved. If something has gone wrong, it returns NULL. If the last record within *database* has already been retrieved, it returns a record whose field **dptr** is NULL.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

dbm_open() — NDBM Function (libgdbm)

Open an NDBM data base
#include <ndbm.h>
DBM **dbm_open (***database***,** *type***,** *mode***)**
char **database***;*
int *type***,** *mode***;**

Function **dbm_open()** opens *database*. Parameters *type* and *mode* are the same as for the system call **open()**; for details, see its Lexicon entry.

To close *database*, call **dbm_close()**.

dbm_open() returns the address of the name of the data base it has opened. If something has gone wrong, it returns NULL.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

dbm_pagfno() — NDBM Function (libgdbm)

Return the file descriptor for an NDBM .pag file
#include <ndbm.h>
int dbm_pagfno (*database***)**
DBM ******database***;**

A NDBM data base consists of two files. One, with the suffix **.dir**, holds the index for the data base; the other, with the suffix **.pag**, holds the data themselves.

Function **dbm_pagfno()** returns the file descriptor for the **.pag** file associated with the data base to which *database* points. *database* must have been returned by a call to **dbm_open()**.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

dbm_rdonly() — NDBM Function (libgdbm)

Set an NDBM data base into read-only mode
#include <ndbm.h>
int dbm_rdonly (*database***)**
DBM ******database***;**

Function **dbm_rdonly()** puts an NDBM data base into read-only mode. *database* points to the data base; it must have been returned by a call to **dbm_open()**.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

dbm_store() — NDBM Function (libgdbm)

Store a record into an NDBM data base
#include <ndbm.h>
int dbm_store (*database***,** *key***,** *content***,** *flags***)**
DBM ******database***;**
datum *key*, *content*;
int *flag*;

Function **dbm_store()** inserts a record into *database*, which must first have been opened by a call to **dbm_open()**. *content* points to the data to be stored within the data base; and *key* points to the key under which the data are to be stored.

flag indicates how the the data are to be inserted into the data base. If it is set to **DBM_INSERT**, the data are appended onto the data base as new records; in this case, **dbm_store()** will not modify existing records that have an identical *key*. If, however, you set *flag* to **DBM_REPLACE**, *contents* replaces any existing record with an identical *key*.

If all goes well, **dbm_store()** returns zero. It returns a negative value should an error occur. If you set *flag* to **DBM_INSERT** and **dbm_store()** finds that *database* already contains a record with *key*, it returns one.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libdbm**.

dbmclose() — DBM Function (libgdbm)

Close a DBM data base
#include <dbm.h>
void dbmclose()

Function **dbmclose()** closes a DBM-style data base. *database* points to the data base to be closed; it must have been returned by a call to function **dbminit()**.

See Also

Notes

This function is non-standard. A program that uses it cannot be recompiled on an orthodox UNIX system.

dbminit() — DBM Function (libgdbm)

Open a DBM data base
#include <dbm.h>
int dbminit(*database***)**
char ******database***;**

Function **dbminit()** opens and initializes a DBM data base. *database* points to the name of the data base to open.

Please note that unlike the GDBM function **gdbm_open()** or the DBM function **dbm_open()**, **dbminit()** does not create a data base — it merely opens it for manipulation. If the data base does not exist, you must first create it. To do so, create the empty files *database***.pag** and *database***.dir**.

If all goes well, **dbminit()** returns zero. If something goes wrong, it returns a negative value.

See Also

Notes

dc — Command

Desk calculator
dc [*file***]**

dc is an arbitrary precision desk calculator. It simulates a stacking calculator with ancillary registers. Input must be entered in reverse Polish notation. **dc** maintains the expected number of decimal places during addition, subtraction, and multiplication, but the user must make an explicit request to maintain any places at all during division.

dc reads input from *file* if specified, and then from the standard input. **dc** accepts an arbitrary number of commands per line; moreover, spaces need not be left between them.

The *scale factor* of a number is the number of places to the right of its decimal point. The *scale factor register* controls decimal places in calculations. The scale factor does not affect addition or subtraction. It affects multiplication only if the sum of the scale factors of the two operands is greater than it. The result of every division command has as many decimal places as it specifies. It affects exponentiation in that multiplication is performed as many times as the integer part of the exponent indicates; any fractional part of the exponent is ignored.

dc recognizes the following commands and constructions:

number
: Stack the value of *number*. A number is a string of symbols taken from the digits '0' through '9', and the capital letters 'A' through 'F' (usual hexadecimal notation), with an optional decimal point. An underscore '_' as a prefix indicates a negative number. The letters retain values ten through 15, respectively, regardless of the base chosen by the user.

+ - / * % ^
: The arithmetic operations: addition(+), subtraction(-), division(/), multiplication(*), remainder(%), and exponentiation(^). **dc** pops the two top stack elements, performs the desired operation by calling the multiprecision routine desired (see **multiprecision arithmetic**), and stacks the result.

c Clear the stack.

d Duplicate the top of the stack (so that it occupies the top two positions of the stack).

f Print the contents of the stack and the values of all registers.

i Remove the top of the stack and use its integer part as the assumed input base (default, ten). The new input base must be greater than one and less than 17.

I Stack the current assumed input base.

k Remove the top of the stack and put it in the internal scale factor register.

K Put the value of the internal scale register (which the **k** command sets) on the top of the stack.

l *x* Load the value of register *x* to the top of the stack. The value of register *x* is unaltered. *x* may be any character.

o Remove the top of the stack and use its integer part as the assumed output base (default, ten). The specified base may be any positive integer.

O Stack the current assumed output base.

p Print the top of the stack. The value remains on the stack.

q Quit the program; control returns to the shell **sh**.

s *x* Remove the top of the stack and store it in register *x*. The previous contents of *x* are overwritten. *x* may be any character.

v Replace the top of the stack by its square root.

x Remove the top of the stack, interpret it as a string containing a sequence of **dc** commands, and execute it.

X Replace the top of the stack by its scale factor (i.e., the number of decimal places it has).

z Place the number of occupied levels of the stack on top of the stack.

[...] Place the bracketed character string on top of the stack. The string may be executed subsequently with the **x** command.

<*x* **>***x* **=***x* **!<***x* **!>***x* **!=***x*
Remove the top two elements of the stack and compare them. If there is no '!' sign before the relation, execute register *x* if the two elements obey the relation. If a '!' sign is present, execute register *x* if the elements do not obey the relation.

! Interpret the rest of the line as a command to the shell **sh**. Control returns to **dc** after command execution terminates.

Example

The following example program prints the first 20 Fibbonacci numbers. The character **l** is printed in boldface to help you tell from a numeric one.

```
1sa1sb1sc
[lalbdsa+psblc1+dsc21<y]sy
lyx
```

See Also

bc, commands

Notes

For most purposes, the in-fix notation of **bc** is more convenient than the Polish notation of **dc**.

dcheck — Command

Check directory consistency
dcheck [-s] [-i *inumber...***]** *filesystem ...*

dcheck checks the consistency of each *filesystem*. It scans all the directories in each *filesystem* and counts all i-nodes referenced. It then compares its counts against the link counts maintained in the i-nodes. **dcheck** notes any discrepancies, and notes allocated i-nodes with a link count of zero.

The **-i** argument tells **dcheck** to compare each *inumber* in the list against those in each directory. It reports matches by printing the i-number, the i-number of the parent directory, and the name of the entry. The **-s** argument tells **dcheck** to correct the link count of errant i-nodes to the entry count.

Because **dcheck** uses two passes to check a *filesystem*, the file system should be unmounted. If **-s** is used on the root file system, the system should be rebooted immediately (without performing a **sync**). The raw device should be used.

See Also

check, commands, icheck, ncheck, sync, umount

Diagnostics

If the link count is zero and there are entries, the file system must be mounted and all entries removed immediately. If the link count is nonzero and the entry count is *larger,* the **-s** option must be used to make the counts agree. In all other cases there may be wasted disk space but there is no danger of losing file data.

Notes

In earlier releases of COHERENT, **dcheck** acted upon a default file system if none was specified.

This command has largely been replaced by **fsck**.

dd — Command

Convert the contents of a file
dd [*option=value***] ...**

dd copies an input file to an output file, while performing requested conversions. Options include case and character set conversions, byte swapping conversion for other machines, and different input and output buffer sizes. **dd** can be used with raw disk files or raw tape files to do efficient copies with large block (record) sizes. Read and write requests can be changed with the **bs** option described below.

The following list gives each available *option*. Any numbers which specify block sizes or seek positions may be written in several ways. A number followed by **w**, **b**, or **k** is multiplied by two (for words), 512 (for blocks), or 1,024 (for kilobytes), respectively, to obtain the size in bytes. A pair of such numbers separated by **x** is multiplied together to produce the size. All buffer sizes default to 512 bytes if not specified.

bs=*n* Set the size of the buffer for both input and output to *n* bytes.

cbs=*n* Set the conversion buffer size to *n* bytes (used only with character set conversions between ASCII and EBCDIC).

conv=*list* Perform conversions specified by the comma-separated *list*, which may include the following:

ascii	Convert EBCDIC to ASCII
ebcdic	Convert ASCII to EBCDIC
ibm	Convert ASCII to EBCDIC, IBM flavor
lcase	Convert upper case to lower
noerror	Continue processing on I/O errors
swab	Swap every pair of bytes before output
sync	Pad input buffers with 0 bytes to size of **ibs**
ucase	Convert lower case to upper

count=*n* Copy a maximum of *n* input records.

files=*n* Copy a maximum of *n* input files (useful for multifile tapes).

ibs=*n* Set the input buffer size to *n* (normally used if input and output blocking sizes are to be different).

if=*file* Open *file* for input; the standard input is used when no **if=** option is given.

obs=*n* Set the output buffer size to *n.*

of=*file* Open *file* for output; the standard output is used when no **of=** option is given.

seek=*n* Seek to position *n* bytes into the output before copying (does not work on stream data such as tapes, communications devices, and pipes).

skip=*n* Read and discard the first *n* input records.

Examples

The first example copies the entire contents of a 1.44-megabyte, 3.5-inch diskette from drive 0 to file **disk.dd**:

```
dd if=/dev/fva0 of=disk.dd bs=36b count=80
```

The second example writes the contents of the previously stored 5.25-inch file **backup.dd** to a 1.2-megabyte, 5.25-inch floppy disk in drive 1:

```
dd if=backup.dd of=/dev/fha1 bs=30b count=80
```

See Also

ASCII, commands, conv, cp, tape, tr

Diagnostics

The command reports the number of full and partial buffers read and written upon completion.

Notes

Because of differing interpretations of EBCDIC, especially for certain more exotic graphic characters such as braces and backslash, no one conversion table will be adequate for all applications. The **ebcdic** table is the American Standard of the Business Equipment Manufacturers Association. The **ibm** table seems to be more practical for line printer codes at many IBM installations.

decvax_d() — General Function (libc)

Convert a double from IEEE to DECVAX format
int
decvax_d(*ddp***,** *idp***)**
double **ddp*, **idp*;*

decvax_d() converts a **double** from IEEE format to DECVAX format. *idp* points to the IEEE-format **double** to convert. *ddp* points to a destination for the converted DECVAX value; *ddp* may be identical to *idp* for in-place conversion.

decvax_d() returns zero on success, -1 on underflow, or one on overflow.

For a description of the IEEE and DECVAX formats for floating-point numbers, see the Lexicon article for **float**.

See Also

decvax_f(), float, ieee_d(), ieee_f(), libc,

decvax_f() — General Function (libc)

Convert a float from IEEE to DECVAX format
int
decvax_f(*dfp***,** *ifp***)**
float **dfp*, **ifp*;*

decvax_f() converts a **float** from IEEE format to DECVAX format. *ifp* points to the IEEE-format **float** to convert. *dfp* points to a destination for the converted DECVAX value; *dfp* may be identical to *ifp* for in-place conversion.

decvax_f() returns zero on success, -1 on underflow, or one on overflow.

For a description of the IEEE and DECVAX formats for floating-point numbers, see the Lexicon article for **float**.

See Also

decvax_d(), float, ieee_d(), ieee_f(), libc

default — C Keyword

Default label in switch statement

default is a prefix used in **switch** statement. If none of the **case** labels match the parameter in the **switch** statement, then the **default** label is used. A **switch** is not required to have a **default** case, but it is good programming practice to use one.

See Also

C keywords, case, switch
ANSI Standard, §6.6.4.2

defined — Preprocessor Operator

Perform an action if a macro is defined

The preprocessor directive **defined** determines whether a symbol is defined to the **#if** preprocessor directive. For example,

```
#if defined(SYMBOL)
```

or

```
#if defined SYMBOL
```

is equivalent to

```
#ifdef SYMBOL
```

except that it can be used in more complex expressions, such as

```
#if defined FOO && defined BAR && FOO==10
```

defined is recognized only in lines beginning with **#if** or **#elif**.

See Also

#elif, #if, #ifdef, cpp, C preprocessor
ANSI Standard, §6.8.1

Notes

Note that **defined** is a preprocessor operator, not a preprocessor directive or a C keyword. The difference lies in the fact that you could write a function called **defined()** without any complaint from the C compiler; and if **defined** does not appear within an **#if** or **#elif** directive, the preprocessor ignores it.

deftty.h — Header File

Define default tty settings
#include <sys/deftty.h>

deftty.h defines the default tty settings.

See Also

header files

delete() — DBM Function (libgdbm)

Delete a record from a DBM data base
#include <dbm.h>
int delete (*key***)**
datum *key***;**

Function **delete()** deletes the record with *key* from the currently opened data base. The data base must first have been opened by a call to **dbminit()**.

If all goes well, **delete()** returns zero. If an error occurs, it returns a negative value.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

deroff — Command

Remove text formatting control information
deroff [-w] [-x] [*file* **...]**

deroff removes text formatting control information from each input text *file*, or from the standard input if no *file* is specified. It regards all lines that begin with '.' or '" as being **nroff** or **troff** commands and deletes them. **deroff**

also recognizes some additional control lines. It deletes **eqn** information (between **.EQ** and **.EN** lines), **tbl** information (between **.TS** and **.TE** lines), and macro definitions. It also deletes embedded **.eqn** requests. It expands source file inclusion with **.so** and **.nx** requests, with the proviso that no input file is read twice. It also deletes some **troff** escape sequences, such as those for font and size change.

When the **-x** flag is present, **deroff** uses some additional knowledge about the **nroff -ms** macro package.

When the **-w** flag is present, **deroff** divides the remaining text into words and prints them to the standard output, one per line. A **word** comprises a sequence of letters, digits, and apostrophes that commences with a letter. **deroff** strips apostrophes from the output. All other characters between words are not printed. The spelling checking programs **spell** and **typo** use this option.

See Also

commands, nroff, spell, troff, typo

detab — Command

Replace tab characters with spaces
detab [*tabsize***]**

The command **detab** reads the standard input, replaces every tab character with spaces, and writes the result to the standard output.

detab assumes that a tab stop occurs every *tabsize*, which must be an integer greater than one and less than 257. If you do not supply a *tabsize*, **detab** assumes that a tab stop occurs every eight characters. You can also override the default tab size by setting the environmental variable **TABSIZE** to a value other than eight.

See Also

commands

device drivers — Overview

A *device driver* is a program that controls the action of one of the physical devices attached to your computer system. The following table lists the device drivers included with the COHERENT system. The first field gives the device's major device number; the second gives its name; and the third describes it. If a major number does not appear in this table, that number is available for a driver yet to be written.

0:	**clock**	System clock
0:	**cmos**	System CMOS
0:	**freemem**	Amount of memory that is free at any given moment
0:	**idle**	System idle time
0:	**kmem**	Device to manage kernel memory
0:	**kmemhi**	
0:	**mem**	Interface to memory and null device
0:	**null**	The "bit bucket"
0:	**ps**	Processes currently being executed
1:	**ct**	Controlling terminal device (**/dev/tty**)
2:	**console**	Video module for console (**/dev/console**)
2:	**vtkb**	Non-configurable keyboard driver, virtual consoles
2:	**vtnkb**	Configurable keyboard driver, virtual consoles
2:	**mm**	The video driver
3:	**lp**	Parallel line printer
4:	**fd**	Floppy-disk drive
4:	**fdc**	765 diskette and floppy-tape controller
4:	**ft**	Floppy-tape drive
5:	**asy**	Serial driver
6:	**tr**	Trace driver
8:	**rm**	Dual RAM disk
9:	**pty**	Pseudoterminals
11:	**at**	AT hard disk
13:	**hai**	Host adapter-independent SCSI driver
14:	**cdu31**	Sony CD-ROM drives
16:	**mcd**	Mitsumi CD-ROM drives

Please note that the devices with major number 0 are not portable, and non-DDI/DKI. Also note that in future releases of COHERENT, the **hai** driver will be divided into several optional SCSI host-bus adapters (HBAs) and target devices.

It is not unusual for one major number to admit several driver service modules. Instances of this include the following major numbers:

0 This number is for a number of system-dependent drivers.

2 This number supports the console, both its keyboard modules and its video modules.

4 This describes varieties of floppy-disk and floppy-tape controllers and drives.

13 This describes a number of SCSI host modules, HBA modules, and target modules.

Major and Minor Numbers

COHERENT uses a system of *major* and *minor* device numbers to manage devices and drivers. In theory, COHERENT assigns a unique major number to each type of device, and a unique minor number to each instance of that type. In practice, however, a major number describes a device driver (rather than a device *per se*). The individual devices serviced by that driver are identified by a minor number. Sometimes, certain parts of the minor number specify configuration. For example, bits 0 through 6 of the minor number for COHERENT RAM disks indicate the size of the allocated device.

Optional Kernel Components

The kernel also contains the following optional components:

em87	Emulate hardware floating-point routines
msg	Perform System V-style message operations
sem	Perform System V-style semaphore operations
shm	Perform System V-style shared-memory operations
streams	Perform STREAMS operations

These components resemble device drivers, in that that they perform discreet work and can be linked into or excluded from the kernel, as shown below. However, they do not perform I/O with a device, and so are not true drivers. For details on these modules, see their entries in the Lexicon.

Configuring Drivers and the Kernel

Beginning with release 4.2, COHERENT lets you tune kernel and driver variables, enable or disable drivers, and easily build a new bootable kernel that incorporates your changes.

The command **idenable** lets you enable or disable a driver within the kernel. The command **idtune** lets you set a user-modifiable variable within the kernel. Finally, the command **idmkcoh** generates a new kernel that incorporates all changes you have made with the other three commands. Changes are entered with **idenable** and **idtune** do not take effect until you invoke **idmkcoh** to generate a new kernel, and boot the new kernel. Scripts **/etc/conf/*/mkdev** simpify the choices of **idenable** and **idtune** during installation and reconfiguration: they invoke **idtune** and **idenable** For details, see these commands' entries in the Lexicon.

Adding a New Device Driver

The commands described above make it easy for you to add a new device driver to your COHERENT kernel.

The following walks you through the processing of adding a new driver. We will add the driver **foo**, which enables the popular "widget" device. Please note that this example has the user modify the files **mtune** and **stune** by hand. It is not a good idea for you to do this; however, we describe how to do this to show how these files fit into the process of building a new kernel:

1. To begin, log in as the superuser **root**.

2. The next step is to create a directory to hold the driver's sources and object. Every driver must have its own directory under directory **/etc/conf**; and the sources must be held in directory **src** in that driver's directory. In this case, create directory **/etc/conf/foo**; then create directory **/et/conf/foo/src**.

3. Copy the sources for the driver into its source directory; in this case, copy them into **/etc/conf/foo/src**.

4. Create a **Makefile** in your driver's source directory, e.g., **/etc/conf/foo/src/makefile**. The easiest way to see what is required is to review several of the driver **Makefile**s shipped in the COHERENT driver kit. You can perform a test compilation of your driver by running **make** with the driver's **src** directory as the current

directory. This should create one object file that has the suffix **.o**. Copy this file in the driver's home directory, and name it **Driver.o**. In this case, the object for the driver should be in file **/etc/conf/foo/Driver.o**. In some rare cases, a driver compile into more than one object. You should store all of these objects into one archive; name the archive **Driver.a** and store it in the driver's home directory. The COHERENT commands that build the new kernel know how to handle archives correctly. The main idea is that files **Space.c** (if one exists) and **Driver.o** or **Driver.a** be placed in the driver directory, i.e., the parent of the **src** directory.

5. Add an entry to file **/etc/conf/sdevice** for this driver. **sdevice**, as described above, names the drivers to be included in the kernel. The entries for practically every entry are identical; you need to note only that the second column marks whether to include the driver in the kernel. In this case, the entry for the driver **foo** should read as follows:

```
foo   Y     0     0     0     0     0x0   0x0   0x00x0
```

For details on what each column means, read the comments in file **/etc/conf/sdevice**.

6. Add an entry to file **/etc/conf/mdevice** for the new driver. This file is a little more complex than **sdevice**; in particular, it distinguishes between STREAMS-style drivers and "old-style" COHERENT drivers. In most cases, you can simply copy an entry for an existing driver of the same type, and modify it slightly. In this case, the entry for **foo** should read as follows:

```
# full  func   misc   code    block  char   minor  minor  dma   cpu
# name  flags  flags  prefix  major  major  min    max    chan  id
foo     -      CGo    foo     15     15     0      255    -1    -1
```

In almost every case, the full name and the code prefix are identical. The code prefix also names the directory that holds the driver's object. Function flags are always always a hyphen, and miscellaneous flags almost always CGo. The block-major and character-major numbers again are almost always identical. The major number is usually assigned by the creator of the device driver. In future releases of the kernel, these will be assigned dynamically by the kernel itself; poorly written drivers that depend upon the driver having a magic major-device number will no longer work. Finally, the last four columns for non-STREAMS drivers are almost always 0, 255, -1, and -1, respectively. See the comments in file **/etc/conf/mdevice**.

7. If the driver has tunable variables, these should be set in the file **Space.c**, which should be stored in the driver's home directory. As it happens, **foo** does not need a **Space.c** file. For examples of such files, look in the various sub-directories of **/etc/conf**.

8. Type the command **idmkcoh** to build a new kernel. If necessary, move the new kernel into the root directory; you cannot boot it until it is in the root directory.

9. Save the old kernel and link the newly build kernel to **/autoboot**. You want save the old kernel, just in case the new one doesn't work. For directions on how to boot a kernel other than **/autoboot**, see the Lexicon entry for **booting**.

10. Back up your files! With a new driver in your kernel, it's best to play it safe.

11. Reboot your system to invoke the new kernel. If all goes well, you will now be enjoying the services of the new device driver.

For scripts on how to add or remove individual drivers from your kernel, see the article of the driver in question.

See Also

Administering COHERENT, asy, at, boot, console, ct, em87, floppy disk, hard disk, idle, kernel, lp, mboot, mdevice, mem, msg, mtune, null, pty, sdevice, sem, sgtty, shm, STREAMS, stty, stune, tape, termio

Notes

Note that in future releases of COHERENT, major numbers will not be static, as they are in the above table. Rather, they will be assigned by the **config** script when you install COHERENT onto your system. This scheme will allow more flexible arrangements of drivers, and will also allow COHERENT to support more than 32 drivers at once. If you write code to work with device drivers, you should *not* make any assumptions about a given driver's major or minor number.

See the Release Notes for your release of COHERENT for a full list of supported devices and device drivers.

Source code for almost all COHERENT device drivers is published in the COHERENT Device-Driver Kit. The only except is the source for **ft**, which includes proprietary information from manufacturers. Experienced writers of device drivers will find the driver kit a good tool for writing or importing drivers for devices that COHERENT does not

yet support.

df — Command

Measure free space on disk
df [-fiv] [-t*filesys***]** (default format)

df measures the amount of space left free on the file system *filesys*. The file system being measured can reside on a hard disk, floppy disk, or RAM disk. For example, to check the amount of space left on file system **x**, type:

```
df /x
```

If you do not name a *filesys*, **df** prints information only about the file system that you in.

By default, **df** prints three statistics: the number of disk blocks free on this device, the total number of disk blocks in the device, and the percent of total disk blocks that is free. Note that a disk block is 512 bytes (1/2 kilobyte).

df recognizes the following command-line options:

-f Suppress i-node information.

-i Give the percentage of i-nodes available used.

-v Give the percentage of blocks used.

See Also

commands, mkfs

dial — System Administration

File that tells UUCP how to dial a system
/usr/lib/uucp/dial

The file **/usr/lib/uucp/dial** holds information about dialers. A *dialer* is a device, usually a modem, through which **uucico** or **cu** "dials" another computer system. The daemon **uucico** and the command **cu** use the information in this file to talk to dialers.

dial consists of a series of descriptions, each of which describes one dialer. A description consists of one or more commands; each command defines an aspect of how to manipulate the dialer. Descriptions must be separated by one blank line.

The following describes the commands you can use in a description:

dialer *name*
Name the dialer being described. Each description must begin with a **dialer** command. For example, the command

```
dialer trailblazer
```

introduces the description for the device named **trailblazer**. (A name need not be technical: you can also use names like **joe** or **junk_modem**.)

chat *from_modem to_modem ... from_modem*
This command gives the chat script with which **uucico** and **cu** initialize the dialer and have it dial a remote system. **chat** can have any number of arguments: the odd-numbered strings are received from the modem, and the even-numbered ones sent to it. Strings are separated by space character; therefore, no string can contain a literal space character. To represent a space character in a string, use the escape sequence **\s**.

If, at a given point in the conversation, nothing is expected from the modem or is to be sent to it, then use an empty pair of quotation marks as a placeholder.

Please note that unlike the chat script used in file **sys**, the chat script in **dial** contains only the information by which the modem is accessed: it does not contain information about how to log into the remote computer system.

A chat script can contain the following escape sequences:

\D Telephone number of the remote system
\T Telephone number plus dialcode translation
\M Do not require carrier
\m Require carrier, fail if not present
\s Represent a space character

uucico and **cu** use the command **phone** in file **/usr/lib/uucp/sys** to expand the escape sequence **\D**.

The following gives an example chat script:

```
chat  "" ATQ0V1E1L2M1DT\D CONNECT\s2400
```

The pair of quotation marks tells **uucico** (or **cu**) to expect nothing from the modem, and to send immediately the string **ATQ0V1E1L2M1DT** followed by the telephone number of the remote system. This is a typical send string for a Hayes-compatible, 2400-baud modem. The string also sets certain registers within the modem: **Q0V1** turns on verbal result codes, **E1** turns on echoing, and **L2M1** sets the duration and volume of the modem's speaker.

The last string in the chat script gives the *expect string*. This is the string that the modem sends when it has succeeded in connecting with the remote computer system. In this example, if the modem does not send

```
CONNECT 2400
```

then the attempt to call the remote system has failed. This example shows, as noted above, that no string to the command **chat** (or any other command used in **dial**) can contain a space character. To represent a space character within a string, use the escape sequence **\s**.

chat-timeout *seconds*
This command gives the number of seconds to await the expect string from the modem. For example, the command

```
chat-timeout 10
```

tells **uucico** to wait ten seconds for the expected string.

chat-fail *failure_string*
This command defines the string that, when received from the modem, indicates that a connection attempt has failed. **uucico** and **cu** abort when they receive *failure_string*. A dialer's description can have multiple **chat-fail** commands (after all, a call can fail for many different reasons). For example, the commands

```
chat-fail   BUSY
chat-fail   NO\sCARRIER
```

tell **uucico** and **cu** to abort when they receive either the strings **BUSY** or **NO CARRIER**.

chat-seven-bit true | false
If **true**, strip all bits to seven bits before comparing them with the expect string within the **chat** script.

chat-program *program* **[** *arguments* **]**
Run *program* before executing the chat script. The optional *arguments* are passed to *program*. The following escape sequences can be embedded within *arguments*:

\Y Name of the port device
\S Speed of the port
**** A literal backspace character

uucico expands these escape sequences before it passes *arguments* to *command*.

dialtone *stirng*
string is the code sequence that tells the modem to wait for a dial tone (e.g., if you must dial '9' and then pause briefly to get an outside line). **uucico** outputs *string* whenever it encounters a '=' within a telephone number. The default code is a comma.

pause *string*
string is the code sequence that tells the modem to pause for one second. **uucico** outputs *string* whenever it encounters a '-' within a telephone number. The default code is a comma.

carrier true | false
true indicates that the dialer supports the modem carrier signal, and **uucico** therefore will require that that carrier be on. **false** indicates that the dialer does not support the modem carrier signal, and **uucico** therefore will not wait for it.

carrier-wait *seconds*
Wait *seconds* for the carrier signal. The default is 60.

dtr-toggle true | false [true | false]
If the first argument is **true**, toggle DTR before using the modem. If the second argument is **true**, sleep for one second after toggling DTR.

complete-chat *string ... string*
complete-chat-timeout *number*
complete-chat-fail *failure_string*
complete-chat-seven-bit true | false
complete-chat-program *program* **[** *arguments* **]**
These commands define a chat script to be run after the UUCP session has run to completion. They are exactly like their **chat** counterparts described above.

abort-chat *string ... string*
abort-chat-timeout *number*
abort-chat-fail *failure_string*
abort-chat-seven-bit true | false
abort-chat-program *program* **[** *arguments* **]**
These commands define a chat script to be run if the UUCP session has aborted. They are exactly like their **chat** counterparts described above.

complete *string*
abort *string*
These are simplified of the **complete-** and **abort-** chat scripts described above. The former sends *string* to the dialer after a call has completed successfully; the latter sends its *string* after a call has aborted.

protocol-parameter *protocol parameter*
Set a protocol parameter. This command is exactly the same as its counterpart used in file **sys**. For details, see the Lexicon entry for **sys**.

seven-bit true | false
When your system negotiates the protocol to use with the remote system, force your system to accept only a protocol that works over seven-bit connection.

reliable true | false
When your system negotiates the protocol to use with the remote system, force your system to accept only a protocol that works over an unreliable connection.

half-duples true | false
If **true**, then the dialer supports only half-duplex connections. This forces your system to avoid bidirectional protocols during protocol negotiation.

Example

The following gives the entry for a 9600-baud Trailblazer modem:

```
dialer tbfast
chat "" AT\sE0\sQ4\sV1\sS7=60\sS50=255\sS51=255\sS66=0 \
      \sS111=30\sDP\D CONNECT\sFAST
chat-timeout 60
chat-fail BUSY
chat-fail NO\sCARRIER
chat-fail NO\sANSWER
abort-chat "" \d+++\dATH0\sV0\sE0\sQ1\sS0=1
abort-chat "" \d+++\dATH0\sV0\sE0\sQ1\sS0=1
```

Most of the commands in this example are optional. A dialer entry could work with only the first two commands. The following describes each command in detail:

dialer Give the dialer the name **tbfast**.

chat Give the chat script with which **uucico** converses with the modem. It sets a number of 'S' registers, turns echoing off, puts the modem into verbose mode, dials the remote system, and indicates that the signal for success is the string **CONNECT FAST**. Note that normally the chat script must be one unbroken string; this example is broken into two lines so it will fit onto the page. For information on the commands from which you would construct a chat script, see the documentation that comes with your modem.

chat-timeout Tells **uucico** how long to wait before it times out. In this case, wait 60 seconds.

chat-fail Define a string with which the modem indicates failure. In this case, there are three such commands, each naming a different message.

abort-chat
abort-chat These give the strings to send to the modem in the case of, respectively, the successful completion of call or an aborted call. For this entry, the same string is send in either case: it turns off echoing and verbose mode, and turns on auto-answering.

See Also

Administering COHERENT, port, sys, UUCP

Notes

Only the superuser **root** can edit **/usr/lib/uucp/dial**.

The file **dial** supports many commands in addition to the ones described here. This article describes only those commands that might be used in typical UUCP connections. For more information, see the original Taylor UUCP documentation, which is in the archive **/usr/src/alien/uudoc104.tar.Z**.

dialups — System Administration

Name every device that may need an additional password
/etc/dialups

The COHERENT system lets you force classes of users who log in through particular devices to enter an extra password. This helps you protect your system against people who may be try to penetrate it via modem.

The file **/etc/dialups** names every device that may require an additional password. Each device must be named on its own line; for example:

```
/dev/com1l
/dev/com3l
/dev/com3r
```

When a device is named in **/etc/dialups**, **login** looks in file **d_passwd** to see if a password has been linked to user's default shell. This permits you, for example, to ask for an extra password for all users who attempt to log in remotely and who have an interactive shell, while letting UUCP accounts enter without the extra password. For examples, see the Lexicon entry for **d_passwd**.

See Also

Administering COHERENT, d_passwd, login

diff — Command

Compare two files
diff [-bdefh] [-c *symbol***]** *file1 file2*

diff compares *file1* with *file2*, and prints a summary of the changes needed to turn *file1* into *file2*.

Two options involve input file specification. First, the standard input may be specified in place of a file by entering a hyphen '-' in place of *file1* or *file2*. Second, if *file1* is a directory, **diff** looks within that directory for a file that has the same name as *file2*, then compares *file2* with the file of the same name in directory *file1*.

The default output script has lines in the following format:

```
1,2 c 3,4
```

The numbers *1,2* refer to line ranges in *file1*, and *3,4* to ranges in *file2*. The range is abbreviated to a single number if the first number is the same as the second. The command *c* was chosen from among the **ed** commands '*a*', '*c*',

and '*d*'. **diff** then prints the text from each of the two files. Text associated with *file1* is preceded by '< ', whereas text associated with *file2* is preceded by '>'.

The following summarizes **diff**'s options.

-b Ignore trailing blanks and treat more than one blank in an input line as a single blank. Spaces and tabs are considered to be blanks for this comparison.

-c *symbol*
Produce output suitable for the C preprocessor **cpp**; the output contains **#ifdef**, **#ifndef**, **#else**, and **#endif** lines. *symbol* is the string used to build the **#ifdef** statements. If you define *symbol* to the C preprocessor **cpp**, it will produce *file2* as its output; otherwise, it will produce *file1*. This option does *not* work for files that already contain **#ifdef**, **#ifndef**, **#else**, and **#endif** statements.

-e Create an **ed** script that will convert *file1* into *file2*.

-f Produce a script in the same manner as the **-e** option, but with line numbers taken directly from the two input files. This will work properly only if applied from end to beginning; it cannot be used directly by **ed**.

-h Compare large files that have a minimal number of differences. This option uses an algorithm that is not limited by file length, but may not discover all differences.

-d Select the **-h** algorithm only for files larger than 25,000 bytes; otherwise, use the normal algorithm.

Example

For an example of a script that uses this command, see the Lexicon entry for **trap**.

See Also

ed, egrep, commands, zdiff

Diagnostics

diff's exit status is zero when the files are identical, one when they are different, and two if a problem was encountered (e.g., could not open a file).

Notes

diff cannot handle files with more than 32,000 lines. Handing **diff** a file that exceeds that limit will cause it to fail, with unpredictable side effects.

diff3 — Command

Summarize differences among three files
diff3 [-ex3] *file1 file2 file3*

diff3 summarizes the differences among three text files. Each difference encountered is headed by one of the following separators, which categorizes how many of the three input files differ in a given range. The headers are as follows

==== All of the files are different.

====*n* Only the *n*th file differs, where *n* may be 1, 2, or 3.

For each set of changes marked as above, the actual change is indicated for each file using a notation similar to commands to **ed**. For each *filen* the following is printed:

*n: l***a** Text is to be appended after line *l* in *filen*.

*n: l,m***c** The text from line *l* to line *m* is to be changed for *filen*. The original text from *filen* follows this line. If this text is identical for two of the files, only the latter (higher numbered) of the two is printed.

Options are available to print a script of commands to **ed**. Option **-e** tells **diff3** to generate a script that makes all changes between *file2* and *file3* to *file1*. This script is based upon all changes flagged with the separators **====** or **====3**, as described above.

The option **-x** prints only those changes where all three files differ, i.e., those flagged with **====**.

The option **-3** requests only those changes where *file3* differs.

Example

The following command sequence produces a script, applies it to *file1*, and sends the result to the standard output.

```
(diff3 -e file1 file2 file3; echo '1,$p') | ed - file1
```

Files

/tmp/d3*
/usr/lib/diff3

See Also

commands, diff, ed

Diagnostics

An exit status of zero indicates all three files were identical, one indicates differences, and two indicates some other failure.

difftime() — Time Function (libc)

Calculate difference between two times
#include <time.h>
double difftime(*newtime***,** *oldtime***)**
time_t *newtime***,** *oldtime***;**

difftime() subtracts *oldtime* from *newtime*, and returns the difference in seconds. Both arguments are of type **time_t**, which is defined in the header **time.h**.

Example

This example uses **difftime()** to show an arbitrary time difference.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main()
{
      time_t      t1, t2;

      time(&t1);
      printf("Press enter when you feel like it.\n");
      getchar();
      time(&t2);

      printf("You waited %f seconds\n", difftime(t2, t1));
      return(EXIT_SUCCESS);
}
```

See Also

clock(), libc, mktime(), time [overview]
ANSI Standard, §7.12.2.2

directors — System Administration

Describe how to resolve local mail addresses
/usr/lib/mail/directors

The program **smail** reads file **/usr/lib/mail/directors** for the rules on how to resolve addresses on your local host. Please note that under COHERENT, the default configuration of **smail** does not use this file; however, if you wish, you can create it to change **smail**'s default rules for resolving local addresses.

Structure of Configuration Files

smail can use five varieties of configuration files:

- One or two *configuration* files, which perform global configuration of **smail**— including naming the other configuration files.
- One *directors* file, which describes how to deliver mail on your local system.

- One *routers* file, which describes resolve the addresses of remote systems.
- One *transports* file, which describes how to move mail from your system to selected remote systems.
- One *methods* file, which matches hosts with methods of transporting mail.

smail permits you to name these files as you choose; under COHERENT, they are named as follows:

/usr/lib/mail/config
/usr/lib/mail/directors
/usr/lib/mail/methods
/usr/lib/mail/routers
/usr/lib/mail/transports

Each is described in its own Lexicon entry. However, the **directors**, **routers**, and **transports** file all have the same format; the following describes it.

Each file consists of a set of entries; each entry, in turn, describes the attributes of one director, router, or transport. The order of entries in **director** and **router** is important, in that the directors and routers are invoked in the order stated in the file. The order of entries in the transport file is not important.

An entry in one of these files defines the following:

- A name by which that entry is known.
- A driver that implements the function for that entry.
- A set of generic attributes from a set that can be applied to any entry in the file.
- A set of driver-specific attributes, from a set that can be applied only to entries that use the specified driver.

For example, **directorsu** specifies the attributes for a director that reads aliases from a file **/private/usr/lib/aliases**:

```
# read aliases from a file private to one machine on the network
private_aliases:
      driver=aliasfile, owner=owner-$user ;
      file=/private/usr/lib/aliases
```

This entry is named **private_aliases**, and depends upon the low-level director driver routine named **aliasfile**. Errors found while processing addresses found by this director are sent to an address formed by prefixing the string **owner-** to the name of the alias; these aliases are stored in file **/private/usr/lib/aliases**. The director-driver **aliasfile** implements a general mechanism for looking up aliases stored in a data base. By default, aliases are kept in a DBM-style data base that is built from the text file **/usr/lib/mail/aliases**. For details on this file and its format, see the Lexicon entry for **aliases**. For details on how DBM-style data bases, see the Lexicon entry for **libgdbm**.

The separation between generic attributes and driver-specific attributes mirrors the internal design of **smail**. Above the driver level, routines exist that implement aspects of drivers, routers, and transports but do not depend upon the specific means for performing the operation. These higher-level functions can be manipulated through the generic attributes. On the other hand, the drivers that actually perform these operations accept a different set of attributes to control their behavior. In the case of a driver thats read or writes to a file, a file attribute usually exists. In the case of a driver that executes a program, a **cmd** attribute usually exists to specify how that program is to be invoked.

Attributes of a Director

The following the generic attributes can be used in an entry in **directors**. Each attribute is followed by its type (Boolean or string). To set a string attribute, its name should be followed by an '=', then the value to which you are setting it. To set a Boolean attribute, prefix it with a '+'; to unset a Boolean attribute, prefix it with a '-'.

caution (Boolean)
If set, then be cautious of the addresses this director produces. If the attribute **nobody** is not set, then reject file, shell-command, or :include:filename-style mailing-list addresses.

default_group (string)
If the driver does not associate a group to an address returned by it, then associate the group identifier for this group name. This will override the group identifier set by the attribute **default_user**.

default_home (string)
If the driver does not associate a home directory with an address returned by it, then use this directory as the default home directory. **smail** expands the value of this attribute to form the directory path name. At present, variable **$user** is not available for this expansion. If the string expansion fails, **smail** ignores it.

default_user (string)
If the driver does not associate a user or group to an address returned by it, then associate the user identifier and group identifier of this user.

driver (string)
This attribute names the set of low-level functions that do the work of directing local mail. This attribute is required.

nobody (Boolean)
If set, then **smail** accesses files or runs shell commands as the user specified by its attribute **nobody**, for addresses flagged with caution by either the caution generic attribute or by the driver. Association of **nobody** with an address overrides the attributes **default_user**, **default_group**, **set_user**, and **set_group**. This attribute is set by default.

owner (string)
This names the address to be sent mail if an error occurs while **smail** is processing the addresses produced by this director. This string is expanded with the variable **$user** set to the local-form address passed to the director. By deafault, the value **owner-$user**. If this string expansion fails, **smail** ignores it.

sender_okay (Boolean)
If set, then it is always okay for this attribute to produce an address equal to the sender. This effectively turns on the "me too" flag for this director. This should generally be set for forwarding directors and should not be set for aliasing and mailing-list directors.

set_group (string)
Associate this group's identifier with the addresses that the driver returns. This overrides any group identifier set by the attribute **set_user**.

set_home (string)
Associate this home directory with all addresses returned by the driver. This will be expanded in the same manner as **default_home**.

set_user (string)
Associate the user and group identifiers for this user with addresses that the driver returns. This overrides any values set by the driver.

smail requires that two addresses exist: **Postmaster** and **Mailer-Daemon**. To avoid the necessity of an alias for these two users, **smail** contains two implicit directors embedded into the directing code; it uses them as a last resort. The first such director maps the address **Mailer-Daemon** onto the address **Postmaster**; and the second maps **Postmaster** onto the address **root**.

The Preloaded Directors

If **smail** does not find a copy of file **directors** in directory **/usr/lib/mail** (which is the case by default under COHERENT), it uses its the default configuration. The default director configuration supports the following directors:

Include Files
smail expands local addresses of the form **:***include***:***filename* into a list of addresses contained in the ASCII file *filename*. The files to which these addresses refer are called *mailing list files*. This form of local address can appear in any alias file, forward file, or mailing-list file. A user cannot supply such an address himself.

Alias Files
This director scans for entries in an DBM-style data base that is built from text file **/usr/lib/mail/aliases**. If this data base does not exist, **smail** ignores it — its absence does not trigger an error condition. If **smail** encounters an error while it is resolving an address produced by an alias, it mails an error message to an address that has the string "owner-" prefixed to the name of the alias, if such a local address is defined.

Forward Files
A user may have a file named **.forward** in his home directory. If such a file exists, **smail** scans it for addresses. If a user has such a file in his home directory, **smail** directs all mail sent to that user to the address or addresses it contains. The file can contain addresses that specify other files or shell commands

as recipients.

If the **.forward** file is owned by **root** or by the user himself, then deliveries to any shell commands or files are performed under the user's user and group identifiers. If **smail** enters an error while it is resolving this list of addresses, it mails an error message to your system's postmaster.

In the **.forward** file for the user **root**, deliveries to shell commands and file addresses are performed under an unprivileged user and group identifier (by default, user **nobody**). The same is true for forward files that were not owned by **root** or by the given user. Finally, shell command and file addresses are not allowed at all in **.forward** files that are directories that can accessed by remote systems.

Mailbox Forwarding

As an alternate way to forward mail, the mailbox file for a user may contain a line of the form:

```
Forward to address, address ...
```

Onlyone line is read from this file, so addresses cannot be placed across multiple lines. The comments that apply to a **.forward** file also apply to this use of a mailbox file, except that **smail** assumes that a mailbox is not accessible by users on other systems.

A user is matched by name, either in upper or lower case, with delivery to that user being performed using a transport by the name of **local**. A user can also be matched by name if the user name is prefixed by **real-**. Delivery is performed by a transport named **local**.

Mailing Lists

Mailing list files can be created under a mailing-list directory — by default, directory **/usr/lib/mail/lists**. To create a new mailing list, create a file in this directory that contains a list of addresses. The basename of this file determines the local address that **smail** expands into this list of addresses. For example, a file named **info-smail** could be created, that contains a list of recipient addresses for a mailing list named "info-smail". **smail** then forwards any mail message mailed to address **info-smail** to every address in file **/usr/lib/mail/lists/info-smail**.

If **smail** encounters an error as it is attempting to deliver a mail message to an address within a list file, it mails an error message to a local address comprised of the base name of the list file prefixed with the string "owner-", if such an address is defined.

The Smart User

If **smail** cannot match a local address by any other means, it can forward that mail to another system — one that presumably has a more complete data base — via the director **smartuser**.

To declare another system to be a "smart user," set the attribute **smart_user** within file **/usr/lib/mail/config**. For example, attribute forwards mail to the host **mwc.com**:

```
smart_user = $user@mwc.com
```

If you do not set this attribute, then **smail** ignores the smart-user director.

Example Entries

The order of entries within **directors** determines the order in which operations are attempted. If a director matches an address, then **smail** calls no other director to expand or resolve that address. The following gives a version of **directors** that is equivalent to the default configuration:

```
# aliasinclude - expand ":include:filename" addresses
#    produced by alias files
aliasinclude:
      driver = aliasinclude,  # use this special-case driver
      nobody;                 # associate nobody user with addresses
#  when mild permission violations
#  are encountered

      copysecure,    # get permissions from alias director
      copyowners     # get owners from alias director

# forwardinclude - expand ":include:filename" addresses
#    produced by forward files
forwardinclude:
      driver = forwardinclude, # use this special-case driver
      nobody;
```

```
      copysecure,    # get perms from forwarding director
      copyowners     # get owners from forwarding director

# aliases - search for alias expansions stored in a database
aliases:
      driver = aliasfile,      # general-purpose aliasing director
      -nobody,                 # all addresses are associated
                               # with nobody by default, so setting
                               # this is not useful.
      owner = owner-$user;     # problems go to an owner address

      file = /usr/lib/aliases,
      modemask = 002,
      optional,                # ignore if file does not exist
      proto = lsearch

# dotforward - expand .forward files in user home directories
dotforward:
      driver = forwardfile,    # general-purpose forwarding director
      owner = Postmaster,      # problems go to the user's mailbox
      nobody,
      sender_okay;             # sender never removed from expansion

      file = ~/.forward,       # .forward file in home directories
      checkowner,              # the user can own this file
      owners = root,           # or root can own the file
      modemask = 002,          # it should not be globally writable
      caution = daemon:root,   # don't run things as root or daemon
                               # be extra careful of remotely
                                   # accessible home directories
      unsecure = "~ftp:~uucp:~nuucp:/tmp:/usr/tmp"

# forwardto - expand a "Forward to " in user mailbox files
#
# This emulates the V6/V7/System-V forwarding mechanism which uses a
# line of forward addresses stored at the beginning of user mailbox
# files prefixed with the string "Forward to "
forwardto:
      driver = forwardfile,
      owner = Postmaster, nobody, sender_okay;

      file = /usr/mail/${lc:user}, # the mailbox file for System V
      forwardto,                   # enable "Forward to " function
      checkowner,                  # the user can own this file
      owners = root,               # or root can own the file
      modemask = 0002,             # under System V, group mail can write
      caution = daemon:root        # don't run things as root or daemon

      # user - match users on the local host with delivery to their mailboxes
      user:     driver = user;# driver to match usernames

      transport = local        # local transport goes to mailboxes

# real_user - match usernames when prefixed with the string "real-"
#
# This is useful for allowing an address which explicitly delivers to
# a user's mailbox file.  For example, errors in a .forward file
# expansion can be delivered here, or forwarding loops between
# multiple machines can be resolved by using a real-username address.
real_user:
      driver = user;

      transport = local,
      prefix = "real-"         # for example, match real-root
```

```
# lists - expand mailing lists stored in a list directory
#
# mailing lists can be created simply by creating a file in the
# /usr/lib/smail/lists directory.
lists:    driver = forwardfile,
      caution,                # flag all addresses with caution
      nobody,                 # and then associate the nobody user
      owner = owner-$user;    # system V sites may wish to use
# o-$user, as owner-$user may be
# too long for a 14-char filename.

      # map the name of the mailing list to lower case
      file = lists/${lc:user}

# smart_user - a partially specified smartuser director
#
# If the config file attribute smart_user is defined as a string such
# as "$user@domain-gateway" then users not matched otherwise will be
# sent off to the host "domain-gateway".
#
# If the smart_user attribute is not defined, this director is ignored.
smart_user:
      driver = smartuser;     # special-case driver

# do not match addresses which cannot be made into valid
# RFC822 local addresses without the use of double quotes.
      well_formed_only
```

See Also

Administering COHERENT, config [smail], .forward, mail [overview], routers, smail, transports

Notes

directory — Definition

A **directory** is a table that maps names to files; in other words, it associates the names of a file with their locations on the mass storage device. Under some operating systems, directories are also files, and can be handled like a file.

Directories allow files to be organized on a mass storage device in a rational manner, by function or owner.

See Also

file, Using COHERENT
POSIX Standard, §5.1.2

dirent.h — Header File

Define directory-related data elements
#include <dirent.h>

dirent.h defines the data type **DIR** and the structure **dirent**. It is used with the portable directory-manipulation routines **closedir()**, **getdents()**, **opendir()**, **readdir()**, **rewinddir()**, and **telldir()**.

See Also

closedir(), getdents(), header files, opendir(), readdir(), rewinddir(), telldir()
POSIX Standard, §5.1.1

dirname — Command

Extract a directory name
dirname *string*

The command **dirname** extracts a directory name from a file's full path name. In effect, it is the complement of the command **basename**.

If *string* contains one or more slashes '/' plus text, then **dirname** prints out the portion of *string* up to (but not including) the last slash. For example, if *string* points to **/bin/sh**, then **dirname** will return **/bin**.

If *string* does not contain a slash or is empty (that is points to the current directory), **dirname** prints a single period '.'. For example, if *string* points to **myprogram**, then **dirname** returns a period.

Finally, if *string* consists only of one slash (that is, indicates the root directory), then **dirname** returns **/**.

See Also

basename, commands, cut, paste

dirs — Command

Print the contents of the directory stack
dirs

The COHERENT shell **sh** maintains an internal "directory stack", which is a stack of names of directories. You can manipulate this stack should you, for any reason, wish to traverse a number of directories quickly and efficiently.

The command **dirs** prints the current contents of the directory stack.

See Also

commands, popd, pushd, sh

disable — Command

Disable a port
/etc/disable *port...*

disable tells the COHERENT system not to create a login process for each given asynchronous *port*. For example, the command

```
/etc/disable com1r
```

disables port **/dev/com1r**. **disable** changes the entry for each given *port* in the terminal characteristics file **/etc/ttys**, and signals **init** to rescan the **ttys** file.

The command **enable** enables a port. The command **ttystat** checks whether a port is enabled or disabled.

Files

/etc/ttys — Terminal characteristics file

See Also

asy, commands, enable, login, ttys, ttystat

Diagnostics

disable normally returns one if it disables the *port* successfully and zero if not. If more than one *port* is specified, **disable** returns the success or failure status of the last port it finds. It returns -1 if it cannot find any given *port*. An exit status of -2 indicates an error.

div() — General Function (libc)

Perform integer division
#include <stdlib.h>
div_t div(*numerator***,** *denominator***)**
int *numerator***,** *denominator***;**

div() divides *numerator* by *denominator*. It returns a structure of the type **div_t**, which is structured as follows:

```
typedef struct {
      int quot;
      int rem;
} div_t;
```

div() writes the quotient into **quot** and the remainder into **rem**.

The sign of the quotient is positive if the signs of the arguments are the same; it is negative if the signs of the arguments differ. The sign of the remainder is the same as the sign of the numerator.

If the remainder is non-zero, the magnitude of the quotient is the largest integer less than the magnitude of the algebraic quotient. This is not guaranteed by the operators **/** and **%**, which merely do what the machine implements for divide.

See Also

ldiv(), libc, stdlib.h
ANSI Standard, §7.10.6.2

Notes

The ANSI Standard includes this function to permit a useful feature found in most versions of FORTRAN, where the sign of the remainder will be the same as the sign of the numerator. Also, on most machines, division produces a remainder. This allows a quotient and remainder to be returned from one machine-divide operation.

If the result of division cannot be represented (e.g., because *denominator* is set to zero), the behavior of **div()** is undefined. *Caveat utilitor.*

do — C Keyword

Introduce a loop

do is a C control statement that introduces a loop. Unlike **for** and **while** loops, the condition in a **do** loop is evaluated *after* the operation is performed. **do** always works in tandem with **while**; for example

```
do {
      puts("Next entry? ");
      fflush(stdout);
} while(getchar() != EOF);
```

prints a prompt on the screen and waits for the user to reply. The **do** loop is convenient in this instance because the prompt must appear at least once on the screen before the user replies.

See Also

break, C keywords, continue, while
ANSI Standard, §6.6.5.2

domain — System Administration

Set your system's mail domain
/etc/domain

The file **/etc/domain** sets the domain that the COHERENT mail system uses to create your fully qualified domain name. Your fully qualified domain name is created by appending the contents of **/etc/domain** to the contents of **/etc/uucpname**, with an intervening '.'. Unless you have a registered domain name, the contents of this file should be **UUCP**.

For information on registering in the United States catch-all domain **.us**, send mail to:

```
us-domain-request@venera.isi.edu
```

UUNET Communications Services of Falls Church, Virginia, will help you set up your own domain for a modest fee. Contact **info@uunet.uu.net** for more information; or telephone them at 703-876-5050.

See Also

Administering COHERENT, mail, paths, uucpname

dos — Command

Manipulate files on MS-DOS file systems
dos [-]dDfFlrtx[*flags***] [***device***] [***file ...***]**

The command **dos** allows the COHERENT user to manipulate an MS-DOS file system, which may be either a hard-disk partition or a floppy disk. It can build an empty MS-DOS file system, label it, list the files in it, transfer files between it and COHERENT, or delete files from it.

The given *device* must be a special file that specifies an MS-DOS file system, such as floppy-disk drive **/dev/fha0** or hard-disk partition **/dev/at0a**. The default *device* is **/dev/dos**, which the system administrator should link to the most commonly used device name.

dos converts between the differing file-name conventions of COHERENT and MS-DOS. An MS-DOS *file* argument may be specified in lower or upper case, using '/' as the path-name separator. When transferring files from MS-DOS to COHERENT, **dos** converts an MS-DOS file name to a COHERENT file name in lower case only. If the MS-DOS file name contains no extension, the COHERENT file name contains no '.'. When transferring files from COHERENT to MS-DOS, **dos** converts all alphabetic characters in a COHERENT file name to upper case; if a period '.' appears at the beginning or end of a file name, **dos** converts it to '_'. **dos** truncates the part of the file name before the last '.' to a maximum of eight characters and truncates the extension to a maximum of three characters.

The command line must specify exactly one of the following *functions*.

d Delete each *file* from the MS-DOS file system. This option also allows the user to delete empty directories.

F Create an empty MS-DOS file system on a formatted diskette. This option is analogous to the COHERENT command **/etc/mkfs**. The COHERENT commands **/etc/fdformat** and **/etc/mkfs** initialize a COHERENT diskette in two steps. The MS-DOS command **format** initializes an MS-DOS diskette by performing both the physical and logical formatting operations with one command. To initialize an MS-DOS diskette under COHERENT, use the command **/etc/fdformat -v** *devicename*, followed by the command **dos F** *devicename*. If *file* is named, **dos** copies it to the boot block of the file system. The **dos** command cannot build a file system on a hard-disk partition.

f Force removal of **readonly** files on the MS-DOS side.

l Label the MS-DOS file system. The command line must specify exactly one *file* argument, which gives the label.

r Replace each *file* on the MS-DOS file system with the COHERENT file of the same name. If a given *file* argument specifies a COHERENT directory, **dos** replaces its subdirectories recursively to the MS-DOS file system unless the **s** flag is used. If no *file* is specified, **dos** copies all files in the current directory to the MS-DOS file system.

t List the files on the MS-DOS file system. If no *file* argument is given, **dos** lists the entire MS-DOS file system; otherwise, it lists each *file*. If a *file* argument specifies an MS-DOS subdirectory, **dos** lists its contents. **dos** lists directories first in alphabetical order, then ordinary files in alphabetical order.

x Extract each *file* from the MS-DOS file system to a COHERENT file of the same name. If a given *file* argument specifies an MS-DOS subdirectory, **dos** extracts its contents recursively unless the **s** flag is used. If no *file* is given, **dos** extracts all files from the MS-DOS file system to the current COHERENT directory.

The following *flags* are available.

a Perform ASCII newline conversion on file transfer. When moving files from COHERENT to MS-DOS, this option converts each COHERENT newline character '\n' (ASCII **LF**) to an MS-DOS end-of-line (ASCII **CR** and **LF**); when moving files from MS-DOS to COHERENT, it does the opposite. By default, **dos** performs binary file transfer, without newline conversion.

k Keep the file modification time (mtime) on extract and replace operations. By default, **dos** gives extracted or replaced files the current time. With this option, **dos** gives the extracted or replaced file the same time as the original file.

n List files in order of creation (newest file last) rather than in alphabetical order. This option applies only to the table-of-contents function. **dos** always lists directories before files, with or without the **n** option.

p Perform a piped extract or replace (for use in pipelines). The command line must specify exactly one *file* argument. For extract, **dos** reads the given *file* and writes it to the standard output. For replace, **dos** reads the standard input and writes it to the given *file*.

s Suppress extraction or replacement of subdirectories. By default, **dos** extracts or replaces subdirectories recursively.

v Verbose option. Provide additional information about each function performed.

[**1-9**] A digit specifies a logical drive number on an extended MS-DOS partition. For example, **dos tv2 /dev/at0c** lists the directory of the second logical drive on extended MS-DOS partition **/dev/at0c**.

dos Commands

dos is an obsolete command. It has largely been superceded by the following family of COHERENT commands that manipulate MS-DOS file systems:

doscat Concatenate a file on an MS-DOS file system.
doscp Copy files to/from an MS-DOS file system
doscpdir Copy a directory to/from an MS-DOS file system
dosdel Delete a file from an MS-DOS file system
dosdir List contents of an MS-DOS directory
dosformat Build an MS-DOS file system on a floppy disk
doslabel Label an MS-DOS floppy disk
dosls List files on an MS-DOS file system
dosmkdir Create a directory in an MS-DOS file system
dosrm Remove a file from an MS-DOS file system
dosrmdir Remove a directory from an MS-DOS file system

For details, see these commands' entries within the Lexicon.

Examples

The first example copies all files located in directories **sources** and **include**, as well as any subdirectories, from floppy drive **/dev/fva1** to correspondingly named subdirectories in the current COHERENT directory:

```
dos xavk /dev/fva1 sources include
```

Note that **fva1** is a high-density, 3.5-inch floppy disk in floppy-disk drive 1 (a.k.a., drive B:). The files will be copied with ASCII newline conversion and will retain the time and date that they had under MS-DOS.

The next example copies a file from an MS-DOS partition on your hard disk. Suppose that **C:** is the primary MS-DOS partition on your first hard drive. The following command copies file **C:\AUTOEXEC.BAT** to **/autoexec.bat** in your COHERENT root partition:

```
dos xa /dev/at0a /autoexec.bat
```

You will want to use the **a** switch any time you are transferring a text file.

Suppose that the second partition on your first hard drive (COHERENT device **/dev/at0b**) is an extended MS-DOS partition with two logical drives, **D:** and **E:**. To copy a COHERENT text file **/tmp/foo** to **D:\TMP\FOO**, use the command

```
dos ra1 /dev/at0b /tmp/foo
```

To copy non-text file **frotz** in the current COHERENT directory to MS-DOS file **E:\DBF\AX\FROTZ**, use the command

```
dos rp2 /dev/at0b dbf/ax/frotz < frotz
```

See Also

commands, fdformat, mkfs, MS-DOS

Notes

dos is an obsolete command. It has been retained for compatibility with earlier versions of COHERENT. We urge you to use the other members in the **dos** family of commands, which have a cleaner syntax and are much easier to use.

dos does not check for unusual characters in a COHERENT file name or for file names that differ from other file names only in case.

The **dos** family of commands now support large file systems, such as those created by MS-DOS versions 4.0 and 5.0.

The COHERENT system's **dos** family of commands do not understand compressed MS-DOS file systems created by programs such as **Stacker** or MS-DOS 6.0 **dblspace**. If you are running MS-DOS with file compression, you must copy files to an uncompressed file system (for example, to an uncompressed floppy disk or to the uncompressed host for a compressed file system) to make them accessible to the COHERENT **dos** commands.

doscat — Command

Concatenate a file on an MS-DOS file system
doscat *device*:**[/***directory***/]***file*

doscat concatenates *file* that is in *directory* on an MS-DOS file system. *device* names the floppy-disk or hard-disk device that holds the file system to be modified, e.g., **/dev/fha0**. You can also build a file of aliases so that you can access the drives as **a:**, **b:**, etc. For details, see the Lexicon entry for **doscp**, which explains how to set up defaults for the **dos** family of commands.

file can name either a single file, or can contain a wildcard character to name more than one file. For example, the command

```
doscat a:foo.c
```

concatenates file **foo.c** which is on the file system contained in device whose alias is **a:** (as defined in file **/etc/default/msdos**). Likewise, the command

```
doscat 'c:/dirname/*.txt'
```

concatenates all files with the suffix **.txt** in directory **dirname**, which, in turn, is on the file system contained in device whose alias is **c:** (as defined in file **/etc/default/msdos**). In this form of the command, **doscat** concatenates the files in the alphabetical order of their names. Note that the tail of the command must be enclosed within apostrophes, or the shell will expand the '*' before it is read by **doscat**.

Files

/etc/default/msdos — Setup file

See Also

cat, commands, dos

Notes

doscat does not understand compressed MS-DOS file systems created by programs such as **Stacker** or MS-DOS 6.0 **dblspace**. If you are running MS-DOS with file compression, you must copy files to an uncompressed file system (for example, to an uncompressed floppy disk or to the uncompressed host for a compressed file system) to make them accessible to the **doscat**.

doscp — Command

Copy files to/from an MS-DOS file system
doscp [-abkmrv] *src dest*

doscp copies files between MS-DOS and COHERENT file systems. The MS-DOS file system can reside either on a floppy disk, or on an MS-DOS partition of a hard disk.

src names the file being copied and the file system where it resides; *dest* names the file system and directory into which the file is copied. The operating system that owns the *src* file is implied by the name of the file system on which it resides. An MS-DOS file system must be named using the device that holds it, such as floppy-disk drive **/dev/fha0** or hard-disk partition **/dev/at0a**. You can also build a file of aliases so that you can access the drives as **a**, **b**, etc. For details, see the section entitled *Configuring the dos Commands*, below.

doscp converts a file's name from one operating system's conventions to the other's. An MS-DOS file argument may be specified in lower or upper case, using '/' as the path-name separator. When transferring files from MS-DOS to COHERENT, **doscp** converts an MS-DOS file name to a COHERENT file name in lower case only. If the MS-DOS file name contains no extension, the COHERENT file name contains no '.'. When transferring files from COHERENT to MS-DOS, **doscp** converts all alphabetic characters in a COHERENT file name to upper case; if a period '.' appears at the beginning or end of a file name, **doscp** converts it to '_'. **doscp** truncates the portion of the file name to the left of the '.' to a maximum of eight characters and portion to the right of the '.' to a maximum of three characters.

doscp recognizes the following options:

a Perform ASCII newline conversion on file transfer. When moving files from COHERENT to MS-DOS, this option converts each COHERENT newline character '\n' (ASCII LF) to an MS-DOS end-of-line (ASCII CR and LF). When moving files from MS-DOS to COHERENT, it does the opposite. By default, **doscp** performs ASCII conversion on files that have an ASCII extention. See **Setup**, below.

b Do not perform any newline conversion on file transfers.

k Keep: give the copied file the same time stamp as its original. By default, **doscp** gives copied files the current time.

m Same as **a**, described above

r Same as **b**, described above.

v Verbose. Provide additional information about each action performed.

Configuring the dos Commands

The **dos** family commands read the file **/etc/default/msdos** before they begin to interpret arguments. By modifying this file, you can establish defaults that let COHERENT's **dos** commands resemble their counterparts under MS-DOS. You can set up two classes of defaults: *device* defaults and *file* defaults.

A device default lets you declare an alias for a device that holds an MS-DOS file system. This device can be a floppy-disk drive, a partition on a hard disk, or an extended partition on a hard disk. The alias must consist of one or two letters. No letter can serve as an alias for more than one device. For example, the following declaration:

```
c=/dev/at0a
```

specifies that the hard-disk partition accessed via device **/dev/at0a** is a "Primary MS-DOS" partition, and that its alias is **c**. Hereafter, the **dos** commands will interpret **c** as being equivalent to **/dev/at0a**.

The declaration

```
d=/dev/at0b;1
```

specifies the first "Extended MS-DOS" partition on the partition accessed via device **/dev/at0b**. Bumping the number from 1 to 2 would specify the second extended MS-DOS partition within partition **/dev/at0b**, as in:

```
e=/dev/at0b;2
```

Notice how the device names (c, d, and e) can correspond to the same drive names as under MS-DOS, whether or not they are primary or extended partitions.

File declarations, on the other hand, simply declare that all files with a given suffix are text files and should always have their newline characters converted from COHERENT to MS-DOS format (or vice versa). For example, placing the line

```
.c
```

in **/etc/default/msdos** tells all of the **dos** commands that all files with the suffix **.c** are text files and should have their newline characters converted by default. You can have any number of file defaults in **/etc/default/msdos**.

Examples

The first example copies all C source files from floppy drive **/dev/fva1** to correspondingly named files in the current COHERENT directory, preserves the time stamp, and performs newline conversion upon them:

```
doscp -akv /dev/fva1:source/\*.c .
```

Note that you must quote wildcard characters with a backslash to keep the shell from interpreting them. Also note that **/dev/fva1** is a high-density, 3.5-inch floppy disk in floppy-disk drive 1. So, if your **default** file contained the entry

```
b=/dev/fva1
.c
```

you could also have typed:

```
doscp -kv b:source/\*.c .
```

The next example copies a file from an MS-DOS partition on your hard disk to a COHERENT file system. Suppose that C is the primary MS-DOS partition on your first hard drive. The following command copies file **C:\AUTOEXEC.BAT** to **/tmp/autoexec.bat** in your COHERENT partition:

```
doscp /dev/at0a:autoexec.bat /tmp
```

If your **/etc/default** file contains the entry

```
c=/dev/at0a
```

then you can also type:

```
doscp c:autoexec.bat /tmp
```

LEXICON

Files

/etc/default/msdos — Setup file

See Also

commands, cp, dos

Notes

For a discussion of the error message

```
Probably not a DOS disk
```

see the notes to the Lexicon entry for **doscp**. **doscp** does not check for unusual characters in a COHERENT file name or for file names that differ from other file names only in case.

Beware of using **doscp** to create impossible files, e.g., **com1**. Such files create serious problems; for example, if you try to **TYPE** or otherwise perform MS-DOS operations on **com1**, you will attack the MS-DOS device driver instead of the file. Be sure to rename all such files when you copy them from a COHERENT to an MS-DOS file system.

doscp does not understand compressed MS-DOS file systems created by programs such as **Stacker** or MS-DOS 6.0 **dblspace**. If you are running MS-DOS with file compression, you must copy files to an uncompressed file system (for example, to an uncompressed floppy disk or to the uncompressed host for a compressed file system) to make them accessible to the **doscp**.

doscpdir — Command

Copy a directory to/from an MS-DOS file system
doscpdir [-akmv] *src dest*

doscpdir copies a directory and its contents between an MS-DOS file system and a COHERENT file system. The MS-DOS file system can reside either on a floppy disk, or on the MS-DOS segment of a hard disk on your system.

src names the directory being copied and the file system where it resides; *dest* names the file system and directory into which the file is copied. The operating system that owns the *src* file is implied by the name of the file system on which it resides. An MS-DOS file system must be named using the device that holds it, such as floppy-disk drive **/dev/fha0** or hard-disk partition **/dev/at0a**. You can also build a file of aliases so that you can access the drives as **a**, **b**, etc. For details, see the Lexicon entry for **doscp**, which explains how to set up defaults for the **dos** family of commands.

doscpdir converts a file's name from one operating system's conventions to the other's. An MS-DOS file argument may be specified in lower or upper case, using '/' as the path-name separator. When transferring files from MS-DOS to COHERENT, **doscpdir** converts an MS-DOS file name to a COHERENT file name in lower case only. If the MS-DOS file name contains no extension, the COHERENT file name contains no '.'. When transferring files from COHERENT to MS-DOS, **doscpdir** converts all alphabetic characters in a COHERENT file name to upper case; if a period '.' appears at the beginning or end of a file name, **doscpdir** converts it to '_'. **doscpdir** truncates the part of the file name before the last '.' to a maximum of eight characters and truncates the extension to a maximum of three characters.

doscpdir recognizes the following options:

a Perform ASCII newline conversion on file transfer. When moving files from COHERENT to MS-DOS, this option converts each COHERENT newline character '\n' (ASCII LF) to an MS-DOS end-of-line (ASCII CR and LF). When moving files from MS-DOS to COHERENT, it does the opposite. By default, **doscpdir** performs ASCII conversion on files that have an ASCII extention.

k Keep: give the copied file the same time stamp as its original. By default, **doscpdir** gives copied files the current time.

m Same as **a**, described above

v Verbose. Provide additional information about each action performed.

Example

The following command copies COHERENT directory **/usr/src** to directory **/mydir** on the MS-DOS file system. It assumes that you have set **c** as a default for a hard-disk device:

```
doscpdir -va /usr/src c:/mydir
```

Files

/etc/default/msdos — Setup file

See Also

commands, cpdir, dos

Notes

doscpdir does not check for unusual characters in a COHERENT file name or for file names that differ from other file names only in case.

doscpdir does not understand compressed MS-DOS file systems created by programs such as **Stacker** or MS-DOS 6.0 **dblspace**. If you are running MS-DOS with file compression, you must copy files to an uncompressed file system (for example, to an uncompressed floppy disk or to the uncompressed host for a compressed file system) to make them accessible to the **doscpdir**.

dosdel — Command

Delete a file from an MS-DOS file system
dosdel [-fv] *device***:/***dir***/***file*

dosdel deletes *file* that lives on MS-DOS file-system *device*. The MS-DOS file system can reside either on a floppy disk, or on the MS-DOS segment of a hard disk on your system. The MS-DOS file system must be named using the device that holds it, such as floppy-disk drive **/dev/fha0** or hard-disk partition **/dev/at0a**. You can also build a file of aliases so that you can access the drives as **a**, **b**, etc. For details, see the Lexicon entry for **doscp**, which explains how to set up defaults for the **dos** family of commands.

dosdel takes the following options:

f Force removal of **readonly** files.

v Verbose output: provide additional information about each action.

Example

The following command deletes **myfile**. It assumes that you have defined **c** to be a default for a device upon which you have set an MS-DOS file system:

```
dosdel c:/mydir/myfile
```

Files

/etc/default/msdos — Setup file

See Also

commands, dos

Notes

dosdel does not understand compressed MS-DOS file systems created by programs such as **Stacker** or MS-DOS 6.0 **dblspace**. If you are running MS-DOS with file compression, you must copy files to an uncompressed file system (for example, to an uncompressed floppy disk or to the uncompressed host for a compressed file system) to make them accessible to the **dosdel**.

dosdir — Command

List contents of an MS-DOS directory
dosdir [-nv] *device***:[***dir***/][***file***]**

dosdir lists the contents of a *directory* that lives on an MS-DOS file system. The MS-DOS file system can reside either on a floppy disk, or on the MS-DOS segment of a hard disk on your system. The MS-DOS file system must be named using the device that holds it, such as floppy-disk drive **/dev/fha0** or hard-disk partition **/dev/at0a**. You can also build a file of aliases so that you can access the drives as **a**, **b**, etc. For details, see the Lexicon entry for **doscp**, which explains how to set up defaults for the **dos** family of commands.

dosdir recognizes the following options:

n Newest: List the files in the order in which they were last modified, from newest to oldest. By default, **dosdir** lists files in alphabetical order.

v Verbose. Provide additional information about each action performed.

Example

The following command lists the contents of **mydir**. It assumes that you have defined **c** as a default for a device on which is set an MS-DOS file system:

```
dosdir c:/mydir
```

Files

/etc/default/msdos — Setup file

See Also

commands, dos, dosls, ls

Notes

If you see the error

```
dosdir: Probably not a DOS disk (media descriptor 0x00)
```

dosdir cannot find a valid boot block on a partition. It happens when you try to access an extended DOS partition as though it were a primary partition. Check the line in **/etc/default/msdos** to see how **dosdir** is accessing that partition.

For example, if are trying to access device **h:** and the entry for that device reads

```
h=/dev/sd1a
```

this device may in fact be an extended partition. It sometimes happens with removable media, such as removable SCSI disks, have extended partitions built on them without the operator's knowledge. To test whether this partition is in fact an extended partition, type the command:

```
dosdir -v /dev/sd1a;1
```

If you then see the contents of the partition, you know that you are on the right track. Change the entry for device **h** to read

```
h=/dev/sd1a;1
```

and all should be well.

dosdir does not understand compressed MS-DOS file systems created by programs such as **Stacker** or MS-DOS 6.0 **dblspace**. If you are running MS-DOS with file compression, you must copy files to an uncompressed file system (for example, to an uncompressed floppy disk or to the uncompressed host for a compressed file system) to make them accessible to the **dosdir**.

dosformat — Command

Build an MS-DOS file system
dosformat [-v] *device***:**

dosformat builds an MS-DOS file system on a floppy disk. The disk must first have been formatted with the command **fdformat -v**. *device* names the floppy-disk drive that holds the disk to receive the file system, such as **/dev/fha0**. See the Lexicon entry **floppy disks** for a table of the COHERENT floppy-disk devices. You can also build a file of aliases so that you can access the drives as **A**, **B**, etc. For details, see the Lexicon entry for **doscp**, which explains how to set up defaults for the **dos** family of commands. Note that the device name must always be suffixed with a colon ':', just like an MS-DOS device name.

The option **-v**, tells **dosformat** to provide additional information about each action it performs.

Example

The following example formats a disk. It assumes that you have defined **a** as a default for a device upon which is set an MS-DOS file system:

```
dosformat a:
```

Files

/etc/default/msdos — Setup file

See Also

commands, dos, fdformat

Notes

To create a double-sided, double-density formatted floppy disk in drive 0 (drive A), use **/dev/fqa0** for 3.5-inch disks, or **/dev/f9a0** for 5.25-inch disks.

doslabel — Command

Label an MS-DOS floppy disk
doslabel [-v] *device***:** *label*

doslabel puts *label* onto an MS-DOS floppy disk. *device* names the floppy-disk drive that holds the disk to be labelled, such as **/dev/fha0**. See the Lexicon entry **floppy disks** for a table of the COHERENT floppy-disk devices. You can also build a file of aliases so that you can access the drives as **a**, **b**, etc. For details, see the Lexicon entry for **doscp**, which explains how to set up defaults for the **dos** family of commands.

The option **-v**, tells **doslabel** to provide additional information about each action it performs.

Example

The following command labels an MS-DOS floppy disk with the string **mydisk**. It assumes that you have defined **a** as a default for a device that holds an MS-DOS file system:

```
doslabel a: mydisk
```

Files

/etc/default/msdos — Setup file

See Also

commands, dos

dosls — Command

List files on an MS-DOS file system
dosls [-v] *device***:[/**_directory_**/][**_file_**]**

dosls lists all files in *directory* on an MS-DOS file system. *device* names the floppy-disk or hard-disk device that holds the file system to be modified, e.g., **/dev/fha0**. You can also build a file of aliases so that you can access the drives as **a**, **b**, etc. For details, see the Lexicon entry for **doscp**, which explains how to set up defaults for the **dos** family of commands.

The option **-v** tells **dosls** to print its output in a long format, analogous to what the command **ls -l** prints.

Example

The following displays the contents of directory **src**. It assumes that you have defined **c** as a default for a device on which you have set an MS-DOS file system:

```
dosls -v c:/src
```

Files

/etc/default/msdos — Setup file

See Also

commands, dos, dosdir, ls

Notes

dosls does not understand compressed MS-DOS file systems created by programs such as **Stacker** or MS-DOS 6.0 **dblspace**.

dosmkdir — Command

Create a directory in an MS-DOS file system
dosmkdir *device***:***directory*

dosmkdir makes *directory* in an MS-DOS file system. *device* names the floppy-disk or hard-disk device that holds the file system to be modified, e.g., **/dev/fha0**. You can also build a file of aliases so that you can access the drives as **a**, **b**, etc. For details, see the Lexicon entry for **doscp**, which explains how to set up defaults for the **dos** family

of commands.

Example

The following command creates directory **mydir**. It assumes that you have defined **a** to be a device in which is set an MS-DOS file system:

```
dosmkdir a:/mydir
```

Files

/etc/default/msdos — Setup file

See Also

commands, dos, mkdir

Notes

dosmkdir does not understand compressed MS-DOS file systems created by programs such as **Stacker** or MS-DOS 6.0 **dblspace**.

dosrm — Command

Remove a file from an MS-DOS file system
dosrm *device*:**[/***directory*/**]***file*

dosrm removes *file* from *directory* on an MS-DOS file system. *device* names the floppy-disk or hard-disk device that holds the file system to be modified, e.g., **/dev/fha0**. You can also build a file of aliases so that you can access the drives as **a**, **b**, etc. For details, see the Lexicon entry for **doscp**, which explains how to set up defaults for the **dos** family of commands.

Example

The following deletes all **.c** files on an MS-DOS disk. It assumes that you have defined **b** to be a device on which you have set an MS-DOS file system:

```
dosrm 'b:*.c'
```

Files

/etc/default/msdos — Setup file

See Also

commands, dos, dosrmdir, rm

Notes

dosrm does not understand compressed MS-DOS file systems created by programs such as **Stacker** or MS-DOS 6.0 **dblspace**.

dosrmdir — Command

Remove a directory from an MS-DOS file system
dosrmdir *device*:*directory*

dosrmdir removes *directory* from an MS-DOS file system. *device* names the floppy-disk or hard-disk device that holds the file system to be modified, e.g., **/dev/fha0**. You can also build a file of aliases so that you can access the drives as **a**, **b**, etc. For details, see the Lexicon entry for **doscp**, which explains how to set up defaults for the **dos** family of commands.

Example

The following command removes directory **foo**. It assumes that you have defined **a** to be a device in which you have set a disk with an MS-DOS file system:

```
dosrmdir c:/foo
```

Files

/etc/default/msdos — Setup file

See Also

commands, dos, dosrm, rmdir

Notes

dosrmdir does not understand compressed MS-DOS file systems created by programs such as **Stacker** or MS-DOS 6.0 **dblspace**.

double — C Keyword

Data type

A **double** is the data type that encodes a double-precision floating-point number. On most machines, **sizeof(double)** is defined as four machine words, or eight **char**s. If you wish your code to be portable, do *not* use routines that depend on a **double** being 64 bits long. The ranges of values that can be held by a COHERENT **double** are set in header file **float.h**.

Different formats are used to encode **double**s on various machines. These formats include IEEE, DECVAX, and BCD (binary coded decimal), as described in the entry for **float**. COHERENT 286 uses DECVAX format; COHERENT 386 uses IEEE format.

See Also

C keywords, data formats, float, float.h, portability
ANSI Standard, §6.1.2.5

dpac — Command

De-fragment a COHERENT file system
dpac [-q] *raw_device*

Command **dpac** de-fragments the COHERENT file system on *raw_device*. Defragmentation leaves each file in the file system physically contiguous. This reduces the number of seeks needed to access a file, and threfore permits disk I/O to run at its maximum speed. The default algorithm also sorts the i-nodes by modification date and puts the oldest ones at the beginning of the partition. This helps the file system remain un-fragmented longer.

You must **umount** the target file system *raw_device* before you run **dpac** on it. Failure to do so will corrupt the file system. For example, the command

```
dpac /dev/rat0a
```

tells **dpac** to map the first partition on the first drive and prompt whether to continue. *raw_device* must be a partition or a floppy disk rather than an entire hard drive.

dpac begins by making a map of the file system. It displays a histogram of its activity as it builds the map; this lets you see what the kernel must do in order to access each file. When it has finished the file system map, **dpac** prompts you and asks whether to quit, continue with defragmentatation using the default date sort, or to continue but to use an unsorted method of defragmentation. **dpac** does not use terminfo or termcap for its display, and is intended for use on the console's **ansipc** terminal setting. This lets you run it from a bootable floppy disk.

See Also

commands, fmap, fsck, qpac, spac, upac

Notes

To see how fragmented a file system is, use the command **fmap**.

Note that you can also de-fragment a file system by copying it to a tape, then deleting it and restoring it from the tape. Another method of defragmentation is to use the command **cpdir** to copy the file system to a spare partition (should you have one that is large enough), then using the spare partition in place of the old partition.

Please note that if you use **dpac** incorrectly or without sufficient amounts of RAM or spare disk space, you can damage or destroy your file system. *Never* run **dpac** on the partition-table device (e.g., /dev/at0x), or on the root device. *Caveat utilitor!*

dpac was written by Randy Wright (rw@rwsys.wimsey.bc.ca).

drand48() — Random-Number Function (libc)

Return a 48-bit pseudo-random number as a double
double drand48()

Function **drand48()** generates and returns a 48-bit pseudo-random number in the form of a **double**.

See Also

libc, srand48()

drvld.all — System Administration

Load loadable drivers at boot time
/etc/drvld.all

The file **/etc/drvld.all** holds commands to load loadable drivers into memory when you boot the COHERENT system. It is read from the script **/etc/brc**, which is executed whenever the COHERENT system is rebooted into single-user mode.

Under COHERENT 286, **drvld.all** (as its name implies) includes calls to the command **drvld** to load loadable drivers. COHERENT 386 does not implement loadable device drivers; however, it uses **drvld.all** to load the keyboard table and perform other useful work.

See Also

Administering COHERENT, brc, keyboard

du — Command

Summarize disk usage
du [-a] [-s] [*directory ...***]**

du prints the total number of disk blocks used by each named *directory*. If no *directory* is specified, **du** prints the disk usage of the current directory.

The **-a** (all) option causes **du** to print a line for every file and directory in the substructure. Normally it prints a line only for each directory.

The **-s** (summary) option prints only the line for the top level directory.

du understands links; it adds a file with more than one link to it into the total only once.

See Also

commands, df, find

Notes

du does not count file-system overhead such as indirect blocks, so occasionally a directory does not fit on a file system which appears to contain enough room for it.

dump — Command

File-system backup utility
dump [*options***] [***argument ...***]**

dump dumps either all or a portion of file system *argument* to magnetic tape or floppy disks. File-system dumps are in a format that permits you to restore all or some of the files to the original file system, and to select files either by name or by i-number.

A file-system dump includes all files changed since the *dump since* date, plus each file's full path name (for the benefit of **dumpdir**).

options specifies both the dump-since date and the processing options. It is made up of characters from the set **0123456789bdfsSuv**, which have the following meanings.

0-9 The digit gives the level number of the dump. The dump-since date is the most recent date in the dump-date file **/etc/ddate** that is (1) associated with this file system and (2) has a level number less than the current dump level. For example, if you request a level-3 dump, **dump** will back up all files not backed up since the last level-2 dump. A level-0 dump by definition backs up all files in the file system.

b The next *argument* gives the output tape's *blocking factor*. The blocking factor is the number of **dumpdata** structures in each tape block. The default blocking factor is 20.

d The next *argument* gives the density of the output tape in bytes per inch. The default density is 1600 bytes per inch (bpi). **dump** uses the density to compute the quantity of tape needed.

f The next *argument* gives the path name of the output file. If no **f** option is given, **/dev/dump** is assumed.

s The next *argument* gives the length of the dump tape in feet. **dump** keeps a running total of the quantity of tape it has written, and it asks for a new reel if it appears that the end of the reel is near. The default length is 2,300 feet.

S The next *argument* gives the size of the dump output device, in blocks. This is used only if you are backing up the file system to floppy disks or streaming cartridge tape rather than to nine-track magnetic tape.

u If the dump completes without error, update the record of successful dumps kept in file **/etc/ddate**. There is an entry in this file for every file system and every dump level.

v Inform the user of the 'dump since' date and the length of tape used in feet. The length is useful for computing the quantity of tape remaining if multiple dumps are written onto a single reel of tape.

If no level number is given, **dump** assumes the *options* **9u**.

Files

/dev/dump — Default dump device
/etc/ddate — Dump date file

See Also

badscan, commands, dumpdate, dumpdir, restor

Diagnostics

Most errors are fatal caused by a table overflowing, or a read or write error on the input or output device.

dump requires that its output be written to disks that are free of bad sectors. If you write a dump to a disk with bad sectors, you will not be able to restore files from that disk.

When formatting disks to be used with **dump**, use the command

```
/etc/fdformat -v device
```

This forces **fdformat** to verify the format. It takes twice as long, but it ensures that the disk is good at least at a first level of testing. Reject any disks that have any defects — or save them for use with COHERENT file systems, which can map out bad sectors.

Notes

Please note that **dump** is now regarded as being obsolete. We strongly encourage users to use **cpio** instead.

dumpdate — Command

Print dump dates
dumpdate [*filesystem ...***]**

dumpdate reads through the dump date file **/etc/ddate** and displays the dump date records associated with each specified *filesystem.*

If no *filesystem* is specified, the records for all file systems are displayed.

Files

/etc/ddate — Dump date file

See Also

commands, dump, dumpdir, restor

dumpdir — Command

Print the directory of a dump
dumpdir [af [*argument ...***]]**

dumpdir reads through a file-system dump created by the **dump** command, gathers up its directory blocks, and displays the names and i-numbers of all files on the dump.

The **a** option causes **dumpdir** to display the directory entries for '.' and '..', which are normally suppressed.

The **f** option causes the next *argument* to be taken as the pathname of the dump device, which is otherwise assumed to be **/dev/dump**.

If no options are specified, **dumpdir** reads from the default dump device **/dev/dump** and suppresses the printing of '.' and '..' entries.

Files

/dev/dump — Default dump device
/tmp/dd*XXXXXX* — To hold directory blocks

See Also

commands, dump

Diagnostics

The dump/restore format puts a header at the beginning of the dump that includes all the information about what lives where in the dump. **dumpdir** reads this header to discover what files are in the dump. If the header is too large to fit onto one disk, **dumpdir** will then prompt you to insert the additional disk or disks; if this happens, insert the requested disk and then type **<return>**.

Notes

dump requires that its output be written to disks that are free of bad sectors. If you write a dump to a disk with bad sectors, you will not be able to restore files from that disk. For details on using disks with **dump**, see its Lexicon entry.

dumptape.h — Header File

Define data structures used on dump tapes
#include <dumptape.h>

dumptape.h defines the data structures used on archives dumped with the command **dump**. Note that the command **dump** is regarded as obsolete. In its place, you should use **pax**, **tar**, or **cpio**.

See Also

header files

Notes

This header file is obsolete, and will be dropped from a future release of COHERENT. Its use is strongly discouraged.

dup() — System Call (libc)

Duplicate a file descriptor
#include <unistd.h>
int dup(*fd***) int** *fd***;**

dup() duplicates the existing file descriptor *fd*, and returns the new descriptor. The returned value is the smallest file descriptor that is not already in use by the calling process.

See Also

dup2(), fopen(), fdopen(), libc, stdio.h, unistd.h
POSIX Standard, §6.2.1

Diagnostics

dup() returns a number less than zero when an error occurs, such as a bad file descriptor or no file descriptor available.

dup2() — General Function (libc)

Duplicate a file descriptor
#include <unistd.h>
int dup2(*fd***,** *newfd***) int** *fd***,** *newfd***;**

dup2() duplicates the file descriptor *fd*. Unlike its cousin **dup()**, **dup2()** allows you to specify a new file descriptor *newfd*, rather than having the system select one. If *newfd* is already open, the system closes it before assigning it to the new file. **dup2()** returns the duplicate descriptor.

See Also

dup(), libc, stdio.h, unistd.h
POSIX Standard, §6.2.1

Diagnostics

dup2() returns a number less than zero when an error occurs, such as a bad file descriptor or no file descriptor available.

Example

echo — Command

Repeat/expand an argument
echo [-n] [*argument ...***]**

echo prints each *argument* on the standard output, placing a space between each *argument*. It appends a newline to the end of the output unless the **-n** flag is present.

echo recognizes the following special character sequences. For each occurrence of the sequence, it substitutes the corresponding ASCII character.

\b	Backspace
\c	Print line without a newline (like **-n** option)
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Tab
\v	Vertical tab
\\	Backslash
\0*nnn*	*nnn* is octal value of character (**sh** only)
nnn	*nnn* is the octal value of character (**ksh** only)

For example, when you enter the command:

```
echo 'Please enter your name: \007\c'
```

The shell rings the bell and prints

```
Please enter your name:
```

on your screen. Note that the **\007** sequence causes the terminal bell to sound, and that since the **\c** sequence was specified, the cursor will be left positioned after the colon.

See Also

commands, ksh, sh

Notes

Under the Korn shell, **echo** is an alias for its built-in command **print**.

Please note that **echo** converts characters to spaces. If you wish to preseve tab characters in an echoed string, you must enclose it within quotation marks. For example, the command

```
echo $RECORD
```

displays:

```
7 5 175 875
```

whereas the command

```
echo "$RECORD"
```

displays:

```
7     5     175   875
```

This is important when you use **echo** with programs for which the tab character is significant.

ecvt() — General Function (libc)

Convert floating-point numbers to strings
char *
ecvt(*d***,** *prec***,** *dp***,** *signp***)**
double *d***; int** *prec***, ****dp***, *signp;**

ecvt() converts *d* into a NUL-terminated string of numerals with the precision of *prec*. Its operation resembles that of **printf()**'s operator **%e**.

ecvt() rounds the last digit and returns a pointer to the result. On return, **ecvt()** sets *dp* to point to an integer that indicates the location of the decimal point relative to the beginning of the string, to the right if positive, to the left if negative. It sets *signp* to point to an integer that indicates the sign of *d*, zero if positive and nonzero if negative.

Example

The following program demonstrates **ecvt()**, **fcvt()**, and **gcvt()**.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* prototypes for extended functions */
extern char *ecvt();
extern char *fcvt();
extern char *gcvt();

main(void)
{
      char buf[64];
      double d;
      int i, j;
      char *s;

      d = 1234.56789;
      s = ecvt(d, 5, &i, &j);
      /* prints ecvt="12346" i=4 j=0 */
      printf("ecvt=\"%s\" i=%d j=%d\n", s, i, j);

      strcpy(s, fcvt(d, 5, &i, &j));
      /* prints fcvt="123456789" i=4 j=0 */
      printf("fcvt=\"%s\" i=%d j=%d\n", s, i, j);

      s = gcvt(d, 5, buf);
      /* prints gcvt="1234.56789" */
      printf("gcvt=\"%s\"\n", s);
}
```

See Also

libc

Notes

ecvt() performs conversions within static string buffers that it overwrites with each execution.

ed — Command

Interactive line editor
ed [-] [+cmopsv] [*file***]**

ed is the COHERENT system's interactive line editor.

ed is a line-oriented interactive text editor. With it, you can locate and replace text patterns, move or copy blocks of text, and print parts of the text. **ed** can read text from input files and can write all or part of the edited text to other files.

ed reads commands from the standard input, usually one command per line. Normally, **ed** does not prompt for commands. If the optional *file* argument is given, **ed** edits the given file, as if the *file* were read with the **e** command described below.

ed manipulates a copy of the text in memory rather than with the file itself. No changes to a file occur until the user writes edited text with the **w** command. Large files can be divided with **split** or edited with the stream editor **sed**.

ed remembers some information to simplify its commands. The *current line* is typically the line most recently edited or printed. When **ed** reads in a file, the last line read becomes the current line. The *current file name* is the last file name specified in an **e** or **f** command. The *current search pattern* is the last pattern specified in a search specification.

ed identifies text lines by integer line numbers, beginning with one for the first line. Several special forms identify a line or a range of lines, as follows:

LEXICON

n A decimal number *n* specifies the *n*th line of the text.

. A period '.' specifies the current line.

**$** A dollar sign '$' specifies the last line of the text.

+,- Simple arithmetic may be performed on line numbers.

/*pattern***/**
Search forward from the current line for the next occurrence of the *pattern*. If **ed** finds no occurrence before the end of the text, the search wraps to the beginning of the text. Patterns, also called *regular expressions,* are described in detail below.

?*pattern***?**
Search backwards from the current line to the previous occurrence of the *pattern*. If **ed** finds no occurrence before the beginning of the text, the search wraps to the end of the text.

'*x* Lines marked with the **k***x* command described below are identified by *'x*. The *x* may be any lower-case letter.

*n***,***m* Line specifiers separated by a comma ',' specify the range of lines between the two given lines, inclusive.

*n***;***m* Line specifiers separated by a semicolon ';' specify the range of lines between the two given lines, inclusive. Normally, **ed** updates the current line after it executes each command. If a semicolon ';' rather than a comma separates two line specifiers, **ed** updates the current line before reading the second.

* An asterisk '*' specifies all lines; it is equivalent to **1,$**.

Commands

ed commands consist of a single letter, which may be preceded by one or two specifiers that give the line or lines to which the command is to be applied. The following command summary uses the notations [**n**] and [**n**[,**m**]] to refer to an optional line specifier and an optional range, respectively. These default to the current line when omitted, except where otherwise noted. A semicolon ';' may be used instead of a comma ',' to separate two line specifiers.

. Print the current line. Also, a line containing only a period '.' marks the end of **a**ppended, **c**hanged, or **i**nserted text.

[*n***]** Print given line. If no line number is given (i.e., the command line consists only of a newline character), print the line that follows the current line.

[*n***]=** Print the specified line number (default: last line number).

[*n***]&** Print a screen of 23 lines; equivalent to **n**,**n**+*22p*.

! *line* Pass the given *line* to the shell **sh** for execution. **ed** prompts with an exclamation point '!' when execution is completed.

? Print a brief description of the most recent error.

[*n***]a** Append new text after line *n*. Terminate new text with line that contains only a period '.'.

[*n***[,***m***]]c**
Change specified lines to new text. Terminate new text with a line that contains only a period '.'.

[*n***[,***m***]]d[p]**
Delete specified lines. If **p** follows, print new current line.

e [*file***]** Edit the specified *file* (default: current file name). An error occurs if there are unsaved changes. Reissuing the command after the error message forces **ed** to edit the *file*.

E [*file***]** Edit the specified *file* (default: current file name). No error occurs if there are unsaved changes.

f [*file***]** Change the current file name to *file* and print it. If *file* is omitted, print the current file name.

[*n***[,***m***]]g/[***pattern***]/***commands*
Globally execute *commands* for each line in the specified range (default: all lines) that contains the *pattern* (default: current search pattern). The *commands* may extend over several lines, with all but the last terminated by '\'.

[*n***]i** Insert text before line *n*. Terminate new text with a line that contains only a period '.'.

[*n***[,***m***]]j[p]**
Join specified lines into one line. If *m* is not specified, use range *n*,*n*+1. If no range is specified, join the current line with the next line. With optional **p**, print resulting line.

[*n***]k***x* Mark given line with lower-case letter *x*.

[*n***[,***m***]]l** List selected lines, interpreting non-graphic characters.

[*n***[,***m***]]m[***d***]**
Move selected lines to follow line *d* (default: current line).

o *options*
Change the given *options*. The *options* may consist of an optional sign '+' or '-', followed by one or more of the letters '*cmopsv*'. Options are explained below.

[*n***[,***m***]][p]**
Print selected lines. The **p** is optional.

q Quit editing and exit. An error occurs if there are unsaved changes. Reissuing the command after the error message forces **ed** to exit.

Q Quit editing and exit. Throw away all changes that you have not yet saved to disk.

[*n***]r [***file***]**
Read *file* into current text after given line (default: last line).

[n[,*m***]]s[***k***]/[***pattern1***]/***pattern2***/[g][p]**
Search for *pattern1* (default, remembered search pattern) and substitute *pattern2* for *k*th occurrence (default, first) on each line of the given range. If **g** follows, substitute every occurrence on each line. If **p** follows, print the resulting current line.

[*n***[,***m***]]t[***d***]**
Transfer (copy) selected lines to follow line *d* (default, current line).

[*n***]u[p]** Undo effect of last substitute command. If optional **p** specified, print undone line. The specified line must be the last substituted line.

[*n***[,***m***]]v/[***pattern***]/***commands***
Globally execute *commands* for each line in the specified range (default: all lines) *not* containing the *pattern* (default: current search pattern). The *commands* may extend over several lines, with all but the last terminated by '\'. The **v** command is like the **g** command, except the sense of the search is reversed.

[*n***[,***m***]]w [***file***]**
Write selected lines (default, all lines) to *file* (default, current file name). The previous contents of *file*, if any, are lost.

[*n***[,***m***]]W [***file***]**
Write specified lines (default, all lines) to the end of *file* (default, current file name). Like **w**, but appends to *file* instead of truncating it.

Patterns

Substitution commands and search specifications may include *patterns*, also called *regular expressions*. A non-special character in a pattern matches itself. Special characters include the following.

^ Match beginning of line, unless it appears immediately after '[' (see below).

$ Match end of line.

* Matches zero or more repetitions of preceding character.

. Matches any character except newline.

[*chars***]** Matches any one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

[^*chars***]**
Matches any character *except* one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

LEXICON

*****c* Disregard special meaning of character *c*.

\(*pattern***\)**
Delimit substring *pattern* for use with **\d**, described below.

The replacement part *pattern2* of the substitute command may also use the following:

& Insert characters matched by *pattern1*.

\d Insert substring delimited by *d*th occurrence of delimiters '\(' and '\)', where *d* is a digit.

Options

The user may specify **ed** options on the command line, in the environment, or with the **o** command. The available options are as follows:

c Print character counts on **e**, **r**, and **w** commands.

m Allow multiple commands per line.

o Print line counts instead of character counts on **e**, **r**, and **w** commands.

p Prompt with an '*' for each command.

s Match lower-case letters in a *pattern* to both upper-case and lower-case text characters.

v Print verbose versions of error messages.

The **c** option is normally set, and all others are normally reset. Options may be set on the command line with a leading '+' sign. The '-' command line option resets the **c** option.

Options may be set in the environment with an assignment, such as

```
export ED=+cv
```

Options may be set with the '+' prefix or reset with the '-' prefix.

See Also

commands, elvis, ex, me, sed, vi
Introduction to the ed Line Editor

Diagnostics

ed usually prints only the diagnostic '?' on any error. When the verbose option **v** is specified, the '?' is followed by a brief description of the nature of the error.

EDITOR — Environmental Variable

Name editor to use by default
EDITOR=*editor*

The environmental variable **EDITOR** names the default editor that you wish to use. For example, **mail** invokes *editor* when you conclude a mail message by typing a question mark '?' at the beginning of a line followed by **<return>**. The screen pager **more** invokes *editor* when you enter the command **v** while displaying a file.

See Also

environmental variables, mail, more

egrep — Command

Extended pattern search
egrep [-Abcefhily] [*pattern***] [***file* **…]**

egrep is an extended and faster version of **grep**. It searches each *file* for occurrences of *pattern* (also called a regular expression). If no *file* is specified, it searches the standard input. Normally, it prints each line matching the *pattern*.

Wildcards

The simplest *patterns* accepted by **egrep** are ordinary alphanumeric strings. Like **ed**, **egrep** can also process *patterns* that include the following wildcard characters:

^ Match beginning of line, unless it appears immediately after '[' (see below).

$ Match end of line.

***** Match zero or more repetitions of preceding character.

. Match any character except newline.

[*chars***]**
Match any one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

[^*chars***]**
Match any character *except* one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

*****c Disregard special meaning of character *c*.

Metacharacters

In addition, **egrep** accepts the following additional metacharacters:

| Match the preceding pattern *or* the following pattern. For example, the pattern **cat | dog** matches either **cat** or **dog**. A newline within the *pattern* has the same meaning as '|'.

+ Match one or more occurrences of the immediately preceding pattern element; it works like '*', except it matches at least one occurrence instead of zero or more occurrences.

? Match zero or one occurrence of the preceding element of the pattern.

(...) Parentheses may be used to group patterns. For example, **(Ivan)+** matches a sequence of one or more occurrences of the four letters 'I' 'v' 'a' or 'n'.

Because the metacharacters '*', '?', '$', '(', ')', '[', ']', and '|' are also special to the shell, patterns that contain those literal characters must be quoted by enclosing *pattern* within apostrophes.

Options

The following lists the available options:

-A Write all lines in which *expression* is found into a temporary file. Then, call COHERENT with its error option to process the source file, with the contents of the temporary file serving as an "error" list. This option resembles the **-A** option to the **cc** command, and lets you build a COHERENT script to make systematic changes to the source file. To exit COHERENT and prevent **egrep** from searching further, **<ctrl-U> <ctrl-X> <ctrl-C>**.

Unlike **cgrep**, **egrep** only matches patterns that are on a single line. Some systems have a context **grep cgrep**) that works like **egrep** but displays lines found in context. The COHERENT **egrep -A** not only displays lines in context, via COHERENT, it lets you edit them.

-b With each output line, print the block number in which the line started (used to search file systems).

-c Print how many lines match, rather than the lines themselves.

-e The next argument is *pattern* (useful if the pattern starts with '-').

-f The next argument is a file that contains a list of patterns separated by newlines; there is no *pattern* argument.

-h When more than one *file* is specified, output lines are normally accompanied by the file name; **-h** suppresses this.

-i Ignore case when matches alphabetic letter in *pattern*. **egrep** takes case into account, even with this option, when you prefix a letter in *pattern* with '\'.

-l Print the name of each file that contains the string, rather than the lines themselves. This is useful when you are constructing a batch file.

-n When a line is printed, also print its number within the file.

-s Suppress all output, just return exit status.

-v Print a line only if the pattern is *not* found in the line.

LEXICON

-y Lower-case letters in the pattern match only upper-case letters on the input lines. A letter escaped with '\' in the pattern must be matched in exactly that case.

Limits

The COHERENT implementation of **egrep** sets the following limits on input and output:

Characters per input record	512
Characters per output record	512
Characters per field	512

See Also

awk, cgrep, commands, ed, expr, grep, lex, sed

Diagnostics

egrep returns an exit status of zero for success, one for no matches, and two for error.

Notes

For matching patterns in C programs, the command **cgrep** is preferred, because it is optimized to recognize C-style expressions.

Besides the difference in the range of patterns allowed, **egrep** uses a deterministic finite automaton (DFA) for the search. It builds the DFA dynamically, so it begins doing useful work immediately. This means that **egrep** is is much faster than **grep**, often by more than an order of magnitude, and is considerably faster than earlier pattern-searching commands, on almost any length of file.

else — C Keyword

Introduce a conditional statement

else is the flip side of an **if** statement: if the condition described in the **if** statement fails, then the statements introduced by the **else** statement are executed. For example,

```
if (getchar() == EOF)
      exit(0);
else
      dosomething();
```

exits if the user types **EOF**, but does something if the user types anything else.

See Also

C keywords, if
ANSI Standard, §6.6.4.1

elvis — Command

Clone of Berkeley-standard screen editor
elvis [*options* **] [+***cmd* **] [** *file1 ... file27* **]**

elvis is a clone of **vi** and **ex**, the standard UNIX screen editors.

elvis is a modal editor whose command structure resembles the **ed** line editor. *Modal* means that a keystroke assumes a different meaning, depending upon the mode that the editor is in. **elvis** uses three modes: *visual-command mode, colon-command mode,* and *input mode.*

The following sections summarize the commands associated with each mode:

Visual-Command Mode

Visual-command mode closely resembles text-input mode. One quick way to tell the modes apart is to press the **<esc>** key. If **elvis** beeps, then you are in visual-command mode. If it does not beep, then you were in input mode, but pressing **<esc>** switched you to visual-command mode.

Most visual-mode commands are one keystroke long. The commands are in two groups: movement commands and edit commands. The former group moves the cursor through the file being edited, and the latter group alters text.

The following sections summarize the command set for **elvis**'s visual-command mode.

Visual-Mode Movement Commands

The following summarizes the visual mode's movement commands. *count* indicates that the command can be optionally prefaced by an argument that tells **elvis** how often to execute the command. *move* indicates that the command can be followed by a movement command, after which the command is executed on the text that lies between the point where the command was first typed and the point to which the cursor was moved. Typing the command a second time executes the command for the entire line upon which the cursor is positioned. *key* means that the command must be followed by an argument.

<ctrl-B>	Move up by one screenful.
[count] **<ctrl-D>**	Scroll down *count* lines (default, one-half screenful).
[count] **<ctrl-E>**	Scroll up *count* lines.
<ctrl-F>	Move down by one screenful.
<ctrl-G>	Show file status and the current line.
[count] **<ctrl-H>**	Move one character to the left.
[count] **<ctrl-J>**	Move down *count* lines.
<ctrl-L>	Redraw the screen.
[count] **<ctrl-M>**	Move to the beginning of the next line.
[count] **<ctrl-N>**	Move down *count* lines (default, one).
[count] **<ctrl-P>**	Move up *count* lines (default, one).
<ctrl-R>	Redraw the screen.
<ctrl-T>	Pop the tag stack — that is, return to the most recently tagged position. **elvis** removes that tag from the tag stack.
[count] **<ctrl-U>**	Scroll up *count* lines (default, one-half screenful).
[count] **<ctrl-X>**	Move the cursor to column *count* on the current line.
[count] **<ctrl-Y>**	Scroll down *count* lines.
<ctrl-]>	If the cursor is on a tag name, go to that tag.
<ctrl-^>	Switch to the previous file.
[count] **<space>**	Move right *count* spaces (default, one).
<quotation mark> *key*	Select which cut buffer to use next.
$	Move to the end of the current line.
%	Move to the matching (){}[] character.
[count] **%**	Move *count* percentage into the file. For example, the command **50%** moves the cursor to the middle of the file.
' *key*	Move to a marked line.
[count] **(**	Move backward *count* sentences (default, one).
[count] **)**	Move forward *count* sentences (default, one).
*	Go to the next error in the error list.
[count] **+**	Move to the beginning of the next line.
[count] **,**	Repeat the previous **f** or **t** command, but move in the opposite direction.
[count] **-**	Move to the beginning of the preceding line.
[count] **.**	Repeat the previous *edit* command.
/ *text*	Search forward for *text,* which can be a regular expression.
0	If not part of a count, move to the first character of this line.
:	Switch to colon-command mode to execute one command.
[count] **;**	Repeat the previous **f** or **t** command.
? *text*	Search backwards for *text,* which can be a regular expression.
@ *key*	Execute the contents of a cut-buffer as **vi** commands.
[count] **B**	Move backwards *count* words (default, one).
[count] **E**	Move forwards *count* words (default, one).
[count] **F** *key*	Move left to the *count*'th occurrence of the given character (default, first).
[count] **G**	Move to to line *count* (default, last).
[count] **H**	Move to the top of the screen.
[count] **L**	Move to the bottom of the screen.
M	Move to the middle of the screen.
N	Repeat the last search, but in the opposite direction.
P	Paste text before the cursor.
Q	Shift to colon-command mode.
[count] **T** *key*	Move left *almost* to the given character.
U	Undo all recent changes to the current line.

V *[move...][command]*
: Like **v**, described below, except it applies to whole lines. For example, the command **Vjj>** first highlights and then indents three lines. It is equivalent to **>2j** or **3>>**.

[count] **W**
: Move forward *count* words (default, one).

[count] **Y**
: Copy (or "yank") *count* lines into a cut buffer (default, one).

Z Z
: Save the file and exit.

[[
: Move back one section.

]]
: Move forward one section.

^
: Move to the beginning of the current line, but after indent.

` *key*
: Move to the *key* character.

[count] **b**
: Move back *count* words.

[count] **e**
: Move forward to the end of the *count*'th word.

[count] **f** *c*
: Move rightward to the *count*'th occurrence of character *c*.

[count] **h**
: Move left *count* characters (default, one).

[count] **j**
: Move down *count* characters (default, one).

[count] **k**
: Move up *count* characters (default, one).

[count] **l**
: Move right *count* characters (default, one).

m *key*
: Mark a line or character.

n
: Repeat the previous search.

p
: Paste text after the cursor.

[count] **t** *key*
: Move rightward *almost* to the *count*'th occurrence of the given character (default, one).

u
: Undo the previous edit command.

v *[move ...][command]*
: Highlight text as the cursor is moved, then apply *command* to the highlighted text. For example, **vwwwd** is approximately the same as **3dw**. To cancel the selection without altering the text, press **v** a second time.

[count] **w**
: Move forward *count* words (default, one).

y *move*
: Copy (or "yank") text into a cut buffer.

z *key*
: Scroll the screen, repositioning the current line as follows: **+** indicates top of the screen, **—** indicates the bottom, **.** indicates the middle.

[count] **{**
: Move back *count* paragraphs (default, one).

[count] **|**
: Move to the *count*'th column on the screen (leftmost, one).

[count] **}**
: Move forward *count* paragraphs (default, one).

If you are running **elvis** within an **X** terminal window, you can use also the mouse to reposition the cursor. To bypass this feature (e.g., to perform the standard X cut-and-paste tasks), press **<shift>** while clicking a mouse button.

Visual-Mode Edit Commands

The following describes the visual mode's editing commands.

! *[move]*
: Run the selected text through an external filter program.

!!
: Replace the current line with the output of an external command.

[count] **#**
: Increment a number by *count* (default, one).

[count] **&**
: Repeat the previous **:s//** command *count* times (default, once).

< *move*
: Shift the enclosed text left.

=*[move]*
: Filter the affected text. The default filter is **fmt**, which performs simple paragraph formatting and word wrap. To change the filter used by **=**, use the command **:set ep=***filter_name*.

> *move*
: Shift the enclosed text right.

[count] **A** *input*
: Append input to end of the line.

C *input*
: Change text from the cursor through the end of the line.

D
: Delete text from the cursor through the end of the line.

[count] **I** *input*
: Insert text at the beginning of the line (after indentations).

[count] **J**
: Join lines the current with the following line.

K
: Look up the word under the cursor. The default lookup program is **ref**. You can change **K** so as to get C language run-time library help for the word under the cursor by executing the command:

```
set kp="help -f/usr/lib/helpfile -i/usr/lib/helpindex -d@"
```

You can write this line into file **$HOME/.exrc**, which **elvis** reads before it begins execution.

[count] **O** *input*	Open a new line above the current line.
R *input*	Overtype.
[count] **S** *input*	Change lines, like **cc**.
[count] **X**	Delete *count* characters from the left of the cursor (default, one).
[count] **a** *input*	Insert text after the cursor.
c *move*	Change text.
d *move*	Delete text.
[count] **i** *input*	Insert text at the cursor.
[count] **o** *input*	Open a new line below the current line.
[count] **r** *key*	Replace *count* characters with text you type (default, one).
[count] **s** *input*	Replace *count* characters with text you type (default, one).
[count] **x**	Delete the character at which the cursor is positioned.
****	Pop up a menu of the most common operations.
[count] **~**	Toggle a character between upper case and lower case.

Colon-Mode Commands

The following summarizes the set of colon-mode commands. It is no accident that these commands closely resemble those for the **ed** line editor: they come, in fact, from **ex**, the editor upon which both **vi** (the UNIX visual editor) and **ed** derive. For that reason, colon-command mode is sometimes called **ex** mode.

line indicates whether the command can be executed on one or more lines. *line* can be a regular expression. Some commands can be used with an optional exclamation point; if done so, the editor assumes you know what you are doing and suppresses the warnings and prompts it would normally issue for these commands.

Most commands can be invoked simply by typing the first one or two letters of their names.

abbr *[word full_form]*
Define *word* as an abbreviation for *full_form*.

and This command is used with the colon-mode command **if** to execute other commands conditionally. It is never used on its own. For more information, see the section *Conditional Commands* , below.

[line] **append** Insert text after the current line.

args *[file1 ... fileN]* With no arguments, print the files list on **elvis**'s command line. With one or more arguments, change the name of the current file.

cc *[files]* Invoke the C compiler to compile *files*, and redirects all error messages into file **errlist**. After the compiler exits, scan the contents of **errlist** for error messages; if one is found, jump to the line and file indicated on the error line, and display the error message on the status line.

cd *[directory]* Switch the current working directory. With no argument, switch to the **$HOME** directory.

[line][,line] **change** *['x]*
Replace the range of lines with the contents of cut-buffer *x*.

chdir *[directory]* Same as the **cd** command.

color *[when] [[type] color] [on color]*
Set the screen's colors. This command works only if you have an ANSI-compatible color terminal. *when* defines the type of text whose color is being manipulated: **normal**, **standout**, **bold**, **underlined**, **italic**, **popup**, and **quit**. The default is normal. You may use the first letter of each as an abbreviation. *color* can be one of the following:

black	**blue**	**green**	**cyan**
red	**magenta**	**brown**	**white**
yellow	**gray**	**grey**	

Valid color *type*s can be one of the following: **light**, **bright**, or **blinking**.

The first use of **color** *must* specify both the foreground and background colors; the background color thereafter defaults to the background color of normal text. For example, the commands

```
color light cyan on blue
color b bright white
```

set the normal text to light cyan in the foreground and a blue background; and then set the foreground color for bold text to bright white.

Not every valid **color** command works as expected on the system console, due to limitations in the current release of the **ansipc** device driver.

[line][,line] **copy** *targetline*
Copy the range of lines to after the *targetline*.

[line][,line] **delete** *['x]*
Move the range of lines into cut buffer *x*.

digraph*[!] [XX [Y]]* Set *XX* as a digraph for *Y*. With no arguments, display all currently defined digraphs. With one argument, undefine the argument as a digraph.

edit*[!] [file]* Edit a file not named on the **elvis** command line.

else This command is used with the colon-mode command **if** to execute other commands conditionally. For more information, see the section *Conditional Commands* , below.

errlist*[!] [errlist]* Find the next error message in file **errlist**, as generated through **elvis**'s **cc** or **make** commands.

file *[file]* With an argument, change the output file to *file*. Without an argument, print information about the current output file.

[line][,line] **global** */regexp/ command*
Search the range of lines for all lines that contain the regular expression *regexp*, and execute *command* upon each.

if This command is used to execute other commands conditionally. For more information, see the section *Conditional Commands* , below.

[line] **insert** Insert text before the current line.

[line][,line] **join** Concatenate the range of lines into one line.

[line][,line] **list** Display the requested range of lines, making all embedded control characters explicit.

make *[target]* Same as the **cc** command, except that **make** is executed.

map*[!] key mapped_to*
Remap *key* to *mapped_to*. Normally, remapping applies just to visual-command mode; '!' tells **elvis** to remap the key under all modes. With no arguments, show all current key mappings.

[line] **mark** *x* Set a mark on *line*, and name it *x*.

mkexrc Save current configuration into file **./.exrc**, which will be read next time you invoke **elvis**.

[line][,line] **move** *targetline*
Move the range of lines to after *targetline*.

next*[!] [files]* Switch to the next file on the **elvis** command line.

Next*[!]* Switch to the preceeding file on the **elvis** command line.

[line][,line] **number** Display the range of lines, with line numbers.

or This command is used with the colon-mode command **if** to execute other commands conditionally. For more information, see the section *Conditional Commands* , below.

pop Pop the tag stack — that is, return to the most recently tagged position. **elvis** removes that tag from the tag stack.

previous*[!]* Switch to the preceeding file on the **elvis** command line.

[line][,line] **print** Display the specified range of lines.

[line] **put** *['x]* Copy text from cut buffer *x* after the current line.

quit*[!]* Quit **elvis**, and return to the shell.

[line] **read** *file* Read the contents of *file* and insert them after **line** (default, the last line).

rewind*[!]* Switch to the first file on the **elvis** command line.

*[line***[,***line]***]s/***oldstring***/***newstring***/***[g]*
Substitute the first instance of *newstring* for *oldstring*. If no range of lines is indicated, the substitution is performed only on the current line. To change every instance of *oldline* into *newline* on a line, append the suffix **g** ("global") to this command.

The command **s** with no arguments repeats the previous substitution. It is a synonym for the command **&**, described below.

set *[options]* Set an **elvis** option. For details, see the section on *set Options*, below

shell Invoke a shell.

source *file* Read a set of colon-mode commands from *file*, and execute them.

[line][,line] **substitute** */regexp/replacement/[p][g][c]*
For the range of lines, replace the first instance of *regexp* with *replacement*. *p* tells **elvis** to print the *last* line upon which a substitution was performed. *g* means perform a global substitution, i.e., replace all instances of *regexp* on each line with *replacement*. *c* tells **elvis** to ask for confirmation before performing each substitution.

tag*[!] tagname* Find *tagname* in file **tags**, which records information about all tags. If found, jump to the file and line upon which the tag is set.

then This command is used with the colon-mode command **if** to execute other commands conditionally. For more information, see the section *Conditional Commands* , below.

[line][,line] **to** *targetline*
Copy the range of lines to after the *targetline*.

unabbr *word* Unabbreviate *word*.

undo Undo the last editing command.

unmap*[!] key* Unmap *key*.

version Display the current version of **elvis**.

[line][,line] **vglobal** */regexp/ command*
Search the range of lines for all lines that do not contain the regular expression *regexp*, and execute *command* upon each.

visual Enter visual-command mode.

wq Save the changed file, and exit.

[line][,line] **write***[!] [[>>][file]*
Write the file being edited into *file*. With the >> argument, append the edited text onto the end of *file*.

xit*[!]* Same as the **wq** command, described above, except that it does not write files that have not changed.

[line][,line] **yank** *["x]* Copy the range of lines into cut buffer *x*.

[line][,line] **!** *command*
Execute *command* under a subshell, then return.

[line][,line] **<** Shift the range of lines left by one tabwidth.

[line][,line] **=** With no range of lines specified, print the number of the current line. With line arguments, print the endpoints of the lines in question, and the number of lines that lie between them. (Remember, *line* can be a regular expression as well as a number.)

[line][,line] **>** Shift the range of lines right by one tabwidth.

[line][,line] **&** Repeat the last substitution command.

@ x Read the contents of cut-buffer *x* as a set of colon-mode commands, and execute them. With no arguments, list all current settings.

\@ Beginning with release 1.8, **elvis** replaces the escape sequence **\@** by the word that the cursor is on. This works in two special contexts: in regular expressions, and in any **ex** command that also replaces '%' with the current file name. This escape sequence can simplify writing certain kinds of macros.

Conditional Commands

Beginning with release 1.8pl3, **elvis** supports conditional commands. Some of these commands set a conditional-execution flag; others examine that flag and perform (or do not perform) commands if the flag is set. You cannot nest conditional commands.

The colon-mode commands **if**, **and**, and **or** test for a condition. Their syntax is typical: each must be followed by an expression, and **and** and **or** must follow an initial **if** command. Each command tests for a single condition. That condition may involve examining the options set by the command **set** (which are described in detail in the next section), set by **termcap** values, or by constants.

Colon-mode commands **then** and **else** execute commands conditionally, based upon the value of the conditional-execution flag set by a preceding **if** command.

These commands most often are embedded in an initialization file, to initialize **elvis** properly under a variety of conditions. The following gives an example **if** command that can be embedded in a user's **.exrc** file. The command correctly sets up the colors for both the system console and for an X terminal window. It works around the fact that the console can handle color, but an X terminal window cannot:

```
if term="console"
then color yellow on blue | color quit white on blue
else color black on white
```

To disable these commands, add **-DNO_IF** to **CFLAGS**, then recompile **elvis**.

set Options

As noted above, the command **set** can set **elvis**'s internal options. Options come in three flavors: *boolean,* which turn on or off a feature of the editor; *string* which define the string associated with a particular action; (e.g., the name of a command or feature); and *numeric*, which set a dimension for the editor (e.g., the number of rows or columns on the terminal screen). To turn off a boolean option, prefix it with the string "no".

The following lists the options that **set** recognizes. Assume that the boolean options are on, unless the entry says otherwise:

LEXICON

autoindent (boolean) Auto-indent during input? Default is **no**.
autoprint (boolean) When in **ex** mode, print the current line.
autotab (boolean) Can auto-indent use tabs?
autowrite (boolean) Is auto-write on when switching files? Default is **no**.
beautify (boolean) Should the editor strip control characters from a file? Default is **no**.
charattr (boolean) Interpret \fX sequences? Default is **no**.
cc (string) Name of the C compiler. Default is **cc -c**.
columns (numeric) Width of the screen. Default is 80.
digraph (boolean) Recognize digraphs? Default is **no**.
directory (string) Where are temporary files kept? Default is **/usr/tmp**.
edcompatible (boolean) Remember ":s//" options? Default is **no**.
equalprg (boolean) Program to run for the '=' operator. Default is **fmt**.
errorbells (boolean) Beep when an error occurs.
exrc (boolean) Read the **./.exrc** file? Default is **no**.
exrefresh (boolean) Write lines individually when in **ex** mode.
flash (boolean) Use visible alternative to bell.
flipcase (string) Non-ASCII chars flipped by the tilde character '~'. Default is the NULL string.
hideformat (boolean) Hide text formatter commands.
ignorecase (boolean) Make searches case sensitive. While in **ignorecase** mode, the searching mechanism does not distinguish between an upper-case letter and its lower-case form. In **noignorecase** mode, upper case and lower case are treated as being different. Default is **no**.
inputmode (boolean) Start **vi** in insert mode? Default is **no**.
keytime (numeric) Timeout for mapped key entry. Default is two.
keywordprg (string) Path name of program invoked by **shift-K**. Default is **ref**.
lines (numeric) Number of lines on the screen. Default is 25.
list (boolean) Display lines in **list** mode? Default is **no**.
magic (boolean) Enable the use of regular expressions in a search. While in **magic** mode, all meta-characters behave as described above. In **nomagic** mode, only **^** and **$** retain their special meaning.
make (string) Name of the "make" program. Default is **make**.
mesg (boolean) Allow messages from other users?
modelines (boolean) Are mode lines processed? Default is **no**.
more (boolean) Pause between messages?
nearscroll (numeric) This governs when to scroll versus when to redraw the screen. If you move the cursor more than the number of lines set by this option, **elvis** redraws the screen; otherwise, it scrolls the screen. The default is 15 lines.
novice (boolean) Set options for ease of use? Default is **no**.
number (boolean) Turn on line numbering.
paragraphs (string) Names of **nroff** "paragraph" commands. Default is **PPppIPLPQP**.
prompt (boolean) Show ':' prompt in **ex** mode.
readonly (boolean) Prevent overwriting of original file. Default is **no**.
remap (boolean) Allow key maps to call other key maps.
report (numeric) Report when a given number of changes occur. Default is five.
ruler (boolean) Display line and column numbers. Default is **no**.
safer Toggle **elvis'** security option. This option is set temporarily during the execution of any command from modeline or **./.exrc**. It disables the following commands:

:!	**:Next**	**:abbreviate**	**:args**	**:cc**
:cd	**:chdir**	**:ex**	**:file**	**:make**
:map	**:mkexrc**	**:next**	**:pop**	**:previous**
:rewind	**:shell**	**:stop**	**:suspend**	**:tag**
:unab	**:unmap**	**:visual**	**:write**	

Note that **set safer** does not disable **:wq**, as this command does not let the user name the file into which to write. **:read** is still allowed, but it will not let the user read from a filter.

set safer forbids the user from altering the following options:

autowrite	**cc**	**directory**	**equalprg**
keywordprg	**make**	**shell**	**trapunsafe**

	It also disables wildcard expansion and the visual '!' command.
scroll (numeric)	Set the number of lines the screen scrolls with the **<ctrl-D>** and **<ctrl-U>** commands. Default is 12.
sections (string)	Names of **nroff** "section" commands. Default is **NHSHSSSEse**.
shell (string)	Path name of the shell. Default is **/bin/sh**.
showmatch (boolean)	Show all matching parentheses, brackets, and braces. Default is **no**.
showmode (boolean)	Say when editor is in input mode. Default is **no**.
shiftwidth (numeric)	Set number of characters the **<** and **>** commands shift the screen. Default is eight.
sidescroll (numeric)	Set number of columns the editor scrolls. Default is eight.
sync (boolean)	Call **sync()** often? Default is **no**.
tabstop (numeric)	Number of columns set by a tab character. Default is eight.
taglength (numeric)	Number of significant characters in a tag name. Default is zero.
tags (string)	Name the list of "tags" files that **elvis** can read.
tagstack (boolean)	Enable the tagstack. Default is **no**.
term (string)	Name of the current terminal's **termcap** entry. Default is **$TERM**.
terse (boolean)	Give shorter error messages? Default is **no**.
timeout (boolean)	Distinguish **<esc>** from an arrow key?
warn (boolean)	Warn if a file has been modified?
window (numeric)	Number of lines to redraw after long move. Default is 24.
wrapmargin (numeric)	Left margin to use when wrapping long lines in input mode. Default is zero.
wrapscan (boolean)	Searches wrap from end to beginning of the file.
writeany (boolean)	Let the write command **:w** clobber a file. Default is **no**.

Input-Mode Commands

Most keystrokes are interpreted as being text and inserted directly into the text; however, some keystrokes are still interpreted as commands. Thus, you can perform an entire session of simple editing directly within input mode without switching to either of the command modes.

The following summarizes the commands that can be executed directly within input mode:

<ctrl-A>	Insert a copy of the last input text.
<ctrl-C>	Send the signal **SIGINT** to interrupt a command.
<ctrl-D>	Delete one indent character.
<ctrl-H>	Erase the character before the cursor.
<ctrl-L>	Redraw the screen.
<ctrl-M>	Insert a newline.
<ctrl-P>	Insert the contents of the cut buffer.
<ctrl-R>	Redraw the screen, like **<ctrl-L>**.
<ctrl-T>	Insert an indent character.
<ctrl-U>	Move to the beginning of the line. When you are typing a command line or search pattern on the bottom line, **<ctrl-U>** backspaces over all characters typed so far.
<ctrl-V>	Insert the following keystroke, even if special.
<ctrl-W>	Backspace to the beginning of the current word.
<ctrl-Z><ctrl-Z>	Write the file and exit **elvis**.
<ctrl-Z>	Save the file if it has been modified, but do not exit from **elvis**. This works only if you have set the mode **autowrite**.
<esc>	Shift from input mode to visual-command mode.
<del>	Delete the current character.

When **elvis** is in input mode, you can use the keystroke **<ctrl-O>** to invoke *some* visual commands without exiting from input mode. For example, when you are in input mode, typing **<ctrl-O>J** moves down a line but leaves you in input mode.

Keyboard Macros

elvis Beginning with release 1.8, **elvis** can record keystokes into a cut buffer. This is equivalent to a MicroEMACS "keyboard macro".

The following commands manipulate keyboard macros:

[a Open a keyboard macro. **elvis** executes all subsequent keystrokes as normal, but also records them within a temporary buffer.

LEXICON

]a Stop recording keystrokes, and copy the keystrokes into the cut buffer.

@a To replay the recorded keystrokes.

Command-line Options

elvis lets you name up to 27 files on the command line, thus allowing you to edit up to 27 files simultaneously. The "next file" and "previous file" commands described above allow you to shift from one file to another during the same editing session; in this way, for example, you can cut text from one file and paste it into another.

elvis recognizes the following command-line options:

-r Recover a previous edit.
-R Invoke **elvis** in "read-only" mode. This is equivalent to invoking **elvis** via the link **view**.
-s Invoke **elvis** in "safer" mode. This is equivalent to the command **set safer**, described above.
-t *tag* Begin editing at *tag*.
-m *[file]*
Invoke **elvis** in error-handling mode. It searches through *file* for something that looks like an error message from a compiler, then positions the cursor at that point for editing.
-e Begin in colon-command mode.
-v Begin in visual-command mode.
-i Begin in input mode.
-w *winsize*
Set the value of option **window**, which sets the size of the screen with which **elvis** works, to *winsize*. **window** is described below.
+*command*
Execute *command* immediately upon beginning editing. For example

```
elvis +237 foo
```

causes **elvis** to move directly to line 237 immediately upon beginning to edit file **foo**.

Regular Expressions

elvis uses regular expressions for searching and substitutions. A regular expression is a text string in which some characters have special meanings. This is much more powerful than simple text matching.

elvis's **regexp** package treats the following one- or two-character strings (called meta-characters) in special ways:
\(\) Delimit subexpressions. When the regular expression matches a chunk of text, **elvis** remembers which portion of that chunk matched the subexpression. The command

```
:s/regexp/newtext/
```

command makes use of this feature.
^ Match the beginning of a line. For example, to find **foo** at the beginning of a line, use the regular expression **/^foo/**. Note that **^** is a metacharacter only if it occurs at the beginning of a regular expression; anywhere else, it is treated as a normal character.
$** Match the end of a line. It is a metacharacter only when it occurs at the end of a regular expression; elsewhere, it is treated as a normal character. For example, the expression **/$$/ searches for a dollar sign at the end of a line.
\< Match a zero-length string at the beginning of a word. A word is a string of one or more letters and digits; it can begin at the beginning of a line or after one or more non-alphanumeric characters.
\> Matches a zero-length string at the end of a word. A word can end at the end of the line or before one or more non-alphanumeric characters. For example, **/\<end>/** finds any instance of the word **end**, but ignores any instances of "end" inside another word, such as "calendar".
. Match any single character.
[*character-list***]**
Match any single character from the *character-list*. Inside the *character-list*, you can denote a span of characters by writing the first and last characters, with a hyphen between them. If the character-list is preceded by a **^**, then the list is inverted — it matches all characters not mentioned in the list. For example, **/[a-zA-Z]/** matches any letter, and **/[^]/** matches anything other than a blank.
\{*n***\}** Repeat the preceding expression *n* times. This operator can only be placed after something that matches a single character. For example, **/^-\{80\}$/** matches a line of eighty hyphens, and **/\<[a-zA-Z]\{4\}\>/** matches any four-letter word.
\{*n***,***m***\}** Repeat the preceding single-character expression between *n* and *m* times, inclusive. If the *m* is omitted (but the comma is present) then it is taken to be infinity. For example, **/"[^"]\{3,5\}"/** matches any pair of quotation marks that enclose three, four, or five non-quotation characters.

* Repeat the preceding single-character expression zero or more times. For example, **/.*/** matches a whole line.

/+ Repeat the preceding single-character expression one or more times. It is equivalent to **\{1,\}**. For example, **/.\+/** matches a whole line, but only if the line contains at least one character. It does not match empty lines.

/? The preceding single-character expression is optional — that is, that it can occur zero or one times. It is equivalent to **\{0,1\}**. For example, **/no[-]?one/** matches **no one**, **no-one**, and **noone**.

Anything else is treated as a normal character that must exactly match a character from the scanned text. The special strings may all be preceded by a backslash to force them to be treated normally.

Substitutions

The command **:s** has at least two arguments: a regular expression and a substitution string. The text that matches the regular expression is replaced by text that is derived from the substitution string.

Most characters in the substitution string are copied into the text literally but a few have special meaning:

& Insert a copy of the original text.
~ Insert a copy of the previous replacement text.
\1 Insert a copy of that portion of the original text that matched the first set of parentheses.
\2-\9 Do the same for the second and all subsequent pairs of parentheses.
\U Convert all characters of any later **&** or **\#** to upper case.
\L Convert all characters of any later **&** or **\#** to lower case.
\E End the effect of **\U** or **\L**.
\u Convert the first character of the next **&** or **#** to upper case.
\l Convert the first character of the next **&** or **\#** to lower case.

These may be preceded by a backslash to force them to be treated normally.

If **nomagic** mode is in effect, then **&** and **~** will be treated normally, and you must write them as **\&** and **\~** for them to have special meaning.

Preserving Text

Should **elvis** sense that it is about to die unexpectedly, it invokes the command **elvprsv** to save the temporary file in which it manipulates the file you are editing. To recover this saved file, use the command **elvrec**. Both commands are described in the Lexicon.

Initialization Files

When you invoke **elvis**, it searches for file **$HOME/.exrc**. If it finds that file, it reads the file and attempts to execute its contents as a series of **ex** commands. (As noted earlier, **ex** commands simply are **elvis'** set of colon-mode commands, but without the preceding colon.)

Usually, this file is used to contain instances of the commands **set** and **color**, to set up **elvis'** environment and appearance to your taste. For example, if your **.exrc** file contains the commands

```
color white on blue
set ignorecase
set inputmode
```

then **elvis** sets the screen's background color to blue and its foreground color to white; turn on **ignorecase** mode (that is, string searches will ignore case), and come up in input mode rather than command mode.

The file **$HOME/elvis.rc** is a synonym for **$HOME/.exrc**.

When you invoke **elvis**, it also searches for the file **$HOME/.exfilerc** This file holds **ex** commands that **elvis** executes every time it loads a text file for editing. You can embed **if** commands in this file so that **elvis** handles special classes of files uniquely. For example, you can use an **if** command to tell **elvis** to handle files with the suffix **.c** differently from other files; this lets you invoke special editing functions for C programs.

Examples

The first example changes every occurrence of "utilize" to "use":

```
:%s/utilize/use/g
```

The next example deletes all white space that occurs at the end of a line anywhere in the file. (The brackets contain a single space and a single tab character):

LEXICON

```
:%s/[    ]+$//
```

The next example converts the current line to upper case:

```
:s/.*/U&/
```

The next example underlines each letter in the current line, by changing it into an **underscore backspace letter** sequence. (The **<ctrl-H>** is entered as **<ctrl-V><backspace>**):

```
:s/[a-zA-Z]/_^H&/g
```

The last example locates the last colon in a line, and swaps the text before the colon with the text after the colon. The first pair of parentheses delimits the stuff before the colon, and the second pair delimits the stuff after. In the substitution text, **\1** and **\2** are given in reverse order to perform the swap:

```
:s/\(.*\):\(.*\)/\2:\1/
```

Environment

elvis reads the following environmental variables:

TERM This names your terminal's entry in the **termcap** or **terminfo** data base.

TERMCAP
Optional. If your system uses **termcap**, and the **TERMCAP** variable is not set, then **elvis** reads your terminal's definition from **/etc/termcap**. If **TERMCAP** is set to the full path name of a file (beginning with a '/'), it reads your terminal's description from the named file instead of from **/etc/termcap**. If **TERMCAP** is set to a value that does not begin with a '/', then **elvis** assumes that its value is the full **termcap** entry for your terminal.

TERMINFO
Optional. **elvis** treats this exactly like the environmental variable **TERMCAP**, except for the **terminfo** data base.

LINES
COLUMNS
Optional. These variables, if set, override the screen-size values given in the **termcap** or **terminfo** description of your terminal On windowing systems such as X, **elvis** has other ways to determine the screen size, so you should probably leave these variables unset.

EXINIT
Optional. This variable can hold **ex** commands that **elvis** executes before it reads any **.exrc** files.

SHELL Optional. This variable sets the default value for the **shell** option, which determines which shell program **elvis** uses to perform wildcard expansion in file names, and to execute filters or external programs. The default value is **/bin/sh**.

HOME This variable should be set to the name of your home directory. **elvis** looks for its initialization file there. If **HOME** is not set, then **elvis** does not execute the initialization file.

TAGPATH
Optional. This variable is used by the program **ref**. See "ref" for more information.

Bug Fixes from Release 1.7

Beginning with release 4.2.10, COHERENT includes **elvis** release 1.8pl3. The following describes the bugs that this release fixes The initial release of **elvis** 1.8 includes the following bug fixes:

- Most screen update bugs are fixed. Most of ones that were not fixed can be avoided by **:set nooptimize**.
- A bug in the visual '@' command was fixed. This bug can be blamed for most of **elvis**' incompatibility with fancy macro packages. **elvis** can now run the "Bouncing Ball," "Word Completion," and "Turing" macros with no changes. NB, it still cannot run "Towers of Hanoi."

The following bug fixes are included in patch-level 1 (**pl1**):

- Fixed a bug that caused core dump when you use the '}' command used on blank line after last paragraph in file.
- Fixed a bug that caused loss of text with **AutoIndent** enabled, when two newlines are inserted into the middle of a line.

The following bug fixes are included in patch-level 2 (**pl2**):

- Fixed a security hole on some UNIX systems.
- After **:w**, **#** refers to the file just written.
- Fixed bug in tag lookup.
- The compiler error parser now allows '_' in a file name.
- Fixed a bug that caused some blank lines in the file **.exrc** to be interpreted as **:p** commands.
- Increased the limit on word size for the command **<ctrl-A>**. The old limit was 30; the new limit is 50. If you exceed this limit, **elvis** will now search for the longest possible substring; before, it would bomb. To change the limit, add **-DWSRCH_MAX=***n* (where *n* gives the limit on word size) to **CFLAGS** in the **Makefile**, then recompile **elvis**.
- Increased the size of an array used while showing option settings. The old size could overflow if you did a **:set all** on some systems. Now, the maximum size is calculated at compile time, and the array is declared to this size.
- The command **5r<ctrl-M>** now leaves the cursor in the right place. In earlier releases, **5r<ctrl-M>** would replace five characters with five newline characters, and leave the cursor five lines lower. Release 1.8 replaced five characters with a single newline character, to mimic the real **vi** better, but still left the cursor five lines lower. This patch finally makes it right.

The following bug fixes are included in patch-level 2 (**pl2**):

- Corrected bugs in **:tag** and **:make**, which caused tag addresses and error messages to be forgotten after switching files. The **.exfilerc** feature interacted with these bugs, and made them pretty obnoxious. A similar bug caused the command **:e +cmd file** to start misbehaving; it has been fixed, too.
- The option **window** now defaults to zero. Zero is a special value, which means "use as many rows as possible." Previously, this option defaulted to the maximum number of rows available when **elvis** started (usually 24), which resulted in '@' signs appearing on the screen if you resized the display while **elvis** was running. This problem only showed up when you ran **elvis** in an X terminal window.
- A bug has been fixed in autoindentation. Previously, if you inserted a newline before the first non-whitespace character on a line, then everything after the insertion point was wiped out. (This is different from the bug that **pl2** fixed. **pl2**'s fix addresses a bug that affected insertion of multiple newlines anywhere in a lines; this one affects inserting a single newline before the first non-whitespace character.)
- To avoid linking problems on various systems, the variable **kD** has been renamed **kDel**, and function **ioctl()** in **pc.c** renamed **elvis_ioctl**.
- A bug that caused **:!** to clobber the value of **#** (i.e., the previous file name) has been fixed.
- There is a bug that affects screen redraws after pasting (the visual **p** and **P** commands). In an attempt to work around this bug, **elvis** will sometimes redraw the screen from scratch after a multi-line paste.
- Some people have reported problems using **fmt** on non-English text. I suspect that this is due to a faulty implementation of **isspace()** in the standard C library. In release 1.8pl3, **fmt** does not use **isspace()** anymore; it uses a built-in macro which explicitly tests for **<space>** or **<tab>**. This may solve the problem.

Files

/tmp/elv* — Temporary files
tags — Data base used by the **tags** command
$HOME/.exrc — File that sets personal defaults
$HOME/.exfilerc — File that sets defaults when a file is read
$HOME/elvis.rc — Same as **.exrc**

See Also

commands, ed, elvprsv, elvrec, ex, fmt, me, vi, view

Notes

elvis returns zero if the file being edited was updated. It returns one if the file was not updated, and a different nonzero value if an error occurred.

LEXICON

Full documentation for **elvis** is included with this release in compressed file **/usr/src/alien/Elvis.doc.Z**.

elvprsv — Command

Preserve the modified version of a file after a crash
elvprsv ["-*why elvis died***"] /tmp/***filename***...**
elvprsv -R /tmp/*filename***...**

The command **elvprsv**, or "elvis preserved," preserves your edited text should **elvis** die unexpectedly. You can later use the command **elvrec** to rebuild the edited buffer.

You should never need to run **elvprsv** from the command line. **elvis** automatically invokes it should it sense that it is about to die. Script **/etc/rc** should also invoke **elvprsv**, to preserve any temporary files that may have been left in directory **/tmp** when the system went down.

If **elvis** were to die unexpectedly while you were editing a file, **elvprsv** would preserve the most recent version of your text. The preserved text is stored in a special directory; **elvprsv** does *not* overwrite your text file. **elvprsv** sends mail to each user whose work it preserves. Should the preservation directory not be set up correctly, **elvprsv** simply sends you a mail message that describes how to it manually.

Files

/tmp/elv*
: Temporary file that **elvis** was using when it died.

/usr/preserve/p*
: Text that is preserved by **elvprsv**.

/usr/preserve/Index
: Text file that names all preserved files and the files in which they are preserved.

See Also

commands, elvis, elvrec

Notes

Due to the permissions on directory **/usr/preserve**, only the superuser **root** can run **elvprsv**.

If you were editing a nameless buffer when **elvis** died, **elvprsv** saves its contents in a file named **foo**.

elvprsv was written by Steve Kirkendall (kirkenda@cs.pdx.edu).

elvrec — Command

Recover the modified version of a file after a crash
elvrec [*preservedfile* **[***newfile***]]**

Should **elvis** die while you were editing a file, it automatically invokes the command **elvprsv** to preserve the most recent version of your edited text. **elvprsv** stores the preserved text in a special directory: it does *not* overwrite your text file

The command **elvrec** locates the preserved version of a file, and either overwrites your text file or creates a new file, whichever you prefer. The recovered file will hold nearly all of your changes.

To see a list of all recoverable files, run **elvrec** with no argument. *preservedfile* names the file into which **elvprsv** had saved the edited buffer. **elvrec** is very picky about file names: you must use exactly the same path name as you did to edit the file.

newfile names the file into which **elvrec** writes the edited buffer. If you do not name a *newfile* on its command line, **elvrec** overwrites your original file with the preserved, edited version.

Files

/usr/preserve/p*
The text that was preserved when **elvis** died.
/usr/preserve/Index
The names of all preserved files, and the names of the files that preserve their text.

See Also

commands, elvis, elvprsv

Notes

Due to the permissions on the directory **/usr/preserve**, only the superuser **root** can run **elvrec**.

If you haven't set up a directory for file preservation, then you must manually run the program **elvprsv** instead of **elvrec**.

If you were editing a nameless buffer when **elvis** died, then **elvrec** saves the text into a file named **foo**.

elvrec was written by Steve Kirkendall (kirkenda@cs.pdx.edu).

em87 — Kernel Module

Perform/emulate hardware floating-point operations

The kernel module **em87** performs or emulates hardware floating-point operations. Whether it performs the operations or emulates them depends whether your computer contains a mathematics co-processor. Note that the Intel 80486-DX processor has the co-processor built in.

em87 is called a *kernel module* because you can link it into the kernel or exclude it from the kernel, just like a device driver. However, it is not a true device driver because it does not perform I/O from a peripheral device. To install **em87** into a kernel (should your kernel not already contain it), log in as the superuser **root** and execute the following commands:

```
cd /etc/conf
em87/mkdev
bin/idmkcoh -o /kernel_name
```

where *kernel_name* is the name of the new kernel to build. When you next boot COHERENT, hardware floating point will be enabled.

See Also

device drivers, float, kernel

emacs — Command

COHERENT screen editor
emacs [-e *errorfile***] [-f** *bindfile***] [***textfile ...***]**

emacs is a link to the COHERENT screen editor, which is a scaled-down version of the EMACS screen editor.

For details, see the Lexicon entry for **me**.

See Also

commands, me

enable — Command

Enable a port
/etc/enable *port...*

The COHERENT system is a multiuser operating system; it allows many users to use the system simultaneously. An asynchronous communication *port* connects each user to the system, normally by a terminal or a modem attached to the port. The system communicates with the port by means of a character special file in directory **/dev**, such as **/dev/com3r** or **/dev/com2l**.

The COHERENT system will not allow a user to log in on a port until the system creates a *login process* for the port. The **enable** command tells the system to create a login process for each given *port*. For example, the command

```
/etc/enable com1r
```

enables port **/dev/com1r**.

enable changes the entry for each given *port* in the terminal characteristics file **/etc/ttys**. The baud rate specified in **/etc/ttys** must be the appropriate baud rate for the terminal or modem connected to the port. See the Lexicon entry for **ttys** for more information.

The command **disable** disables a port. The command **ttystat** checks whether a port is enabled or disabled.

Files

/etc/ttys — Terminal characteristics file
/dev/com* — Devices serial ports

See Also

asy, commands, disable, getty, login, ttys, ttystat

Diagnostics

enable normally returns one if it enables the *port* successfully and zero if not. If more than one *port* is specified, **enable** returns the success or failure status of the last port it finds. It returns -1 if it cannot find any given *port*. An exit status of -2 indicates an error.

Notes

It is not recommended that you attempt to enable a port that is already enabled. To make sure, run **/etc/disable** before running **/etc/enable**.

endgrent() — General Function (libc)

Close group file
#include <grp.h>
endgrent()

endgrent() closes the file **/etc/group**. It returns NULL if an error occurs.

Files

/etc/group
<grp.h>

See Also

group, libc

endhostent() — Sockets Function (libsocket)

Close file /etc/hosts
#include <netdb.h>
void endhostent();

The function **endhostent()** is one of a set of functions that interrogate the file **/etc/hosts** to look up information about a remote host on a network. It closes **/etc/hosts** upon the conclusion of searching.

See Also

gethostbyaddr(), gethostbyname(), libsocket, sethostent()

endnetent() — Sockets Function (libsocket)

Close network file
#include <netdb.h>
void endnetent();

Function **endnetent()** closes file **/etc/networks** which describes all entities on your local network, after it had been opened by function **getnetent()** or **setnetent()**.

See Also

getnetbyaddr(), getnetent(), libsocket, netdb.h, setnetent()

endprotoent() — Sockets Function (libsocket)

Close protocols file
#include <netdb.h>
void endprotoent();

Function **endprotoent()** closes file **/etc/protocols** which describes all protocols recognized by your local network, after it had been opened by function **getprotoent()** or **setprotoent()**.

See Also

getprotobyaddr(), getprotobyname(), getprotoent(), libsocket, netdb.h, setprotoent()

endpwent() — General Function (libc)

Close password file
#include <pwd.h>
endpwent()

The COHERENT system has five routines that search the file **/etc/passwd**, which contains information about every user of the system. **endpwent()** closes the password file. Please note that this function does not return a meaningful value.

Example

For an example of this function, see the entry for **getpwent()**.

Files

/etc/passwd
pwd.h

See Also

getpwent(), getpwnam(), getpwuid(), libc, pwd.h, setpwent()

endservent() — Sockets Function (libsocket)

Close protocols file
#include <netdb.h>
void endservent();

Function **endservent()** closes file **/etc/protocols** which describes the services offered by TCP/IP on your local network. after it had been opened by function **getservent()** or **setservent()**.

See Also

getservbyname(), getservbyport(), getservent(), libsocket, netdb.h, setservent()

endspent() — General Function (libc)

Close the shadow-password file
#include <shadow.h>
endspent()

The COHERENT system has four routines that search the file **/etc/shadow**, which contains the password of every user of your system. **endspent()** closes **/etc/shadow**. It does not return a meaningful value.

Files

/etc/shadow
/usr/include/shadow.h

See Also

getspent(), libc, setspent(), shadow, shadow.h

endutent() — General Function (libc)

Close the login logging file
#include <utmp.h>
void endutent()

Function **endutent()** closes the logging file. Usually this is the system file **/etc/utmp**. The file must have been opened by a call to function **getutent()**, **getutid()**, or **getutline()**.

For a summary of the family of functions that manipulate logging files, see the Lexicon entry for **utmp.h**.

See Also

getutent(), libc, utmp.h

enum — C Keyword

Declare a type and identifiers

An **enum** declaration is a data type whose syntax resembles those of the **struct** and **union** declarations. It lets you enumerate the legal value for a given variable. For example,

```
enum opinion {yes, maybe, no} GUESS;
```

declares type **opinion** can have one of three values: **yes**, **no**, and **maybe**. It also declares the variable **GUESS** to be of type **opinion**.

As with a **struct** or **union** declaration, the tag (**opinion** in this example) is optional; if present, it may be used in subsequent declarations. For example, the statement

```
register enum opinion *op;
```

declares a register pointer to an object of type **opinion**.

All enumerated identifiers must be distinct from all other identifiers in the program. The identifiers act as constants and can be used wherever constants are appropriate.

COHERENT assigns values to the identifiers from left to right, normally beginning with zero and increasing by one. In the above example, the values of **yes**, **no**, and **maybe** are set, respectively, to one, two, and three. The values often are **int**s, although if the range of values is small enough, the **enum** will be an **unsigned char**. If an identifier in the declaration is followed by an equal sign and a constant, the identifier is assigned the given value, and subsequent values increase by one from that value; for example,

```
enum opinion {yes=50, no, maybe} guess;
```

sets the values of the identifiers **yes**, **no**, and **maybe** to 50, 51, and 52, respectively.

See Also

C keywords
ANSI Standard, §6.5.2.2

ENV — Environmental Variable

File read to set environment

Whenever the Korn shell is invoked, it executes the script named in the environmental variable **ENV**. By custom, this is set to **${HOME}/.kshrc**, although you can name any file you wish. This script usually sets aliases and environmental variables, and executes the **set** command to modify the behavior of the shell itself.

By defining **ENV** in your **.profile**, you can ensure that this file is executed whenever you invoke a shell. If you wish to have its definitions read only by the login shell, insert the instruction

```
unset ENV
```

at the end of the script named by **ENV**.

See Also

environmental variables, ksh, .kshrc

env — Command

Execute a command in an environment
env [-] [*VARIABLE***=***value ...***] [***command args***]**

The command **env** executes *command* with *args*, modifying the existing environment by performing the requested assignments.

The '-' option tells **env** to replace the environment with the arguments of the form **VARIABLE=***value*; otherwise the assignments are added to the environment.

If *command* is omitted, the resulting environment is printed.

See Also

commands

environ — C Language

Process environment

extern char **environ;

environ is an array of strings, called the *environment* of a process. By convention, each string has the form

name=value

Normally, each process inherits the environment of its parent process. The shell **sh** and various forms of **exec** can change the environment. The shell adds the name and value of each shell variable marked for *export* to the environment of subsequent commands. The shell adds assignments given on the same line as a command to the environment of the command, without affecting subsequent commands.

See Also

C language, exec, getenv(), Programming COHERENT, putenv(), sh
POSIX Standard, §3.1.2

environmental variables — Technical Information

The *environment* is a set of information that is read by all programs that run on your system. It consists of one or more *environmental variables* that you set. For example, when you set the environmental variable **PATH**, you tell COHERENT that you wish to pass this information to all programs on your system, including COHERENT itself.

By changing the environment, you can change the way a command works without rewriting any commands that you may have embedded in batch files, scripts, or **makefile**s.

Your programs may request environmental variables of their own definition. COHERENT uses the following environmental variables to set its environment. Note that the variables marked with an asterisk are used only by the Korn shell **ksh**.

ASKCC . . . . . Have **mail** prompt for CC names
CWD* . . . . . Current working directory
EDITOR . . . . . Editor used by default by **mail**
ENV* . . . . . File read to set environment
FCEDIT* . . . . . Editor used by the **fc** command
IFS . . . . . Characters recognized as white space
HOME . . . . . User's home directory
KSH_VERSION*. . . . . List current version of Korn shell
LASTERROR* . . . . . Program that last generated an error
LIBPATH. . . . . Directories that hold compiler phases and libraries
LOGNAME . . . . . Name user's identifier
MAIL . . . . . File that holds user's mail messages
MLP_COPIES . . . . . Set default number of copies to print
MLP_FORMLEN. . . . . Set default page length
MLP_LIFE . . . . . Set default life for print jobs
MLP_PRIORITY. . . . . Set default priority for print spooling
MLP_SPOOL. . . . . Pass user-specific information to print spooler
PAGER . . . . . User's preferred output filter
PATH . . . . . Directories that hold executable files
PS1 . . . . . User's default prompt
PS2 . . . . . Prompt when unbalanced quotation marks span a line
SECONDS*. . . . . Number of seconds since current shell started
SHELL . . . . . Name the default shell
TERM. . . . . Name the default terminal type
TIMEZONE. . . . . User's current time zone
TMPDIR . . . . . Directory that holds temporary files

USER . . . . . Name user's identifier

You can also set the following environmental variables to control the default settings of the COHERENT assembler **as**, the C compiler **cc**, and the linker **ld**:

ARHEAD . . . . . Append options to beginning of **ar** command line
ARTAIL. . . . . Append options to end of **ar** command line
ASHEAD . . . . . Append options to beginning of **as** command line
ASTAIL. . . . . Append options to end of **as** command line
CCHEAD . . . . . Append options to beginning of **cc** command line
CCTAIL. . . . . Append options to end of **cc** command line
CPPHEAD . . . . . Append options to beginning of **cpp** command line
CPPTAIL . . . . . Append options to end of **cpp** command line
LDHEAD . . . . . Append options to beginning of **ld** command line
LDTAIL. . . . . Append options to end of **ld** command line

See Also

get_env(), unset, Using COHERENT

Notes

To delete an environmental variable, use the command **unset**.

envp — C Language

Argument passed to main()
char *envp[];

envp is an abbreviation for environmental parameter. It is the traditional name for a pointer to an array of string pointers passed to a C program's **main** function, and is by convention the third argument passed to **main**.

Example

The following example demonstrates **envp**, **argc**, and **argv**.

```
#include <stdio.h>

main(argc, argv, envp)
int argc;                           /* Number of args */
char *argv[];                       /* Argument ptr array */
char *envp[];                       /* Environment ptr array */
{
      int a;

      printf("The command name (argv[0]) is %s\n", argv[0]);
      printf("There are %d arguments:\n", argc-1);
      for (a=1; a<argc; a++)
            printf("\targument %2d:\t%s\n", a, argv[a]);

      printf("The environment is as follows:\n");
      a = 0;
      while (envp[a] != NULL)
            printf("\t%s\n", envp[a++]);
}
```

See Also

argc, argv, C language, environ, main()

EOF — Manifest Constant

Indicate end of a file
#include <stdio.h>

EOF is an indicator that is returned by several STDIO functions to indicate that the current file position is the end of the file.

Many STDIO functions, when they read **EOF**, set the end-of-file indicator that is associated with the stream being read. Before more data can be read from the stream, its end-of-file indicator must be cleared. Resetting the file-position indicator with the functions **fseek**, **fsetpos**, or **ftell** will clear the indicator, as will returning a character to the stream with the function **ungetc**.

See Also

file, manifest constant, stream, stdio.h
ANSI Standard, §7.9.1

epson — Command

Prepare files for Epson printer
epson [-cdfnrw8] [-b *head* **] [-i** *n* **] [-o** *file* **] [-s** *n* **] [** *file ...* **]**

epson prepares text for printing an Epson or Epson-compatible dot-matrix printer. It recognizes the **nroff** output sequences for boldface and italics and converts them into the Epson codes for emphasized print and italics.

If you do not name a file on its command line, **epson** reads the standard input. By default, **epson** writes its output to the standard output. Thus, you can use **epson** as a filter within an MLP backend script.

By default, **epson** outputs the string "\033 @ \033 t \0" at the beginning of each job to initialize the printer. The sequence "\033 @" clears the printer and prepares it to receive new data; while the escape sequence "\033 t \0" makes an Epson printer's built-in italics font available. To suppress the italics-font portion of the initialization sequence, use the command-line option **-n**, described below.

epson recognizes the following command-line options:

-b *head* Print the given *head* as a double-width banner at the top of the first output page.

-c Use compressed printing mode.

-d Print boldface as double strikes. Normally, **epson** recognizes the sequence "*c***\b***c*" as boldface and prints *c* in emphasized printing mode. **-d** is useful in conjunction with **-c**.

-f Do not print a formfeed character at the end of each *file*.

-i*n* Indent *n* spaces at the start of each output line.

-n No italics: suppress the italics portion of the printer-initialization string. When you use this switch, **epson** outputs the string "\033 @" to initialize the printer.

-o *file* Write output into *file*, instead of sending it to device **/dev/lp**.

-r Print all characters in Roman; do not use italics. Normally, **epson** recognizes the sequence "_**\b***c*" as italic and prints *c* in its italic character set.

-s*n* Print *n* newlines at the end of each line. *n* must be 1, 2, or 3; the default is 1.

-w Use double width printing mode.

-8 Print lines with vertical spacing of eight lines per inch instead of the default six lines per inch.

See Also

commands, lp, nroff, pr, printer

Notes

Prior to release 4.2.12 of COHERENT, **epson** wrote its output to device **/dev/lp** instead of to the standard output.

erand48() — Random-Number Function (libc)

Return a 48-bit pseudo-random number as a double
double erand48(*xsubi***)**
unsigned short *xsubi***[3];**

Function **erand48()** generates a 48-bit pseudo-random number, and returns it in the form of a **double**. The value returned is (or should be) uniformly distributed through the range of 0.0 through 1.0. *xsubi* is an array of three unsigned short integers from which the pseudo-random number is built.

See Also

libc, srand48()

errno — Global Variable

External integer for return of error status
#include <errno.h>
extern int errno;

errno is an external integer that COHERENT links into each of its programs. COHERENT sets **errno** to the negative value of any error status returned to any function that performs COHERENT system calls.

Mathematical functions use **errno** to indicate classifications of errors on return. **errno** is defined within the header file **errno.h**. Because not every function uses **errno**, it should be polled only in connection with those functions that document its use and the meaning of the various status values. For the names of the error codes (as defined in **errno.h**, their value, and the message returned by the function **perror**, see **errno.h**.

Example

For an example of using **errno** in a mathematics program, see the entry for **acos**.

See Also

errno.h, libm, perror(), Programming COHERENT, signal()
ANSI Standard, §7.1.4
POSIX Standard, §2.4

errno.h — Header File

Error numbers used by errno()
#include <errno.h>

errno.h is the header file that defines and describes the error numbers returned in the external variable **errno**. The following lists the error numbers defined in **errno.h**:

EPERM: Permission denied
You lack permission to perform the operation you have requested.

ENOENT: No such file or directory
A program could not find a required file or directory.

ESRCH: No such process
You are attempting to communicate with a process that does not exist.

EINTR: Interrupted system call
A COHERENT system call failed because it received a signal or an alarm expired.

EIO: I/O error
A physical I/O error occurred on a device driver. This could be a tape error, a CRC error on a disk, or a framing error on a synchronous HDLC link.

ENXIO: No such device or address
You attempted to access a device that does not exist. It may be that a specified minor device is invalid, or the unit is powered off. This error can also indicate that a block number given to a minor device is out of range. If you attempt to open a pipe in write-only mode, if **O_NDELAY** is set, and if there are currently no readers on this pipe, **open()** returns immediately and sets **errno** to **ENXIO**.

E2BIG: Argument list too long
The number of bytes of arguments passed in an **exec** is too large.

ENOEXEC: exec() format error
The file given to **exec** is not a valid executable module (probably because it does not have the magic number at the beginning), even though its mode indicates that it is executable.

EBADF: Bad file descriptor
You passed a file descriptor to a system call for a file that was not open or was opened in a manner inappropriate to the call. For example, a file descriptor opened only for reading may not be accessed for writing.

ECHILD: No child processes
A process issued a **wait()** call when it had no outstanding children.

EAGAIN: No more processes
The system cannot create any more processes, either because it is out of table space or because the invoking process has reached its quota or processes.

ENOMEM: not enough memory
The system does not have enough memory available to map a process into memory. This occurs in response to a the system calls **exec()** or **brk()**.

EACCES: Permission denied
You do not have permission to perform the requested operation upon a given file.

EFAULT: Bad address
You requested an address that does not lie within the address space. Normally, this generates signal **SIGSYS**, which terminates the process.

ENOTBLK: Block device required
You passed to system calls **mount()** and **umount()** the descriptor of file that is not a block-special device.

EBUSY: Mount device busy
You passed to the system call **mount()** the file descriptor of a device that is already mounted; or you passed to the system call **umount()** the descriptor of a device that has open files or active working directories.

EEXIST: File exists
An attempt was made to **link** to a file that already exists.

EXDEV: Cross-device link
You attempted to link a file on one file system with a file on another. This is not permitted.

ENODEV: No such device
You attempted to manipulate a device that does not exist.

ENOTDIR: Not a directory
You attempted to perform a directory operation upon a file that is not a directory. For example, you passed the file descriptor of a character-special device to system calls **chdir()** or **chroot()**.

EISDIR: Is a directory
You attempted to perform an inappropriate operation upon a directory. For example, you passed the file descriptor of a directory to **write()**.

EINVAL: Invalid argument
An argument to a system call is out of range. For example, you passed to **kill()** or **umount()** the file descriptor of a device that is not mounted.

ENFILE: File table overflow
The COHERENT kernel uses a static table to record which files are open. This error indicates that this table is full. Until a file is closed, thus freeing space on this table, no more files can be opened on your system.

EMFILE: Too many open files
The COHERENT kernel limits the number of files that any one process can have open at any given time; this error indicates that you have exceeded this number. The system call **sysconf()** returns the number of files that a process can open (among other items of information). For details, see its entry in the Lexicon.

ENOTTY: Not a teletypewriter (tty)
You attempted to perform a terminal-specific operation upon a device which is not a terminal.

ETXTBSY: Text file busy
The text segment of a shared load module is unwritable. Therefore, an attempt to execute it while it is being written or an attempt to open it for writing while it is being executed will fail.

EFBIG: File too large
The block-mapping algorithm for a file fails above 1,082,201,088 bytes. Attempting to write a file larger than this will generate this error.

ENOSPC: No space left on device
You attempt to write onto a device that is full. If the attemped write was onto a file system, either the file system's supply of blocks was exhausted, or its supply of i-nodes was exhausted.

ESPIPE: Tried to seek on a pipe
It is illegal to invoke the system call **lseek()** on a pipe.

EROFS: Read-only file system
You attempted to write onto a file system mounted read-only.

EMLINK: Too many links
A file can have no more than 32,767 links. The attempted link operation would exceed this value.

EPIPE: Broken pipe
You attempted to invoke the system call **write()** on a pipe for which there are no readers. This condition is accompanied by the signal **SIGPIPE,** so the error will be seen only if the signal is ignored or caught.

EDOM: Mathematics library domain error
An argument to a mathematical routine falls outside that function's domain.

ERANGE: Mathematics library result too large
The result of a mathematical function is too large to be represented.

ENOMSG: No message of desired type
You invoked **msgrcv()** to read a message of a given type, but none was waiting to be read.

EIDRM: Identifier removed

EDEADLK: Deadlock condition
A process is deadlocked for some reason.

ENOLCK: No record locks available
The maximum number of record locks has been exceeded.

ENOSTR: Device not a stream
You attempted to perform a STREAMS operation on a file that is not a stream.

ENODATA: No data available

ETIME: Timer expired

ENOSR: Out of STREAMS resources

ENOPKG: Package not installed

EPROTO: Protocol error

EBADMSG: Not a data message

ENAMETOOLONG: File name too long

EOVERFLOW: Value too large for defined data type

ENOTUNIQ: Name not unique on network

EBADFD: File descriptor in bad state

EREMCHG: Remote address changed

ELIBACC: Cannot access a needed shared library
COHERENT does not yet support shared libraries.

ELIBBAD: Accessing a corrupted shared library
COHERENT does not yet support shared libraries.

ELIBSCN: **.lib** section in **a.out** corrupted

ELIBMAX: Maximum number of shared libraries exceeded
COHERENT does not yet support shared libraries.

ELIBEXEC: Cannot **exec()** a shared library directly
COHERENT does not yet support shared libraries.

EILSEQ: Illegal byte sequence

ENOSYS: Operation not applicable

ELOOP: Symbolic links error.
Number of symbolic links encountered during path name traversal exceeds **MAXSYMLINKS**. COHERENT does not yet support symbolic links.

EUSERS: Too many users

ENOTSOCK: Socket operation on non-socket

EDESTADDRREQ: Destination address required

EMSGSIZE: Message too long

EPROTOTYPE: Protocol wrong type for socket

ENOPROTOOPT: Protocol not available

EPROTONOSUPPORT: Protocol not supported

ESOCKTNOSUPPORT: Socket type not supported

EOPNOTSUPP: Operation not supported on transport endpoint

EPFNOSUPPORT: Protocol family not supported

EAFNOSUPPORT: Address family not supported by protocol family

EADDRINUSE: Address already in use

EADDRNOTAVAIL: Cannot assign requested address

ENETDOWN: Network is down

ENETUNREACH: Network is unreachable

ENETRESET: Network dropped connection because of reset

ECONNABORTED: Software-caused connection abort

ECONNRESET: Connection reset by peer

ENOBUFS: No buffer space available

EISCONN: Transport endpoint is already connected

ENOTCONN: Transport endpoint is not connected

ESHUTDOWN: Cannot send after transport endpoint shutdown

ETIMEDOUT: Connection timed out

ECONNREFUSED: Connection refused

EHOSTDOWN: Host is down

EHOSTUNREACH: No route to host

EALREADY: Operation already in progress

EINPROGRESS: Operation now in progress

ESTALE: Stale NFS file handle
COHERENT does not yet support nonproprietary file systems.

See Also

errno, header files, perror(), signal()
ANSI Standard, §7.1.3
POSIX Standard, §2.4

eval — Command

Evaluate arguments
eval [*token ...*]

The shell normally evaluates each token of an input line before executing it. During evaluation, the shell performs parameter, command, and file-name pattern substitution. The shell does *not* interpret special characters after performing substitution.

eval is useful when an additional level of evaluation is required. It evaluates its arguments and treats the result as shell input. For example,

```
A='>file'
echo a b c $A
```

simply prints the output

```
a b c >file
```

because '>' has no special meaning after substitution, but

```
A='>file'
eval echo a b c $A
```

redirects the output

```
a b c
```

to **file**. Similarly,

```
A='$B'
B='string'
echo $A
eval echo $A
```

prints

```
$B
string
```

In the first **echo** the shell performs substitution only once.

The shell executes **eval** directly.

See Also

commands, ksh, sh

ex — Command

Berkeley-style line editor
ex [*options* **] [+***cmd* **] [** *file1 ... file27* **]**

ex is a link to **elvis**, which is a clone of the UNIX **vi/ex** set of editors. Invoking **elvis** through this link forces it to operate solely in colon-command mode, just as the UNIX **ex** editor operates.

For information on how to use this version of **ex**, see the Lexicon page for **elvis**.

See Also

commands, ed, elvis, me, vi, view

Notes

exec — Command

Execute command directly
exec [*command*]

The shell normally executes commands through the system call **fork()**, which creates a new process. The shell command **exec** directly executes the given *command* through one of the **exec()** functions instead. Normally, this terminates execution of the current shell.

If the *command* consists only of redirection specifications, **exec** redirects the input or output of the current shell accordingly without terminating it. If the *command* is omitted, **exec** has no effect.

See Also

commands, execution, fork(), ksh, sh, xargs
POSIX Standard, §3.1.2

execl() — General Function (libc) (libc)

Execute a load module
#include <unistd.h>
execl(*file, arg0, arg1, ..., argn,* **NULL**)
char **file, *arg0, *arg1, ..., *argn***;**

The function **execl()** calls the COHERENT system call **execve()** to execute a program. It specifies arguments individually, as a NULL-terminated list of *arg* parameters. For more information on file execution, see **execution**.

See Also

execution, execve(), getuid(), libc, unistd.h
POSIX Standard, §3.1.2

Diagnostics

execl() does not return if successful. It returns -1 for errors, such as *file* being nonexistent, not accessible with execute permission, having a bad format, or too large to fit in memory.

execle() — General Function (libc) (libc)

Execute a load module
#include <unistd.h>
execle(*file, arg0, arg1, ..., argn,* **NULL,** *env*)
char **file, *arg0, *arg1, ..., *argn***, char** **env*[];**

The function **execle()** calls the COHERENT system call **execve()** to execute a program. It first initializes the new stack of the process to contain a list of strings that are command arguments. It specifies arguments individually, as a NULL-terminated list of *arg* parameters. The argument *envp* points to an array of pointers to strings that define *file*'s environment. For more information on program execution and environments, see **execution**.

See Also

environ, execution, execve(), libc, unistd.h
POSIX Standard, §3.1.2

Diagnostics

execle() does not return if successful. It returns -1 for errors, such as *file* being nonexistent, not accessible with execute permission, having a bad format, or being too large to fit into memory.

execlp() — General Function (libc)

Execute a load module
#include <unistd.h>
execlp(*file, arg0, arg1, ..., argn,* **NULL)**
char **file, *arg0, *arg1, ..., *argn***;**

The function **execlp()** calls the COHERENT system call **execve()** to execute a program. It initializes the new stack of the process to contain a list of strings that are command arguments. It specifies arguments individually, as a NULL-terminated list of *arg* parameters. Unlike the related function **execl()**, **execlp()** searches for *file* in all directories named in the environmental variable **PATH**. For more information on program execution, see **execution**.

See Also

environ, execution, execve(), libc, unistd.h
POSIX Standard, §3.1.2

Diagnostics

execlp() does not return if successful. It returns -1 for errors, such as *file* not existing in the directories named in **PATH**, not accessible with execute permission, having a bad format, or too large to fit in memory.

execlpe() — General Function (libc)

Execute a load module
execlpe(*file, arg0, arg1, ..., argn,* **NULL,** *env***)**
char **file, *arg0, *arg1, ..., *argn***;**
char **env***[];*

The function **execlpe()** calls the COHERENT system call **execve()** to execute a program. It initializes the new stack of the process to contain a list of strings that are command arguments. It specifies arguments individually, as a NULL-terminated list of *arg* parameters.

The argument *env* points to an array of pointers to strings that define *file*'s environment.

Unlike the related function **execl()**, **execlpe()** searches for *file* in all directories named in the environmental variable **PATH**— that is, the current **PATH**, not the one contained within the environmented pointed to by *env*.

For more information on program execution, see **execution**.

See Also

environ, execution, execl(), execvep(), libc

Diagnostics

execlpe() does not return if successful. It returns -1 for errors, such as *file* not existing in the directories named in **PATH**, not accessible with execute permission, having a bad format, or too large to fit in memory.

execlpe() is not part of the SVID specification. Therefore, it may not be present on non-COHERENT operating systems.

execution — Definition

Program execution under COHERENT is governed by the various forms of the COHERENT system call **exec()**. This call allows a process to execute another executable *file* (or load module). This is described in **coff.h**.

The code, data and stack of *file* replace those of the requesting process. The new stack contains the command arguments and its environment, in the format given below. Execution starts at the entry point of *file*.

During a successful call to **exec()**, the system deactivates profiling, and resets any caught signals to **SIG_DFL.**

Every process has a real-user id, an effective-user id, a saved-effective user id; and a real-group id, an effective-group id, and a saved-effective group id. These identifiers are defined in the Lexicon entries for, respectively, **setuid()** and **setgid()**. For most load modules, **exec()** does not change any of these. However, if the *file* is marked with the set user id or set group id bit (see **stat()**), **exec()** sets the effective-user id (effective-group id) of the process to the user id (group id) of the *file* owner. In effect, this changes the file access privilege level from that of the real id to that of the effective id. The owner of *file* should be careful to limit its abilities, to avoid compromising file security.

exec() initializes the new stack of the process to contain a list of strings, which are command arguments. **execl()**, **execle()**, **execlp()**, and **execlpe()** specify arguments individually, as a NULL-terminated list of *arg* parameters. **execv()**, **execve()**, **execvp()**, and **execvpe()** specify arguments as a single NULL-terminated array **argv** of parameters.

The **main()** function of a C program is invoked in the following way:

```
main(argc, argv, envp)
int argc;
char *argv[], *envp[];
```

argc is the number of command arguments passed through **exec()**, and **argv** is an array of the actual argument strings. **envp** is an array of strings that comprise the process environment. By convention, these strings are of the form *variable=value*, as described in the Lexicon entry **environ**. Typically, each *variable* is an **export**ed shell

variable with the given *value*.

execl() and **execv()** simply pass the old environment, referenced by the external pointer **environ**.

execle(), **execlpe()**, **execve()**, and **execvpe()** pass a new environment *env* explicitly.

execlp(), **execlpe()**, **execvp()**, and **execvpe()** search for *file* in each of the directories indicated by the shell variable **$PATH**, in the same way that the shell searches for a command. These calls execute a shell command *file*. Note that **execlpe()** and **execvpe()** search the current **PATH**, not the **PATH** contained within the environment pointed to by *env*.

Files

/bin/sh — To execute command files

See Also

environ, exec(), execl(), execle(), execlp(), execlpe(), execv(), execve(), execvp(), execvpe(), fork(), ioctl(), Programming COHERENT, signal(), stat(), xargs

Diagnostics

None of the **exec()** routines returns if successful. Each returns -1 for an error, such as if *file* does not exist, is not accessible with execute permission, has a bad format, or is too large to fit in memory.

Notes

Each **exec()** routine now examines the beginning of an executable file for the token **#!**. If found, it invokes the program named on that line and passes it the rest of the file. For example, if you wish to ensure that a given script is executed by the by the Bourne shell **/bin/sh**, begin the script with the line:

```
#!/bin/sh
```

execv() — General Function (libc)

Execute a load module
#include <unistd.h>
execv(*file, argv***)**
char **file, *argv***[];**

The function **execv()** calls the COHERENT system call **execve()** to execute a program. It specifies arguments as a single, NULL-terminated array of parameters, called *argv*. **execv()** passes the environment of the calling program to the called program. For more information on program execution, see **execution**.

See Also

environ, execution, execve(), libc, unistd.h
POSIX Standard, §3.1.2

Diagnostics

execv() does not return if successful. It returns -1 for errors, such as *file* being nonexistent, not accessible with execute permission, having a bad format, or too large to fit in memory.

execve() — System Call (libc)

Execute a load module
#include <unistd.h>
execve(*file, argv, env***)**
char **file, *argv***[], ***env***[];**

The function **execve()** executes a program. It specifies arguments as a single, NULL-terminated array of parameters, called *argv*. The argument *env* is the address of an array of pointers to strings that define *file*'s environment. This allows **execve()** to pass a new environment to the program being executed. For more information on program execution, see **execution**.

Example

The following example demonstrates **execve()**, as well as **tmpnam()**, **getenv()**, and **path()**. It finds all lines with more than **LIMIT** characters and calls MicroEMACS to edit them.

```
#include <stdio.h>
#include <path.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>

#define LIMIT 70

extern **environ, *tempnam();

main(argc, argv)
int argc; char *argv[];
{
      /*               me     -e   tmp   file */
      char *cmda[5] = { NULL, "-e", NULL, NULL, NULL };
      FILE *ifp, *tmp;
      char line[256];
      int  ct, len;

      if ((NULL == (cmda[3] = argv[1])) ||
          (NULL == (ifp = fopen(argv[1], "r")))) {
            fprintf(stderr, "Cannot open %s\n", argv[1]);
            exit(EXIT_FAILURE);
      }

      if ((cmda[0] = path(getenv("PATH"), "me", X_OK)) == NULL) {
            fprintf(stderr, "Cannot locate me\n");
            exit(EXIT_FAILURE);
      }

      if (NULL == (tmp = fopen((cmda[2] = tempnam(NULL, "lng")), "w"))) {
            fprintf(stderr, "Cannot open tmpfile\n");
            exit(EXIT_FAILURE);
      }

      for (ct = 1; NULL != fgets(line, sizeof(line), ifp); ct++)
            if (((len = strlen(line)) > LIMIT) ||
                ('\n' != line[len -1]))
                  fprintf(tmp, "%d: %d characters long\n", ct, len);

      fclose(tmp);
      fclose(ifp);

      if (execve(cmda[0], cmda, environ) < 0) {
            fprintf(stderr, "cannot execute me\n");
            exit(EXIT_FAILURE);
      }
}
```

See Also

environ, execution, libc, unistd.h
POSIX Standard, §3.1.2

Diagnostics

execve() does not return if successful. It returns -1 for errors, such as *file* being nonexistent, not accessible with execute permission, having a bad format, or too large to fit in memory.

execvp() — General Function (libc)

Execute a load module
#include <unistd.h>
execvp(*file, argv***)**
char **file, *argv***[];**

The function **execvp()** calls the COHERENT system call **execve()** to execute a program. It specifies arguments as a single, NULL-terminated array of parameters, called *argv*. Unlike the related call **execv()**, **execvp()** searches for *file* in all of the directories named in the environmental variable **PATH**. For more information on program execution, see **execution**.

See Also

environ, execution, execve(), libc, unistd.h
POSIX Standard, §3.1.2

Diagnostics

execvp() does not return if successful. It returns -1 for errors, such as *file* being nonexistent, not accessible with execute permission, having a bad format, or too large to fit in memory.

execvpe() — General Function (libc)

Execute a load module
execvp(*file, argv***,** *env***)**
char **file, *argv***[],* ***env***[];*

The function **execvpe()** calls the COHERENT system call **execve()** to execute a program. It specifies arguments as a single, NULL-terminated array of parameters, called *argv*. The argument *env* is the address of an array of pointers to strings that define *file*'s environment. This allows **execvpe()** to pass a new environment to the program being executed.

Unlike the related call **execv()**, **execvpe()** searches for *file* in all of the directories named in the environmental variable **PATH**— that is, the current **PATH**, not the one contained within the environmented pointed to by *env*.

For more information on program execution, see **execution**.

See Also

environ, execution, execv(), execve(), libc

Diagnostics

execvp() does not return if successful. It returns -1 for errors, such as *file* being nonexistent, not accessible with execute permission, having a bad format, or too large to fit in memory.

execvpe() is not part of the SVID specification. Therefore, it may not be present on non-COHERENT operating systems.

exit — Command

Exit from a shell
exit [*status***]**

exit terminates a shell. If the optional *status* is specified, the shell returns it; otherwise, the previous status is unchanged. From an interactive shell, **exit** sets the *status* if specified, but does not terminate the shell. The shell executes **exit** directly.

See Also

commands, ksh, sh

exit() — General Function (libc)

Terminate a program gracefully
#include <stdlib.h>
void exit(*status***) int** *status***;**

The library function **exit()** is the normal method to terminate a program directly. *status* information is passed to the parent process. By convention, an exit status of zero indicates success, whereas an exit status greater than zero indicates failure. If the parent process issued a **wait()** call, it is notified of the termination and is passed the least significant eight bits of *status*. As **exit()** never returns, it is always successful. Unlike the system call **_exit()**, **exit()** does extra cleanup, such as flushing buffered files and closing open files.

Example

For an example of this function, see the entry for **fopen()**.

See Also

_exit(), atexit(), close(), EXIT_FAILURE, EXIT_SUCCESS, libc, stdlib.h, wait()
ANSI Standard, §7.10.4.3
POSIX Standard, §8.1

Notes

If you do not explicitly set *status* to a value, the program returns whatever value happens to have been in the register EAX. You can set *status* to either **EXIT_SUCCESS** or **EXIT_FAILURE**.

EXIT_FAILURE — Manifest Constant

Indicate program failed to execute successfully
#include <stdlib.h>

EXIT_FAILURE is a manifest constant that is defined in the header **stdlib.h**. It is used as an argument to the function **exit()** to indicate that the program failed to execute successfully.

See Also

exit(), manifest constant, stdlib.h
ANSI Standard, §7.10.4.3

EXIT_SUCCESS — Manifest Constant

Indicate program executed successfully
#include <stdlib.h>

EXIT_SUCCESS is a manifest constant that is defined in the header **stdlib.h**. It is used as an argument to the function **exit()**, to indicate that the program executed successfully.

See Also

exit(), manifest constant, stdlib.h
ANSI Standard, §7.10.4.3

exp() — Mathematics Function (libm)

Compute exponent
#include <math.h>
double exp(*z***) double** *z***;**

exp() returns the exponential of *z*, or *e*^*z*.

Example

The following example, called **apr.c**, computes the annual percentage rate (APR) for a given rate of interest. Compile it with the command:

```
cc -f apr.c -lm
```

It is by Brent Seidel (brent_seidel@chthone.stat.com):

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
      double rate, APR;
      char buffer[50];

      printf("Enter interest rate in percent (e.g., 12.9): ");
      fflush(stdout);

      if (gets(buffer) == NULL)
            exit(EXIT_FAILURE);
      rate = strtod(buffer);

      APR = (exp(rate/100.0) - 1) * 100.0;
      printf("The APR for %g%% compounded daily is %g%%\n", rate, APR);
}
```

See Also

errno, frexp(), ldexp(), libm
ANSI Standard, §7.5.4.1
POSIX Standard, §8.1

Diagnostics

exp() indicates overflow by an **errno** of **ERANGE** and a huge returned value.

export — Command

Add a shell variable to the environment
export [*name ...***]**
export [*name***=***value***]**

When the shell executes a command, it passes the command an *environment*. By convention, the environment consists of assignments, each of the form *name=value*. For example, typing

```
export TERM=vt100
```

sets the environmental variable **TERM** to equal the string **vt100**.

A command may look for information in the environment or may simply ignore it. In the above example, a program that reads the variable **TERM** (such as COHERENT) will assume that you are working on a DEC VT-100 terminal or one that emulates it.

The shell places the *name* and the value of each shell variable that appears in an **export** command into the environment of subsequently executed commands. It does not place a shell variable into the environment until it appears in an **export** command.

With no arguments, **export** prints the name and the value of each shell variable currently marked for export.

The shell executes **export** directly.

See Also

commands, environ, exec, ksh, sh

expr — Command

Compute a command-line expression
expr *argument ...*

The arguments to **expr** form an expression. **expr** evaluates the expression and writes the result on the standard output. Among other uses, **expr** lets the user perform arithmetic in shell command files.

Each *argument* is a separate token in the expression. An argument has a logical value 'false' if it is a null string or has numerical value zero, 'true' otherwise. Integer arguments consist of an optional sign followed by a string of decimal digits. The range of valid integers is that of signed long integers. No check is made for overflow or illegal arithmetic operations. Floating point numbers are not supported.

The following list gives each **expr** operator and its meaning. The list is in order of increasing operator precedence; operators of the same precedence are grouped together. All operators associate left to right except the unary operators '!', '-', and '**len**', which associate right to left. The spaces shown are significant - they separate the tokens of the expression.

{ *expr1***,** *expr2***,** *expr3* **}**
Return *expr2* if *expr1* is logically true, and *expr3* otherwise. Alternatively, **{** *expr1* **,** *expr2* **}** is equivalent to **{** *expr1* **,** *expr2* **, 0 }**.

expr1 **|** *expr2*
Return *expr1* if it is true, *expr2* otherwise.

expr1 **&** *expr2*
Return *expr1* if both are true, zero otherwise.

expr1 relation expr2
Where *relation* is one of <, <=, >, >=, ==, or !=, return one if the *relation* is true, zero otherwise. The comparison is numeric if both arguments can be interpreted as numbers, lexicographic otherwise. The lexicographic comparison is the same as **strcmp** (see **string**).

expr1 **+** *expr2*

expr1 **-** *expr2*
Add or subtract the integer arguments. The expression is invalid if either *expr* is not a number.

expr1 ***** *expr2*

expr1 **/** *expr2*

expr1 **%** *expr2*
Multiply, divide, or take remainder of the arguments. The expression is invalid if either *expr* is not numeric.

expr1 **:** *expr2*
Match patterns (regular expressions). *expr2* specifies a pattern in the syntax used by **ed**. It is compared to *expr1*, which may be any string. If the \(...\) pattern occurs in the regular expression the matching operator returns the matched field from the string; if there is more than one \(...\) pattern the extracted fields are concatenated in the result. Otherwise, the matching operator returns the number of characters matched.

len *expr*
Return the length of *expr*. It behaves like **strlen** (see **string**). *len* is a reserved word in **expr**.

!*expr* Perform logical negation: return zero if *expr* is true, one otherwise.

-*expr* Unary minus: return the negative of its integer argument. If the argument is non-numeric the expression is invalid.

(*expr* **)**
Return the *expr*. The parentheses allow grouping expressions in any desired way.

The following operators have special meanings to the shell **sh**, and must be quoted to be interpreted correctly: **{ } ( ) < > & | ***.

See Also

commands, ed, ksh, sh, test

Notes

expr returns zero if the expression is true, one if false, and two if an error occurs. In the latter case an error message is also printed.

extern — C Keyword

Declare storage class

extern indicates that a C element belongs to the *external* storage class. Both variables and functions may be declared to be **extern**. Use of this keyword tells the C compiler that the variable or function is defined outside of the present file of source code. All functions and variables defined outside of functions are implicitly **extern** unless declared **static**.

When a source file references data that are defined in another file, it must declare the data to be **extern**, or the linker will return an error message of the form:

```
undefined symbol name
```

For example, the following declares the array **tzname**:

```
extern char tzname[2][32];
```

When a function calls a function that is defined in another source file or in a library, it should declare the function to be **extern**. In the absence of a declaration, **extern** functions are assumed to return **int**s, which may cause serious problems if the function actually returns a 32-bit pointer (such as on the 68000 or i8086 LARGE model), a **long**, or a **double**.

For example, the function **malloc** appears in a library and returns a pointer; therefore, it should be declared as follows:

```
extern char *malloc();
```

If you do not do so, the compiler assumes that **malloc** returns an **int**, and generate the error message

```
integer pointer pun
```

when you attempt to use **malloc** in your program.

See Also

auto, C keywords, pun, register, static, storage class
ANSI Standard, §6.5.1

fabs() — Mathematics Function (libm)

Compute absolute value
#include <math.h>
double fabs(*z***) double** *z***;**

fabs() implements the absolute value function. It returns *z* if *z* is zero or positive, or -*z* if *z* is negative.

Example

For an example of this function, see the entry for **ceil()**.

See Also

abs(), ceil(), floor(), frexp(), libm
ANSI Standard, §7.5.6.2
POSIX Standard, §8.1

factor — Command

Factor a number
factor [*number ...* **]**

factor computes and prints the prime factorials for each of a list of given *number*s. If no *number*s are given on the command line, **factor** reads numbers from the standard input.

See Also

commands

false — Command

Unconditional failure
false

false does nothing. It is guaranteed to fail. It can be useful in shell scripts, to force certain situations to occur.

See Also

commands, ksh, sh, true

Notes

Under the Korn shell, **false** is an alias for its built-in command **let**.

fc — Command

Edit and re-execute one or more previous commands
fc [-e *editor***] [-ln] [***first* **[***last***]]**

fc, the "fix command", is a command built into the Korn shell **ksh**. It permits you to edit and re-execute one or more commands that have been executed previously.

fc selects commands *first* through *last* and inserts them into a text editor. You can edit the commands in the editor; exiting from the editor re-executes the commands. *first* and *last* can be addressed either by the command's number (the first command issued to the shell is number one, the second is number two, and so on), or by a string that matches the beginning of the command. When called without a *last* variable, the command selects just *first*. Option **-l** prints the commands on the standard output rather than buffering the commands for editing and re-execution. Option **-n** suppresses the default command numbers.

fc uses the editor named in the environmental variable **FCEDIT**; if this variable is not set, it uses MicroEMACS. The option **-e** lets you select another editor.

See Also

commands, FCEDIT, ksh

FCEDIT — Environmental Variable

Editor used by fc command

The Korn shell's command **fc** reads the environmental variable **FCEDIT** to see which editor it should use to edit commands.

See Also

environmental variables, ksh

fclose() — STDIO Function (libc)

Close a stream
#include <stdio.h>
int fclose(*fp***) FILE** ***fp*;**

fclose() closes the stream **fp**. It calls **fflush()** on the given *fp*, closes the associated file, and releases any allocated buffer. The function **exit()** calls **fclose()** for open streams.

Example

For examples of how to use this function, see the entries for **fopen()** and **fseek()**.

See Also

libc
ANSI Standard, §7.9.5.1
POSIX Standard, §8.1

Diagnostics

fclose() returns **EOF** if an error occurs.

fcntl() — System Call (libc)

Control open files
#include <fcntl.h>
int fcntl(*fd***,** *command***,** *arg***)**
int *fd***,** *cmd***,** *arg***;**

The COHERENT system call **fcntl()** manipulates an open file.

fd is the file descriptor; this description must have been obtained from a call to **creat()**, **dup()**, **fcntl()**, **open()**, or **pipe()**.

command identifies the task that you want **fcntl()** to perform. The value **fcntl()** returns varies, depending on what command you ask it to perform. *arg* is an argument specific to the given *command*.

fcntl() commands **F_GETLK**, **F_SETLK**, and **F_SETLKW** (described in detail below) implement file-record locking. File-record locks use the **flock** structure, which is defined in header file **<fcntl.h>** as follows:

```
typedef struct flock {
        short           l_type;                 /* F_RDLCK, F_WRLCK, or F_UNLCK*/
        short           l_whence;               /* SEEK_SET, SEEK_CUR, SEEK_END */
        long            l_start;                /* location */
        long            l_len;                  /* 0 is through EOF */
        short           l_sysid;                /* system id of lock (for GETLK) */
        short           l_pid;                  /* process id of owner (for GETLK) */
};
```

You can lock a section of a file for reading (excluding subsequent write locks) or for writing (excluding all subsequent locks). The locked section begins at the specified location **l_start** and can extend backwards (when **l_len** is negative) or forwards (when it is positive). If **l_len** is zero, the lock extends to the end of the file. A lock may extend past the current end of file, but may not extend to before the beginning of the file.

fcntl() Commands

fcntl() recognizes the following commands:

F_DUPFD Duplicate file descriptor *fd* onto the first available file descriptor greater than or equal to *arg*. **fcntl()** returns the new file descriptor.

F_GETFD Get the current value of the close-on-exec flag **FD_CLOEXEC** for the file. If the low-order bit of the return value of **fcntl()** is zero, the file descriptor remains open if the process uses **exec()** to execute another process, If the low-order bit of the return value is one, the file descriptor is closed upon **exec()**.

F_GETFL Get the file flags for the file specified by *fd*. With this option, **fcntl()** returns the file flags.

F_GETLK *arg* must point to a **struct flock** that describes a section of the file to lock. If the system does not have any locks on the specified section, **fcntl()** sets the lock type of *arg* to **F_UNLCK** and leaves the other members unchanged. Otherwise, it sets the contents of *arg* to the first existing lock that blocks the requested lock.

F_SETFD Set the close-on-exec flag of the file to the value of the low bit of *arg*.

F_SETFL Set file flags for file descriptor *fd* to the value specified by *arg*. Here, **fcntl()** returns the new file flags.

F_SETLK Set or clear a file-record lock. *arg* must point to a **struct flock**. Set member **l_type** to **F_RDLCK** to request a read lock, to **F_WRLCK** to request a write lock, or to **F_UNLCK** to unlock a previously locked section. If the requested lock cannot be set, **fcntl()** returns with an error value of -1 and sets **errno** to **EACCES**.

F_SETLKW is just like **F_SETLK** unless the requested lock is not available, in which case **F_SETLKW** causes the current process to sleep until the requested lock becomes available. If sleeping would cause a deadlock, **fcntl()** returns -1 and sets **errno** to **EDEADLK**.

Upon failure, each *cmd* returns -1 and sets **errno** to an appropriate value. Possible **errno** values include the following:

EAGAIN Section already locked.

EBADF Bad file desciptor.

EINVAL Invalid command.

EMFILE Too many files open.

ENOLCK No more locks available.

EDEADLK Deadlock would result.

See Also

close(), creat(), dup(), exec(), fcntl.h, file, file descriptor, libc, lockf(), open(), pipe()
POSIX Standard §6.5.2

Notes

Use **fcntl()** with the unbuffered I/O routines (**open()**, **write()**, and so on) rather than with standard I/O library routines (**fopen()**, **fprintf()**, **fwrite()**, and so on). The buffering used by the standard I/O library may cause unexpected behavior with file locking.

fcntl.h — Header File

Manifest constants for file-handling functions
#include <fcntl.h>

fcntl.h declares manifest constants that are used by the file-handling functions **open()**, **creat()**, and **fcntl()**.

See Also

header files
POSIX Standard, §6.5.1

fcvt() — General Function (libc)

Convert floating-point numbers to strings
char *
fcvt(*d***,** *w***,** ******dp***,** ******signp***)**
double *d***; int** *w***,** ******dp***,** ******signp***;**

fcvt() converts floating point numbers to ASCII strings. Its operation resembles that of **printf()**'s operator **%f**.

fcvt() converts *d* into a NUL-terminated string of decimal digits. The argument *w* sets the precision of the string, i.e., the number of characters to the right of the decimal point.

fcvt() rounds the last digit and returns a pointer to the result. On return, **fcvt()** sets *dp* to point to an integer that indicates the location of the decimal point relative to the beginning of the string: to the right if positive, and to the left if negative. Finally, it sets *signp* to point to an integer that indicates the sign of *d*: zero if positive, and nonzero if negative. **fcvt()** rounds the result to the FORTRAN F-format.

Example

For an example of this function, see the entry for **ecvt()**.

See Also

libc

Notes

fcvt() performs conversions within static string buffers that it overwrites on each execution.

fd — Device Driver

Floppy disk driver

The files **/dev/f*** and **/dev/rf*** are entries for the floppy-disk driver **fd**. Each entry is assigned major device number 4, is accessed as a block-special device, and has a corresponding character-special device entry. **fd** handles up to four 5.25-inch floppy-disk drives, each in one of several formats.

The least-significant four bits of an entry's minor device number identify the type of drive. The next least-significant two bits identify the drive.

The following table summarizes the name, minor device number, sectors per track, partition sector size, characteristics, and addressing method for each device entry of floppy-disk drive 0.

9 sectors/track					
fqa0	13	9	1440	DSQD	cylinder (3.25 inch — 720K)
f9a0	12	9	720	DSDD	cylinder (5.25 inch — 360K)
15 sectors/track					
fha0	14	15	2400	DSHD	cylinder (5.25 inch — 1.2MB)
18 sectors/track					
fva0	15	18	2880	DSHD	cylinder (3.5 inch — 1.44MB

Prefixing an **r** to a device name given above gives the name of the corresponding character-device entry. Corresponding device entries for drives 1, 2, and 3 have minor numbers with offsets of 16, 32, and 48 from the minor numbers given above, and have 1, 2, or 3 in place of 0 in the names given above.

For device entries whose minor number's fourth least-significant bit is zero (minor numbers 0 through 7 for drive 0), the driver uses surface addressing rather than cylinder addressing. This means that it increments tracks before heads when computing sector addresses and the first surface is used completely before the second surface is accessed. For devices whose minor number's fourth least significant bit is 1 (minor numbers 8 through 15 for drive 0), the driver uses cylinder addressing.

For a floppy disk to be accessible from the COHERENT system, a device file must be present in directory **/dev** with the appropriate type, major and minor device numbers, and permissions. The command **mknod** creates a special file for a device.

The following table gives the all floppy-disk devices that COHERENT recognizes, by minor number. Note that some specialized devices skip the first cylinder on the disk, to support some third-party program that requires this feature:

Minor Number	*Drive*	*Diameter*	*Density*	*Cylinders*
0	0	Both	Any	1-39/79
1	0	Both	Any	0-39/79
4	0	5.25”	360KB	1-39
5	0	3.5”	720KB	1-79
6	0	5.25”	1.2MB	1-79
7	0	3.5”	1.44MB	1-79
12	0	5.25”	360KB	0-39
13	0	3.5”	720KB	0-79
14	0	5.25”	1.2MB	0-79
15	0	3.5”	1.44MB	0-79
16	1	Both	Any	1-39/79
17	1	Both	Any	0-39/79
20	1	5.25”	360KB	1-39
21	1	3.5”	720KB	1-79
22	1	5.25”	1.2MB	1-79
23	1	3.5”	1.44MB	1-79
28	1	5.25”	360KB	0-39
29	1	3.5”	720KB	0-79
30	1	5.25”	1.2MB	0-79
31	1	3.5”	1.44MB	0-79

Example

The following program examines a COHERENT floppy-disk device and prints its size in bytes. It was written by Sanjay Lal (sanjayl@tor.comm.mot.com):

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

#define BLOCK 512

struct FDATA {
      int fd_size; /* Blocks per diskette */
      int fd_nhds; /* Heads per drive */
      int fd_trks; /* Tracks per side */
      int fd_offs; /* Sector base */
      int fd_nspt; /* Sectors per track */
      char fd_GPL[4]; /* Controller gap param (indexed by rate) */
      char fd_N; /* Controller size param */
      char fd_FGPL; /* Format gap length */
};

/* Parameters for each kind of format */
struct FDATA fdata [] = {
/* 8 sectors per track, surface by surface seek. */
      {  320, 1, 40, 0, 8, { 0x00, 0x20, 0x20 }, 2, 0x58 }, /* Single sided */
      {  640, 2, 40, 0, 8, { 0x00, 0x20, 0x20 }, 2, 0x58 }, /* Double sided */
      { 1280, 2, 80, 0, 8, { 0x00, 0x20, 0x20 }, 2, 0x58 }, /* Quad density */

/* 9 sectors per track, surface by surface seek. */
      {  360, 1, 40, 0, 9, { 0x00, 0x20, 0x20 }, 2, 0x50 }, /* Single sided */
      {  720, 2, 40, 0, 9, { 0x00, 0x20, 0x20 }, 2, 0x50 }, /* Double sided */
      { 1440, 2, 80, 0, 9, { 0x00, 0x20, 0x20 }, 2, 0x50 }, /* Quad density */

/* 15 sectors per track, surface by surface seek. */
      { 2400, 2, 80, 0, 15, { 0x1B, 0x00, 0x00 }, 2, 0x54 }, /* High capacity */

/* 18 sectors per track, surface by surface seek. */
      { 2880, 2, 80, 0, 18, { 0x1B, 0x00, 0x00 }, 2, 0x6C }/* 1.44 3.5" */
};
```

```
#define funit(x) (minor(x) >> 4)   /* Unit/drive number */
#define fkind(x) (minor(x) & 0x7)  /* Kind of format */

static int ctrl;

int main(argc, argv)
int argc; char **argv;
{
      int size;
      struct stat sbuf;
      struct FDATA *fdp;

      if (argc!=2) {
            fprintf(stderr, "usage : %s /dev/fd...\n",argv[0]);
            exit(EXIT_FAILURE);
      }

      if (strcmp(argv[1], "conv")==0) {
            /*special case*/
            size = getchar() + getchar() * 256;
            printf("%ld\n", (long)((long)size * (long)512) );
            return (EXIT_SUCCESS);
      }

      if (ctrl = stat(argv[1], &sbuf)) {
            fprintf (stderr,"%s : error stating %s.\n", argv[0], argv[1]);
            exit(EXIT_FAILURE);
      }

      fdp = & fdata [fkind (sbuf.st_rdev)];
      printf("%ld\n", (long)((long)fdp->fd_size * (long)512) );

      return (EXIT_SUCCESS);
}
```

Files

<fdioctl.h> — Driver command header file
/dev/fd* — Block-special files
/dev/rfd* — Character special files

See Also

device drivers, fdformat, floppy disk, ft, mkfs, mknod

Diagnostics

The driver reports any error status received from the controller and retries the operation several times before it reports an error to the program that initiated an operation.

Notes

The floppy-tape driver **ft** also works through major-device number 4.

fd assumes that the disk is formatted with eight, nine, 15, or 18 sectors of 512 bytes each per track, depending upon the **/dev** entry. Cylinder addressing is the norm for COHERENT.

Programs that use the raw device interface must read whole sectors into buffers that do not straddle DMA boundaries.

fd.h — Header File

Declare file-descriptor structure
#include <sys/fd.h>

fd.h declares the file-descriptor structure **fd**, plus associated constants.

See Also

header files

fdformat — Command

Low-level format a floppy disk
/etc/fdformat [*option ...* **]** *special*

fdformat formats a floppy disk. The given *special* should be the name of the special file that correspond to the floppy disk drive.

fdformat recognizes the following options:

-a Print information on the standard output device during format. As it formats a cylinder, it will print a line of the form

```
hd=0 cyl=25
```

on your screen.

-i *number*
Use *number* (0 through 7) as the interleave factor in formatting. Note that the default interleave is six.

-o *number*
Use *number* (default, 0) as the skew factor for sector numbering.

-v Verify formatting and verify data written with the **-w** option.

-w *file* Format the floppy disk and then copy *file* to it track by track. The raw device should be used.

The command **mkfs** builds a COHERENT file system on a formatted floppy disk. The command **dosformat** builds a DOS file system on a formatted floppy disk. The command **mount** mounts a floppy disk containing a file system to allow access to it through the COHERENT directory structure. The command **umount** unmounts a floppy disk.

Examples

The following command formats a 2880-block (1.44-megabyte), 3.5-inch floppy disk in drive 1 (otherwise known as drive B):

```
/etc/fdformat -v /dev/rfva1
```

The following command formats a 2400-block (1.2-megabyte), 5.25-inch floppy disk in drive 0 (otherwise known known as drive A):

```
/etc/fdformat -v /dev/rfha0
```

Note that using the raw device (**/dev/rfha0**) speeds up formatting noticeably.

See Also

commands, dosformat, fd, mkfs, mount, umount

Diagnostics

When errors occur on floppy-disk devices the driver prints on the system console an error message that describes the error.

Notes

fdformat formats a track at a time. **fdformat** can be interrupted between tracks, which may result in a partially formatted floppy disk.

fdioctl.h — Header File

Control floppy-disk I/O
#include <sys/fdioctl.h>

fdioctl.h declares constants and structures used to control floppy-disk I/O.

See Also

header files

fdisk — Command

Hard-disk partitioning utility
/etc/fdisk [-r] [-c] [-b *mboot***]** *xdev* ...

The command **fdisk** lets you view how a hard disk is partitioned, alter how it is partitioned, and mark a partition so that the COHERENT bootstrap will automatically boot the operating system it contains. If you wish, you can use **fdisk** to assign partitions to different operating systems, e.g., MS-DOS, CP/M, Windows NT, COHERENT, and XENIX.

fdisk recognizes the following command-line options:

-b Use the first 446 bytes of the file *mboot* to replace the bootstrap information in *xdev*. Use this option to overwrite the COHERENT bootstrap with another bootstrap.

-c Specify the disk geometry (i.e., number of cylinders, heads, sectors) for disk drives that your system's BIOS does not support.

-r Read-only access. **fdisk** reads the partition table and displays its contents, but does not let you change how a disk is partitioned. This is the "safe" option.

-V Display the version number of **fdisk**. PP When you invoke **fdisk**, it reads the first block from the special device *xdev*, which holds the partitioning information for that disk. *xdev* is the device whose name ends in **x**; for example, if you have one SCSI hard disk and one AT-style hard disk installed in your machine, *xdev* would be either **/dev/sd0x** or **/dev/at0x**. If you use **fdisk** with a device other than the **x** device (e.g., with device **/dev/at0a**), **fdisk** displays values for your partitions that are totally bogus — and probably quite alarming.

After you invoke **fdisk**, it displays a warning message, then the layout of the disk whose partition-table device you named on the command line. The following gives an example layout, for a 33-megabyte AT disk:

```
Drive 0 Currently has the following logical partitions:
              [In Cylinders] [  In Tracks  ]
Number  Type  Start End Size Start  End Size Mbyte Blocks Name
0 Boot MS-DOS    0  149  150     0  899  900  7.83  15300 /dev/at0a
1     EXT-DOS  150  614  464   900 3684 2784 24.28  47430 /dev/at0b
2      UNUSED    0    0    0     0    0    0     0      0 /dev/at0c
3      UNUSED    0    0    0     0    0    0     0      0 /dev/at0d
```

In this example, partition 1 (which is accessed via device **/dev/at0a**) holds an MS-DOS file system. It is marked as the "Boot" partition, which means that the COHERENT bootstrap will boot its operating system automatically when you reboot your computer. The other columns show the size of each partition, and its beginning and end points in both cylinders and tracks.

If you invoked **fdisk** with its option **-r**, the program exits at this point. If you did *not* invoke it with option **-r**, it displays the following menu of actions:

```
Possible actions:
   0 = Quit
   1 = Change active partition (or make no partition active)
   2 = Change one logical partition
   3 = Change all logical partitions
   4 = Delete one logical partition
   5 = Change drive characteristics
   6 = Display drive information
   7 = Proceed to next drive
```

The following describes each action in detail:

0. Quit **fdisk**.

1. Change which partition is the active partition. You can also say that your system has *no* active partition. If you do so, the COHERENT bootstrap will prompt you at boot time to enter the number of the partition whose operating system you wish to boot. **fdisk** will let you set only one active partition at a time.

2. Change the dimensions (i.e., the size, beginning point, or end point) of one partition. Doing this destroys the data on that partition.

3. Change the dimensions of every partition. Doing this destroys the data on your hard disk.

4. Delete a partition.
5. Change the parameters of the drive. Use this option if COHERENT somehow has a faulty notion of your disk's size. You should never have to use this option; using it will wipe out all data on your hard disk.
6. Give summary information about the disk — that is, re-display the table shown above.
7. This option appears only if you have more than one hard disk drive. Use this option to display information about another hard disk on your system.

Before you change the dimensions of any partition on your system, read the warnings given in the notes below. When you have finished modifying your disk, **fdisk** then writes your changes into *xdev*.

Files

<fdisk.h>

See Also

commands, hard disk, ideinfo

Notes

If you change a device's partition table, reboot your system. Most device drivers will not recognize the revised partition information until a reboot occurs.

As the **-r** and **-b** options are contradictory, attempting to use them together triggers an error message.

Note that many operating systems implement a program named **fdisk**. Each manipulates a hard disk's partition table, but not all respect the fact that a disk may hold more than one operating system. In particular, the MS-DOS edition of **fdisk** can rearrange the order of entries in the partition table. If this happens, you may lose the ability to run COHERENT until the table is restored to its previous order. A sign of this problem is seeing the prompt **AT boot?** when you try to start COHERENT after running any **fdisk** program, and not being able to get past it.

Computer systems that use older releases of a BIOS may report incorrect disk parameters. Users of such systems should change the CMOS setup values if possible, but the BIOS on some older systems will not allow you to specify arbitrary values for disk parameters. Users with such systems can use the option **fdisk -c** option instead.

If you plan to install and run COHERENT and MS-DOS on the same hard disk, note the following:

- If you wish to install COHERENT and MS-DOS on the same hard drive, you *must* run the MS-DOS **fdisk** first!
- If you plan on running both operating systems, you *must* install MS-DOS first and leave some free cylinders on the disk for COHERENT as well as a free partition. You can have both primary as well as extended MS-DOS partitions on the same drive as COHERENT, but COHERENT cannot use a sub-partition of the MS-DOS extended partition. COHERENT must have one of the four *real* partitions. Failure to observe these rules will result in loss of data! *Caveat utilitor.*

fdisk.h — Header File

Fixed-disk constants and structures
#include <sys/fdisk.h>

fdisk.h declares structures and constants used to manipulate a fixed (hard) disk.

See Also

header files

fdopen() — STDIO Function (libc)

Open a stream for standard I/O
#include <stdio.h>
FILE ***fdopen(***fd, type***) int** *fd***; char** **type***;*

fdopen() allocates and returns a **FILE** structure, or *stream*, for the file descriptor *fd*, as obtained from **open()**, **creat()**, **dup()**, or **pipe()**. *type* is the manner in which you want *fd* to be opened, as follows:

r	Read a file
w	Write into a file
a	Append onto a file

Example

The following example obtains a file descriptor with **open()**, and then uses **fdopen()** to build a pointer to the **FILE** structure.

```
#include <ctype.h>
#include <stdio.h>

void adios(message)
char *message;
{
      fprintf(stderr, "%s\n", message);
      exit(1);
}

main(argc, argv)
int argc; char *argv[];
{
      extern FILE *fdopen();
      FILE *fp;
      int fd;
      int holder;

      if (--argc != 1)
            adios("Usage: example filename");

      if ((fd = open(argv[1], 0)) == -1)
            adios("open failed.");
      if ((fp = fdopen(fd, "r")) == NULL)
            adios("fdopen failed.");

      while ((holder = fgetc(fp)) != EOF) {
            if ((holder > '\177') || (holder < ' '))
                  switch(holder) {
                  case '\t':
                  case '\n':
                        break;
                  default:
                        fprintf(stderr, "Seeing char %d\n", holder);
                        exit(1);
                  }
            fputc(holder, stdout);
      }
}
```

See Also

creat(), dup(), fopen(), libc, open()
POSIX Standard, §8.2.2

Diagnostics

fdopen() returns NULL if it cannot allocate a **FILE** structure. Currently, only 20 **FILE** structures can be allocated per program, including **stdin**, **stdout**, and **stderr**.

feof() — STDIO Function (libc)

Discover stream status
#include <stdio.h>
int feof(*fp***) FILE** ***fp*;**

feof() tests the status of the argument stream *fp*. It returns a number other than zero if *fp* has reached the end of file, and zero if it has not. One use of **feof()** is to distinguish a value of -1 returned by **getw()** from an **EOF**.

Example

For an example of how to use this function, see the entry for **fopen()**.

See Also

EOF, libc
ANSI Standard, §7.9.10.2
POSIX Standard, §8.1

ferror() — STDIO Function (libc)

Discover stream status
#include <stdio.h>
int ferror(*fp***) FILE** ***fp*;**

ferror() tests the status of the file stream *fp*. It returns a number other than zero if an error has occurred on *fp*. Any error condition that it discovers persists until you either close the stream or call **clearerr()** to clear it. For write routines that employ buffers, call **fflush()** before you call **ferror()**, in case an error occurs on the last block written.

Example

This example reads a word from one file and writes it into another.

```
#include <stdio.h>

main()
{
      FILE *fpin, *fpout;
      int inerr = 0;
      int outerr = 0;
      int word;
      char infile[20], outfile[20];

      printf("Name data file you wish to copy:\n");
      gets(infile);
      printf("Name new file:\n");
      gets(outfile);

      if ((fpin = fopen(infile, "r")) != NULL) {
            if ((fpout = fopen(outfile, "w")) != NULL) {

                  for (;;) {
                        word = fgetw(fpin);
                        if (ferror(fpin)) {
                              clearerr(fpin);
                              inerr++;
                        }

                        if (feof(fpin))
                              break;
                        fputw(word, fpout);
                        if (ferror(fpout)) {
                              clearerr(fpout);
                              outerr++;
                        }
                  }

            } else {
                  printf
                        ("Cannot open output file %s\n",
                              outfile);
                  exit(1);
            }

      } else {
            printf("Cannot open input file %s\n", infile);
            exit(1);
      }

      printf("%d - read error(s)  %d - write error(s)\n",
            inerr, outerr);
      exit(0);
}
```

See Also

libc
ANSI Standard, §7.9.10.3
POSIX Standard, §8.1

fetch() — DBM Function (libgdbm)

Fetch a record from a DBM data base
#include <dbm.h>
datum fetch (*key***)**
datum *key***;**

Function **fetch()** retrieves the record with *key* from the currently opened DBM data base. The data base must first have opened by a call to function **dbminit()**.

fetch() returns a pointer to the retrieved record. If no record is available, or if an error occurred, field **dptr** within the returned record is initialized to NULL.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

fflush() — STDIO Function (libc)

Flush output stream's buffer
#include <stdio.h>
int fflush(*fp***) FILE** ***fp***;**

fflush() flushes any buffered output data associated with the file stream *fp*. The file stream stays open after **fflush()** is called. **fclose()** calls **fflush()**, so there is no need for you to call it when normally closing a file or buffer.

Example

This example demonstrates **fflush()**. When run, you will see the following:

```
Line 1
-----
Line 1
-----
Line 1
Line 2
-----
```

The call

```
fprintf(fp, "Line 2\n");
```

goes to a buffer and is not in the file when file **foo** is listed. However if you redirect the output of this program to a file and list the file, you will see:

```
Line 1
Line 1
Line 1
Line 2
-----
-----
-----
```

because the line

```
printf("-----\n");
```

goes into a buffer and is not printed until the program is over and all buffers are flushed by **exit()**.

Although the COHERENT screen drivers print all output immediately, not all operating systems work this way, so when in doubt, **fflush()**.

```
#include <stdio.h>

main()
{
      FILE *fp;
```

```
      if (NULL == (fp = fopen("foo", "w")))
            exit(1);
      fprintf (fp, "Line 1\n");
      fflush (fp);
      system ("cat foo"); /* print Line 1 */

      printf("-----\n");
      fprintf(fp, "Line 2\n");
      system("cat foo"); /* print Line 1 */
      printf("-----\n");

      fflush(fp);
      system("cat foo"); /* print Line 1 Line 2 */
      printf("-----\n");
}
```

See Also

fclose(), libc, setbuf(), write()
ANSI Standard, §7.9.5.2
POSIX Standard, §8.1

Diagnostics

fflush() returns **EOF** if it cannot flush the contents of the buffers; otherwise it returns a meaningless value.

Note, also, that all STDIO routines are buffered. **fflush()** should be used to flush the output buffer if you follow a STDIO routine with an unbuffered routine.

ffs() — Sockets Function (libsocket)

Translate a bit mask into an integer value
int mask (*mask***);**

Function **ffs()** translates the bit mask *mask* into an integer value. It returns the integer value of the first bit to be turned on (i.e., one, two, three, etc.). If no bit is turned on within *mask*, it returns zero.

See Also

libsocket

Notes

This function is used by a number of X programs that manipulate fonts. COHERENT includes it for compatibility with X11R6.

fgetc() — STDIO Function (libc)

Read character from stream
#include <stdio.h>
int fgetc(*fp***) FILE** ******fp***;**

fgetc() reads characters from the input stream *fp*. In general, it behaves the same as the macro **getc()**: it runs more slowly than **getc()**, but yields a smaller object module when compiled.

Example

This example counts the number of lines and "sentences" in a file.

```
#include <stdio.h>

main()
{
      FILE *fp;
      int filename[20];
      int ch;
      int nlines = 0;
      int nsents = 0;

      printf("Enter file to test: ");
      gets(filename);
```

```
      if ((fp = fopen(filename,"r")) == NULL) {
            printf("Cannot open file %s.\n", filename);
            exit(1);
      }

      while ((ch = fgetc(fp)) != EOF) {
            if (ch == '\n')
                  ++nlines;

            else if (ch == '.' || ch == '!' || ch == '?') {
                  if ((ch = fgetc(fp)) != '.')
                        ++nsents;

                  else
                        while((ch=fgetc(fp)) == '.')
                              ;
                  ungetc(ch, fp);
            }
      }

      printf("%d line(s), %d sentence(s).\n",
            nlines, nsents);
}
```

See Also

getc(), libc
ANSI Standard, §7.9.7.1
POSIX Standard, §8.1

Diagnostics

fgetc() returns EOF at end of file or on error.

fgetpos() — STDIO Function (libc)

Get value of file-position indicator
#include <stdio.h>
int
fgetpos(*fp***,** *position***)**
FILE **fp*; **fpos_t** **position*;

fgetpos() copies the value of the file-position indicator for the file stream pointed to by *fp* into the area pointed to by *position*. *position* is of type **fpos_t**, which is defined in the header **stdio.h**.

The function **fsetpos()** can use the information written into *position* to return the file-position indicator to where it was when **fgetpos()** was called.

fgetpos() returns zero if all went well. If an error occurred, it returns nonzero and sets **errno** to an appropriate value.

Example

This example seeks to a random line in a very large file.

```
#include <math.h>
#include <stdarg.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void
fatal(message)
char *message;
{
      fprintf(stderr, "%s\n", message);
      exit(1);
}
```

```
main(argc, argv)
int argc; char *argv[];
{
      int c;
      long count;
      FILE *ifp, *tmp;
      fpos_t loc;

      if (argc != 2)
            fatal("usage: fscanf inputfile\n");
      if ((ifp = fopen(argv[1], "r")) == NULL)
            fatal("Cannot open %s\n", argv[1]);
      if((tmp = tmpfile()) == NULL)
            fatal("Cannot build index file");

      /* seed random-number generator */
      srand ((unsigned int)time(NULL));

      for (count = 1;!feof(ifp); count++) {
            /* for monster files */
            if (fgetpos(ifp, &loc))
                  fatal ("fgetpos error");

            if (fwrite(&loc, sizeof(loc), 1, tmp) != 1)
                  fatal("Write fail on index");
            rand();
            while('\n' != (c = fgetc(ifp)) && EOF != c)
                  ;
      }

      count = rand() % count;
      fseek(tmp, count * sizeof(loc), SEEK_SET);

      if(fread(&loc, sizeof(loc), 1, tmp) != 1)
            fatal("Read fail on index");

      fsetpos(ifp, &loc);
      while((c = fgetc(ifp)) != EOF) {
            if(c =='@')
                  putchar('\n');
            else
                  putchar(c);

            if(c == '\n')
                  break;
      }
}
```

See Also

fseek(), fsetpos(), ftell(), libc, rewind()
ANSI Standard, §7.9.9.1

Notes

The ANSI Standard introduced **fgetpos()** and **fsetpos()** to manipulate a file whose file-position indicator cannot be stored within a **long**. Under COHERENT **fgetpos()** behaves the same as the function **ftell()**.

fgets() — STDIO Function (libc)

Read line from stream
#include <stdio.h>
char *fgets(*s***,** *n***,** *fp***)**
char ****s*; int** *n***; FILE ****fp***;**

fgets() reads characters from the stream *fp* into string *s* until either *n*-1 characters have been read, or a newline or EOF is encountered. It retains the newline, if any, and appends a null character at the end of of the string. **fgets()** returns the argument *s* if any characters were read, and NULL if none were read.

Example

This example looks for the pattern given by **argv[1]** in standard input or in file **argv[2]**. It demonstrates the

functions **pnmatch()**, **fgets()**, and **freopen()**.

```
#include <stdio.h>
#define MAXLINE 128
char buf[MAXLINE];

void fatal(s) char *s;
{
      fprintf(stderr, "pnmatch: %s\n", s);
      exit(1);
}

main(argc, argv)
int argc; char *argv[];
{
      if (argc != 2 && argc != 3)
            fatal("Usage: pnmatch pattern [ file ]");

      if (argc==3 && freopen(argv[2], "r", stdin)==NULL)
            fatal("cannot open input file");

      while (fgets(buf, MAXLINE, stdin) != NULL) {
            if (pnmatch(buf, argv[1], 1))
                  printf("%s", buf);
      }

      if (!feof(stdin))
            fatal("read error");
      exit(0);
}
```

See Also

fgetc(), gets(), libc
ANSI Standard, §7.9.7.2
POSIX Standard, §8.1

Diagnostics

fgets() returns NULL if an error occurs, or if **EOF** is seen before any characters are read.

fgetw() — STDIO Function (libc)

Read integer from stream
#include <stdio.h>
int fgetw(*fp***) FILE** ***fp*;**

fgetw() reads an integer from the stream *fp*.

Example

For an example of this function, see the entry for **ferror()**.

See Also

fputw(), libc

Notes

fgetw() returns EOF on errors. A call to **feof()** or **ferror()** may be necessary to distinguish this value from a genuine end-of-file signal.

field — Definition

A **field** is an area that is set apart from whatever surrounds it, and that is defined as containing a particular type of data. In the context of C programming, a field is either an element of a structure, or a set of adjacent bits within an **int**.

See Also

bit map, data formats, Programming COHERENT, structure

file — Definition

The way to access bits

The term **file** is used throughout the world of computing. Because there are several distinct types of COHERENT "files," understanding what COHERENT means by a "file" can help you grasp how COHERENT works.

A file is a mass of bits that is given a name and is stored on some physical medium (e.g., floppy disk, hard disk, RAM disk, or CD-ROM). These bits may represent data (e.g., ASCII or EBCDIC characters) or machine-executable instructions. COHERENT defines a number of different types of files. A file's type defines its behavior. Some common file types include the following:

regular This file points to a location on a disk, which can be read or written. The location pointed to can contain data (e.g., text) or executable instructions in the form of shell commands or binary instructions. Regular files are sometimes called *ordinary* files.

directory
A directory holds the names and addresses of other files, including other directories.

special Special files designate COHERENT devices. A device can represent a physical device, such as a floppy disk drive, a printer port, or a serial port. It can also represent a part of a physical device, such as a RAM disk (representing part of memory) or one partition of a hard disk. It can also represent a logical device that has no physical counterpart, like the bit bucket **/dev/null**.

Special files come in two flavors: *character special* and *block special*. The former access data in streams (that is, one character at a time), and so access devices like tape drives and serial ports. The latter access one block at a time, and so access disk drives and other devices that return their data in block-sized chunks. (COHERENT defines a block as being 512 characters.)

FIFO This is a variety of regular file that contains semantics to hook together two processes, just like a pipe '|' in the COHERENT shell. See the Lexicon article **named pipe** for details on this variety of file.

process
This kind of file corresponds one-to-one with the existence of a process on a system. It tends to be short-lived.

Files live with a *file system*, which organizes the files hierarchically within directories. The Lexicon entry for the command **mkfs** gives some technical information on how a file system is constructed. The Lexicon entry for the command **mount** gives some information on how a file system relates to device on which it lives, and how different file systems from different partitions are hooked together to form one large file system for the entire computer.

The same file can have (and be accessed by) more than one name. The Lexicon entry for the command **ln** shows how you can link additional names to a file. The entry for the system call **unlink()** gives some details on the relationship between a file and its names.

Finally, a file has *permissions* associated with it. Every file is owned by someone; and the owner can restrict access to the file if she wishes. The Lexicon entry for the command **ls** describes what permissions are available for a file. The entry for the command **chmod** shows how you can change permissions on a file. The entry for the command **umask** shows how you can change the permissions that COHERENT gives by default to any files that you create.

See Also

chgrp, chmod, chown, directory, FILE, device drivers, ls, mkfs, named pipe, open(), Programming COHERENT, stream, umask, Using COHERENT
ANSI Standard §4.9.3

file — Command

Guess a file's type
file *file ...*

file examines each *file* and takes an educated guess as to its type. **file** recognizes the following classes of text files: files of commands to the shell; files containing the source for a C program; files containing **yacc** or **lex** source; files containing assembly language source; files containing unformatted documents that can be passed to **nroff**; and plain text files that fit into none of the above categories.

file recognizes the following classes of non-text or binary data files: the various forms of archives, object files, and link modules for various machines, and miscellaneous binary data files.

See Also

commands, ls, size

Notes

Because **file** only reads a set amount of data to determine the class of a text file, mistakes can happen.

FILE — Definition

Descriptor for a file stream
#include <stdio.h>

FILE describes a *file stream* which can be either a file on disk or a peripheral device through which data flow. It is defined in the header file **stdio.h**.

A pointer to **FILE** is returned by **fopen()**, **freopen()**, **fdopen()**, and related functions.

The **FILE** structure is as follows:

```
typedef struct    FILE
{
      unsigned char *cp,
                    *dp,
                    *bp;
      int   cc;
      int   (*gt)(),
            (*pt)();
      int   ff;
      char  fd;
      int   uc;
} FILE;
```

cp points to the current character in the file. **dp** points to the start of the data within the buffer. **bp** points to the file buffer. **cc** is the number of unprocessed characters in the buffer. **gt** and **pt** point, respectively, to the functions **getc()** and **putc()**. **ff** is a bit map that holds the various file flags, as follows:

_FINUSE	0x01	Unused
_FSTBUF	0x02	Used by macro **setbuf()**
_FUNGOT	0x04	Used by **ungetc()**
_FEOF	0x08	Tested by macro **feof()**
_FERR	0x10	Tested by macro **ferror()**

fd is the file descriptor, which is used by low-level routines like **open()**; it is also used by **reopen()**. Finally, **uc** is the character that has been "ungotten" by the function **ungetc()**, should it be used.

See Also

fopen(), freopen(), Programming COHERENT, stdio.h, stream
ANSI Standard, §7.9.1

file descriptor — Definition

A **file descriptor** is an integer that indexes an area in COHERENT's internal list of file descriptors. COHERENT system calls, including **open()**, **close()**, and **lseek()**, use it to describe a file.

Please note that a file descriptor is *not* the same as a **FILE** structure, which is used by the STDIO routines **fopen()**, **fclose()**, or **fread()**.

See Also

file, FILE, Programming COHERENT

fileno() — STDIO Function (libc)

Get file descriptor
#include <stdio.h>
int fileno(*fp***) FILE** ***fp*;**

fileno() returns the file descriptor associated with the file stream *fp*. The file descriptor is the integer returned by **open()** or **creat()**; it corresponds to a **FILE** structure, as returned by the STDIO function **fopen()**.

Example

This example reads a file descriptor and prints it on the screen.

```
#include <stdio.h>

main(argc,argv)
int argc; char *argv[];
{
      FILE *fp;
      int fd;

      if (argc !=2) {
            printf("Usage: fd_from_fp filename\n");
            exit(0);
      }

      if ((fp = fopen(argv[1], "r")) == NULL) {
            printf("Cannot open input file\n");
            exit(0);
      }

      fd = fileno(fp);
      printf("The file descriptor for %s is %d\n",
            argv[1], fd);
}
```

See Also

FILE, file descriptor, libc
POSIX Standard, §8.2.1

filsys.h — Header File

Structures and constants for super block
#include <sys/filsys.h>

filsys.h declares structures and constants used by functions that manipulate a file system's super block.

See Also

header files

filter — Definition

A *filter* is a program that reads a stream of input, transforms it in a precisely defined manner, and writes it to another stream. Two or more filters can be coupled with *pipes* to perform a complex transformation on a stream of input.

See Also

pipe, Using COHERENT

find — Command

Search for files satisfying a pattern
find *directory ...* **[***expression ...***]**

find traverses each given *directory*, testing each file or subdirectory found with the *expression* part of the command line. The test can be the basis for deciding whether to process the file with a given command.

If the command line specifies no *expression* or specifies no execution or printing (**-print**, **-exec**, or **-ok**), by default **find** prints the pathnames of the files found.

In the following, *file* means any file: directory, special file, ordinary file, and so on. Numbers represented by *n* may be optionally prefixed by a '+' or '-' sign to signify values greater than *n* or less than *n*, respectively.

find recognizes the following *expression* primitives:

-**atime** *n* Match if the file was accessed in the last *n* days.

-**ctime** *n* Match if the i-node associated with the file was changed in the last *n* days, as by **chmod**.

-**exec** *command*
Match if *command* executes successfully (has a zero exit status). The *command* consists of the following arguments to **find**, terminated by a semicolon ';' (escaped to get past the shell). **find** substitutes the current pathname being tested for any argument of the form '{}'.

-**group** *name*
Match if the file is owned by group *name*. If *name* is a number, the owner must have that group number.

-**inum** *n* Match if the file is associated with i-number *n*.

-**links** *n* Match if the number of links to the file is *n*.

-**mtime** *n* Match if the most recent modification to the file was *n* days ago.

-**name** *pattern*
Match if the file name corresponds to *pattern*, which may include the special characters '*', '?', and '[...]' recognized by the shell **sh**. The *pattern* matches only the part of the file name after any slash ('/') characters.

-**newer** *file* Match if the file is newer than *file*.

-**nop** Always match; does nothing.

-**ok** *command*
Same as **-exec** above, except prompt interactively and only executes *command* if the user types response **'y'**.

-**perm** *octal* Match if owner, group, and other permissions of the file are the *octal* bit pattern, as described in **chmod**. When *octal* begins with a **'-'** character, more of the permission bits (setuid, setgid, and sticky bit) become significant.

-**print** Always match; print the file name.

-**size** *n* Match if the file is *n* blocks in length; a block is 512 bytes long.

-**type** *c* Match if the type of the file is *c*, chosen from the set **bcdfmp** (for block special, character special, directory, ordinary file, multiplexed file, or pipe, respectively).

-**user** *name* Match if the file is owned by user *name*. If *name* is a number, the owner must have that user number.

exp1 exp2 Match if both expressions match. **find** evaluates *exp2* only if *exp1* matches.

exp1 **-a** *exp2*
Match if both expressions match, as above.

exp1 **-o** *exp2*
Match if either expression matches. **find** evaluates **exp2** only if **exp1** does not match.

! *exp* Match if the expression does *not* match.

(*exp*) Parentheses are available for expression grouping.

Examples

A **find** command to print the names of all files and directories in user **fred**'s directory is:

```
find /usr/fred
```

The following, more complicated **find** command prints out information on all **core** and object (**.o**) files that have not been changed for a day. Because some characters are special both to **find** and **sh**, they must be escaped with '\' to avoid interpretation by the shell.

```
find / \( -name core -o -name \*.o \) -mtime +1 \
-exec ls -l {} \;
```

Finally, the following example implements a simple tool for keeping files on two COHERENT systems in synch with each other. **find** reads directory **src** and passes to **uucp** the names of all files that are newer than file **last_upload**. It then uses the command **touch** to update the date on **last_upload**, to use it as a marker of when the last upload was performed.

LEXICON

```
find $HOME/src -type f -newer last_upload | while read filename
do
      uucp -r -nyou $filename yoursystem!~/
      echo Queued file $filename to yoursystem ...
done | mail somebodyorother
touch last_upload
```

See Also

chmod, commands, ls, sh, srcpath, test

findmouse — Command

Examine a port to see if a mouse is plugged into it
/usr/local/bin/findmouse *port*

The command **findmouse** opens *port* so you can examine whether a mouse is plugged into it.

port must be the full path name of the local, polled, serial-port device. For example, to check whether a mouse is plugged into serial port 1, use the command:

```
/usr/local/bin/findmouse /dev/com1pl
```

When you invoke **findmouse**, it opens *port*, then asks you to "wiggle" your mouse. As you move the mouse around your desk, **findmouse** polls the port and display on the screen any data read from it. You should see the mouse data on your screen in the form of two-digit hexadecimal numbers.

To exit from **findmouse**, press any key.

findmouse prints an error message an exits should use incorrect command syntax, or if it cannot open a requested port.

See Also

asy, commands, poll()

firstkey() — DBM Function (libgdbm)

Retrieve the first record from a DBM data base
#include <dbm.h>
datum firstkey()

Function **firstkey()** retrieves the first record from the currently open DBM data base. The data base must have been opened by a call to function **dbminit()**.

firstkey() returns a pointer to the retrieved record. If no record is available (i.e., the data base is empty), or if an error occurred, field **dptr** within the returned record is initialized to NULL.

Please note that the hashing algorithm used the DBM functions dictates which record is "first" within the data base. A loop that uses this function plus the function **nextkey()** will retrieve every record from the data base; however, the records probably will not be in the order you expect.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

fixterm() — terminfo Function

Set the terminal into program mode
#include <curses.h>
fixterm()

COHERENT comes with a set of functions that let you use **terminfo** descriptions to manipulate a terminal. **fixterm()** restores the terminal to its internal conditions, as set by the **curses/terminfo** library. Your program should call **fixterm()** after it returns from a shell escape.

See Also

curses.h, resetterm(), terminfo

float — C Keyword

Data type

Floating point numbers are a subset of the real numbers. Each has a built-in radix point (or "decimal point") that shifts, or "floats", as the value of the number changes. It consists of the following: one sign bit, which indicates whether the number is positive or negative; bits that encode the number's *exponent*; and bits that encode the number's *fraction*, or the number upon which the exponent works. In general, the magnitude of the number encoded depends upon the number of bits in the exponent, whereas its precision depends upon the number of bits in the fraction.

The ranges of values that can be held by a COHERENT **float** are set in header file **float.h**.

The exponent often uses a *bias*. This is a value that is subtracted from the exponent to yield the power of two by which the fraction will be increased.

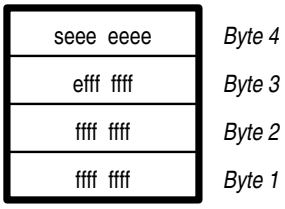
Floating point numbers come in two levels of precision: single precision, called **float**s; and double precision, called **double**s. With most microprocessors, **sizeof(float)** returns four, which indicates that it is four **char**s (bytes) long, and **sizeof(double)** returns eight.

Several formats are used to encode **float**s, including IEEE, DECVAX, and BCD (binary coded decimal).

The following describes DECVAX, IEEE, and BCD formats, for your information.

DECVAX Format

The 32 bits in a **float** consist of one sign bit, an eight-bit exponent, and a 24-bit fraction, as follows. Note that in this diagram, 's' indicates "sign", 'e' indicates "exponent", and 'f' indicates "fraction":



The exponent has a bias of 129.

If the sign bit is set to one, the number is negative; if it is set to zero, then the number is positive. If the number is all zeroes, then it equals zero; an exponent and fraction of zero plus a sign of one ("negative zero") is by definition not a number. All other forms are numeric values.

The most significant bit in the fraction is always set to one and is not stored. It is usually called the "hidden bit".

The format for **double**s simply adds another 32 fraction bits to the end of the **float** representation, as follows:

IEEE Format

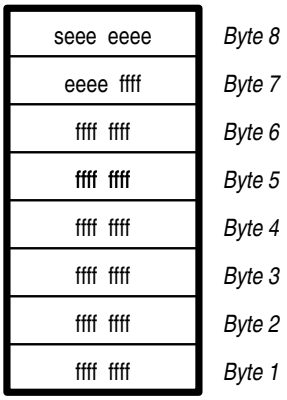
The IEEE encoding of a **float** is the same as that in the DECVAX format. Note, however, that the exponent has a bias of 127, rather than 129.

LEXICON

Unlike the DECVAX format, IEEE format assigns special values to several floating point numbers. Note that in the following description, a *tiny* exponent is one that is all zeroes, and a *huge* exponent is one that is all ones:

- A tiny exponent with a fraction of zero equals zero, regardless of the setting of the sign bit.
- A huge exponent with a fraction of zero equals infinity, regardless of the setting of the sign bit.
- A tiny exponent with a fraction greater than zero is a denormalized number, i.e., a number that is less than the least normalized number.
- A huge exponent with a fraction greater than zero is, by definition, not a number. These values can be used to handle special conditions.

An IEEE **double**, unlike DECVAX format, increases the number of exponent bits. It consists of a sign bit, an 11-bit exponent, and a 53-bit fraction, as follows:

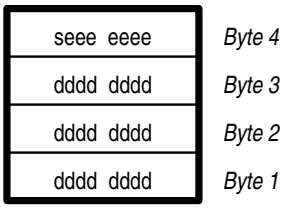


The exponent has a bias of 1,023. The rules of encoding are the same as for **float**s.

BCD Format

The BCD format ("binary coded decimal", also called "packed decimal") is used to eliminate rounding errors that alter the worth of an account by a fraction of a cent. It consists of a sign, an exponent, and a chain of four-bit numbers, each of which is defined to hold the values zero through nine.

A BCD **float** has a sign bit, seven bits of exponent, and six four-bit digits. In the following diagrams, 'd' indicates "digit":



A BCD **double** has a sign bit, 11 bits of exponent, and 13 four-bit digits, as follows:

Passing the hexadecimal numbers A through F in a digit yields unpredictable results.

The following rules apply when handling BCD numbers:

- A tiny exponent with a fraction of zero equals zero.
- A tiny exponent with a fraction of non-zero indicates a denormalized number.
- A huge exponent with a fraction of zero indicates infinity.
- A huge exponent with a fraction of non-zero is, by definition, not a number; these non-numbers are used to indicate errors.

COHERENT Floating Point

COHERENT 286 uses DECVAX floating-point format. COHERENT 386 uses IEEE floating-point format. Please note that this does *not* mean that the COHERENT-386 floating-point software fully implements the IEEE standard; for example, it does not support denormals.

To allow you to convert binary data from one floating-point format to another, COHERENT comes with four functions with which you can convert DECVAX-format floating-point numbers to IEEE format, and vice versa. They are as follows:

decvax_d() Convert an IEEE **double** to DECVAX format.

decvax_f() Convert an IEEE **float** to DECVAX format.

ieee_d() Convert a DECVAX **double** to IEEE format.

ieee_f() Convert a DECVAX **float** to IEEE format.

For details, see their respective entries in the Lexicon.

See Also

C keywords, data formats, decvax_d, decvax_f, double, ecvt(), em87, fcvt(), float, float.h, gcvt(), ieee_d, ieee_f
The Art of Computer Programming, vol. 2, page 180*ff*
ANSI Standard, §6.1.2.5

Notes

The COHERENT-386 preprocessor implicitly defines the macro **_IEEE**, whereas the COHERENT-286 preprocessor implicitly defines the macro **_DECVAX**. These can be used to conditionally include code that applies to a specific edition of COHERENT. If you were writing code that intensively used floating-point numbers and you want to compile the code under both editions of COHERENT, you can write code of the form:

```
#ifdef _DECVAX
        ...
#elif _IEEE
        ...
#endif
```

The C preprocessor under each edition of COHERENT will ensure that the correct code is included for compilation.

float.h — Header File

Define constants for floating-point numbers

The header file **float.h** defines the following manifest constants, which mark the limits for computation of floating-point numbers. The prefixes **DBL**, **FLT**, and **LDBL** refer, respective, to **double**, **float**, and **long double**:

DBL_DIG
Number of decimal digits of precision.

DBL_EPSILON
Smallest possible floating-point number *x*, such that 1.0 plus *x* does not test equal to 1.0.

DBL_MANT_DIG
Number of digits in the floating-point mantissa for base **FLT_RADIX**.

DBL_MAX
Largest number that can be held by type **double**.

DBL_MAX_EXP
Largest integer such that the value of **FLT_RADIX** raised to its power minus one is less than or equal to **DBL_MAX**.

DBL_MAX_10_EXP
Largest integer such that ten raised to its power is less than or equal to **DBL_MAX**.

DBL_MIN
Smallest number that can be held by type **double**.

DBL_MIN_EXP
Smallest integer such that the value of **FLT_RADIX** raised to its power minus one is greater than or equal to **DBL_MIN**.

DBL_MIN_10_EXP
Smallest integer such that ten raised to its power is greater than or equal to **DBL_MAX**.

FLT_DIG
Number of decimal digits of precision.

FLT_EPSILON
Smallest floating-point number *x*, such that 1.0 plus *x* does not test equal to 1.0.

FLT_MANT_DIG
Number of digits in the floating-point mantissa for base **FLT_RADIX**.

FLT_MAX
Largest number that can be held by type **float**.

FLT_MAX_EXP
Largest integer such that the value of **FLT_RADIX** raised to its power minus one is less than or equal to **FLT_MAX**.

FLT_MAX_10_EXP
Largest integer such that ten raised to its power is less than or equal to **FLT_MAX**.

FLT_MIN
Smallest number that can be held by type **float**.

FLT_MIN_EXP
Smallest integer such that the value of **FLT_RADIX** raised to its power minus one is greater than or equal to **FLT_MIN**.

FLT_MIN_10_EXP
Smallest integer such that ten raised to its power is greater than or equal to **FLT_MIN**.

FLT_RADIX
Base in which the exponents of all floating-point numbers are represented.

FLT_ROUNDS
Manner of rounding used by the implementation. The ANSI Standard defines the rounding codes as follows:

-1 Indeterminable, i.e., no strict rules apply
0 Toward zero, i.e., truncation
1 To nearest, i.e., rounds to nearest representable value
2 Toward positive infinity, i.e., always rounds up
3 Toward negative infinity, i.e., always rounds down

COHERENT uses type-1 rounding.

LDBL_DIG
Number of decimal digits of precision.

LDBL_EPSILON
Smallest floating-point number *x*, such that 1.0 plus *x* does not test equal to 1.0.

LDBL_MANT_DIG
Number of digits in the floating-point mantissa for base **FLT_RADIX**.

LDBL_MAX
Largest number that can be held by type **long double**.

LDBL_MAX_EXP
Largest integer such that the value of **FLT_RADIX** raised to its power minus one is less than or equal to **LDBL_MAX**.

LDBL_MAX_10_EXP
Largest integer such that ten raised to its power is less than or equal to **LDBL_MAX**.

LDBL_MIN
Smallest number that can be held by type **long double**. Must be no greater than 1E-37.

LDBL_MIN_EXP
Smallest integer such that the value of **FLT_RADIX** raised to its power minus one is greater than or equal to **LDBL_MIN**.

LDBL_MIN_10_EXP
Smallest integer such that ten raised to its power is greater than or equal to **LDBL_MIN**.

See Also

double, float, header files
ANSI Standard, §5.2.4.2.2

Notes

COHERENT's C compiler does not yet implement type **long double**.

floor() — Mathematics Function (libm)

Set a numeric floor
#include <math.h>
double floor(*z***) double** *z***;**

floor() sets a numeric floor. It returns a double-precision floating point number whose value is the largest integer less than or equal to *z*.

Example

For an example of this function, see the entry for **ceil()**.

See Also

abs(), ceil(), fabs(), frexp(), libm
ANSI Standard, §7.5.6.3
POSIX Standard, §8.1

floppy disks — Technical Information

The COHERENT system lets you read or write to floppy disks, using a variety of different formats. You can choose the format that best suits the task at hand.

Disks Supported

COHERENT lets you use either 3.5-inch or 5.25-inch disks, in either high or low density; what you use depends upon the type of hardware that you have. The following table gives some commonly used diskette device names and formats. The minor number of each device is also given; note that all floppy-disk devices have the major number of 4:

Device Name	Sectors/Track	Heads	Sectors	Bytes	Format	Minor Number
/dev/f9a0	9	2	720	360 KB	5.25"	12
/dev/f9a1	9	2	720	360 KB	5.25"	28
/dev/fqa0	9	2	1440	720 KB	3.5"	13
/dev/fqa1	9	2	1440	720 KB	3.5"	29
/dev/fha0	15	2	2400	1.2 MB	5.25"	14
/dev/fha1	15	2	2400	1.2 MB	5.25"	30
/dev/fva0	18	2	2880	1.44 MB	3.5"	15
/dev/fva1	18	2	2880	1.44 MB	3.5"	31

Device names ending in '0' indicate drive A:, names ending in '1' indicate drive B:. For a fuller description of COHERENT's floppy-disk devices, see the Lexicon entry for **fd**.

MS-DOS Format

COHERENT lets you read or write to floppy disks that contain MS-DOS file systems. Both tasks use the commands **doscp** or **doscpdir**. These commands are discussed in full in their respective Lexicon entries.

To read files from an MS-DOS disk, use **doscp** with the name of the appropriate for the floppy-disk device that you will be using (as given in the above table). For example, to copy binary file **fred.exe** to the current directory from a low-density, 5.25-inch MS-DOS floppy disk in drive A, use the following command:

```
doscp /dev/f9a0:fred.exe .
```

The following command copies to the current directory all files on a high-density, 5.25-inch MS-DOS floppy disk in drive B:

```
doscp /dev/fha1:\* .
```

To write a file to a preformatted MS-DOS floppy disk, again use the **doscp** command, but invert the order of the arguments. For example, to write file **fred.ms**, which contains text, to a low-density, 5.25-inch MS-DOS floppy disk in drive A, use the following command:

```
doscp -a fred.ms /dev/f9a0:
```

Note that the 'a' flag in the command line tells COHERENT to convert linefeeds to the linefeed/carriage return combination, as used by MS-DOS. You will want to use this flag *only* when transferring text files to or from an MS-DOS floppy disk.

The following command copies all files in the current directory to a high-density, 3.5-inch MS-DOS floppy disk in drive B:

```
doscpdir . /dev/fva1:
```

Note that when you copy a file to an MS-DOS floppy disk, COHERENT observes the MS-DOS file-name conventions: it permits only eight characters to the left of the period, and only three characters to the right of it.

(It should be noted in passing that you can use the **doscp** or **doscpdir** to read files from or write files to an MS-DOS partition on your hard disk. All that is necessary is to replace the name of floppy-disk device with that of the hard-disk device for the partition in question. See the Lexicon entry for **at** for a list of hard-disk devices; see the entry for **fdisk** for information on how to read the layout of your hard disk; and see the entries for **doscp** and **doscpdir** for details of how to use these commands.)

Finally, COHERENT lets you format a floppy disk and create an MS-DOS file system on it. To do so, you must use the commands **fdformat** and **dosformat**. **fdformat** is described in detail in its Lexicon article.

To format a high-density, 5.25-inch floppy disk in drive B and write an MS-DOS file system onto it, use the following commands:

```
/etc/fdformat -av /dev/fha1
dosformat /dev/fha1:
```

COHERENT Format

If you wish, you can create a COHERENT file system on a floppy disk, mount it, and use standard COHERENT commands to manipulate the files on it. This illustrates well the fact that to COHERENT a file system is a file system, whether it resides on a hard, a floppy disk, or any other mass-storage device. You can use such mountable floppy disks as an easy method of backing up files, or as a flexible extension to any other file system that you have currently mounted.

To create a COHERENT file system on a floppy disk, you must use the commands **fdformat** and **mkfs**. Each is described in detail in its own Lexicon article. The following example creates a COHERENT file system on a high-density, 3.5-inch floppy disk placed in drive B:

```
/etc/fdformat -av /dev/rfva1
/etc/mkfs /dev/fva1 2880
```

In this example, command **fdformat** formatted the disk. The option **-v** tells **fdformat** to use its verification mode. This takes longer, but ensures that the disk is good. If this command fails, it means that the floppy disk has a bad block or sector: throw it away and try again.

Command **mkfs** builds a COHERENT file system on the disk. The file system has 2,880 blocks (1.44 megabytes) of space, which is appropriate for a high-density, 3.5-inch floppy disk.

Now that the file system is created on the disk, you must mount it. To do so, use the script **mount**; this is described in its Lexicon entry. This mounts the file system on directory **/f0** if the disk drive is drive 0 (A:); or **f1** if the disk drive is drive 1 (B:).

While it is customary to mount file systems under directory '/', you are not required to do it. For example, if your login identifier is **fred** and your home directory is **/usr/fred**, you can mount the floppy disk's file system onto a subdirectory of **/usr/fred** and so make the floppy disk, in effect, an extension of your home directory. To mount a floppy on a directory other than its default, use the command **/etc/mount**. The following command does this for the 3.5-inch disk we formatted in the above example:

```
/etc/mount /dev/fva1 /usr/fred/temp
```

Now, all files you copy into directory **/usr/fred/temp** using the **cp** command will be written directly onto the floppy disk. Note that you may need to log in as the superuser **root** and use the command **chown** to ensure that **fred** owns the file system on that floppy disk. For details on **chown**, see its entry in the Lexicon. For details on shorthand notations for **mount**, see its entry in the Lexicon.

One important point about mounting file systems: before you remove a COHERENT-formatted floppy disk from its drive, you **must** first use the command **/etc/umount** to unmount its file system. If you do not, all data that COHERENT has stored in its buffers will not be written to the disk, and may be lost. Worse, if you remove one COHERENT disk and insert another without unmounting the old disk and mounting the new one, COHERENT will write all data in its buffers onto the new disk without regard for what that disk contains; in all likelihood, this will trash the file system on the new disk and render its data unreadable. So, the lesson is: *always unmount a floppy disk before you remove it!*. To unmount the floppy disk we used in our previous example, use the command:

```
/etc/umount /dev/fva1
```

By the way, that's not a misprint: the command is **umount**, not "unmount".

Finally, please note that you can mount only a COHERENT file system. You *cannot* mount a file system created with MS-DOS, XENIX, or any other operating system.

You can, however *import* a set of files — including their directory structure — from UNIX, XENIX, or any other UNIX-like operating system by using the utilities. **cpio** or **tar**. Each of these utilities uses a backup algorithm that is implemented on many operating systems. To import files from another operating system, go to the machine that holds the files you want and use its version of **cpio** or **tar** to back up the files or directories to a set of floppy disks or cartridge tape. Then bring the floppy disks back to your COHERENT system and use COHERENT's implementation of **cpio** to read the back-up disks. The following section gives directions on how to do this; or see the Lexicon entries for **cpio** and **tar** for more information.

Raw Format

Finally, COHERENT lets you use floppy disks in their raw form as a backup medium, much as you would use magnetic tape on a larger computer. You must first use the command **/etc/fdformat** with the **-v** option to format

the floppy disks you will be using; it is also wise to label and number the disks so you can keep them in some reasonable order. Then you can use any of COHERENT's archiving utilities, such as **tar** or **cpio**, to archive directories or entire file systems onto the disks. It is recommended that you format a generous supply of floppy disks before you begin; if you run short of disks while archiving your files, you will have to abort, format more disks, and begin again. For details on how to use the archiving programs, see their respective entries in the Lexicon.

Interleave

The "interleave" of a disk device refers to the pattern with which blocks are scattered around a disk cylinder. It can have a drastic effect on the speed with which data are read from and written to a disk.

The interleave is set by the file system written onto that disk. Thus, under COHERENT the interleave is set by the command **/etc/mkfs**. By default, this command sets the interleave pattern to six. You can request a different interleave pattern; however, the proper interleave for a floppy disk can vary wildly, depending upon what disk drives you have, your CPU speed, amount of RAM, and several other variables. The best way to discover the interleave pattern is to experiment.

The following script, by Fred Smith (fredex%fcshome@merk.merk.com), formats a floppy disk to a specified set of factors, generates a file system, and runs a program to exercise it. By running this program with a number of different settings, you can find which is best for your system. You will find this to be especially helpful if you work frequently with floppy disks:

```
# usage: doit <interleave> <skew> <device name> <tracks (not sectors) per drive>
#   for a 3.5dshd in drive 1:  sh doit 3 6 fva1 2880
#   for a 5.25dshd in drive 0: sh doit 3 6 fha0 2400
# assumes that iozone is in the current directory, and that there is a
# subdirectory named 'test', over which the floppy can be mounted.

echo /etc/fdformat -a -i $1 -o $2 /dev/r$3
/etc/fdformat -a -i $1 -o $2 /dev/r$3
/etc/badscan -v -o flop /dev/$3 $4

# in case you want to modify the permissions of the new file system.
# if you don't want to do the vi, then run this as root.
#vi flop

      /etc/mkfs /dev/$3 flop
      /etc/mount /dev/$3 ./test
      cd test
      ../iozone
      cd ..
      /etc/umount /dev/$3
```

Debugging Floppy-Disk Problems

The COHERENT floppy-disk driver has been used frequently by tens of thousands of users over a number of years, and has been found to be sound. However, from time to time a problem can arise. This usually occurs when users install new equipment into their systems. If you continually see error messages that indicate a problem with the floppy-disk drive, e.g., **door open**, try the following steps to diagnose the problem:

1. Is CMOS configured for the floppy-disk drives? The CMOS on your machine may have been "clobbered" by an event that has nothing to do with COHERENT — e.g., a power surge.

   To check your CMOS, you can reboot your system; the BIOS on practically every computer includes a program for reading and resetting the CMOS. Or, you can read the output of device **/dev/cmos**. The Lexicon entry **cmos** describes how to interpret the output of this device.

2. If you have switched hard drives, did you change IDE controllers or alter any jumpers? If the same card controls both floppy and hard drives, you may have moved a jumper wrongly. It may also be that the new controller has a bug.

3. Try using the command **/etc/conf/bin/idtune** to change the value of variable **FL_DSK_CH_PROB**; then use the command **/etc/conf/bin/idmkcoh** to link a new kernel, and boot the new kernel. To check the current value of that variable (or of any tunable variable), use the command **idtune -p**.

4. Is any other equipment conflicting with the drive in question, such as a QIC-80 or QIC-40 tape drive? Try pulling the device in question, and see if that makes the problem go away.

5. Check that all cables are secure and all cards seated properly. If your machine is loaded with equipment, its interior can be a rat's nest of cables and connectors; and while installing new equipment, it is easy to loosen a cable or jar a card so that it no longer works.

6. Try the following command with a floppy disk in place, just after you have booted COHERENT and before any other access to the drive:

```
dd if=/dev/rdev of=/dev/null count=2 bs=30b
```

dev is names the floppy-disk device in question, e.g., **fha0** or **fva1**. This command may help if the driver is not getting the recalibration status it expects.

7. If all else fails, try swapping out the controller or drive. It may be that the device simply has failed.

See Also

Administering COHERENT, badscan, cpio, doscp, doscpdir, dosformat, fd, fdformat, gtar, mkfs, mount, umount

Notes

You can create a version of the COHERENT operating system that runs from a floppy disk. Such a version of COHERENT can be used to create test or backup systems for device drivers or other applications. For directions on how to make a version of COHERENT that boots from a floppy disk, see the Lexicon entry **booting**.

fmap — Command

Measure fragmentation of the free list
fmap *device*

Command **fmap** tests how fragmented the free list is on COHERENT file system *device*. It briefly displays its results and returns an exit status that is equivalent to the percent of fragmentation found on *device*.

You can use the amount of fragmentation of the free list to decide whether to de-fragment *device*. When the freelist is fragmented, writing a file creates a file that is not physically contiguous; and this, in turn, slows disk I/O.

device must be a partition on your hard disk or a floppy disk rather than an entire hard drive. It can be either the raw or the "cooked" (block) device. For example, the command

```
fmap /dev/rat0a
```

tells **fmap** to map the free list on the first partition on the first drive.

Because **fmap** returns an exit status equal to the integer portion of the percentage of fragmentation found, you can use it in a shell script to alert the system administrator when the file system needs attention. For example, the following shell script tests the level of fragmentation on the device given as an argument; if the fragmentation exceeds 5%, it sends mail to the superuser **root**:

```
fmap /dev/$1
a=$?
if expr $a > 5
then
      echo  -n "fmap of " >/tmp/rootmail
      echo -n $1 >>/tmp/rootmail
      echo -n " shows " >>/tmp/rootmail
      echo -n $a >>/tmp/rootmail
      echo " percent fragmentaion" >>/tmp/rootmail
      echo -n $a >>/tmp/rootmail
      echo " is greater than 5"  >>/tmp/rootmail
      echo -n "therefore, it is time to defrag " >>/tmp/rootmail
      echo $1 >>/tmp/rootmail
      echo "bye" >>/tmp/rootmail
      mail root </tmp/rootmail
      rm /tmp/rootmail
fi
exit 0
```

See Also

commands, dpac, fsck, qpac, upac

Notes

fmap was written by Randy Wright (rw@rwsys.wimsey.bc.ca).

fmod() — Mathematics Function (libm)

Calculate modulus for floating-point number
#include <math.h>
double
fmod(*number***,** *divisor***)**
double *number***,** *divisor***;**

The mathematics function **fmod()** divides *number* by *divisor* and returns the remainder. If *divisor* is nonzero, the return value will have the same sign as *divisor*. If *divisor* is zero, however, the COHERENT implementation of **fmod()** returns 0.0 and sets **errno EDOM**.

See Also

ceil(), fabs(), floor(), libm
ANSI Standard, §7.5.6.4
POSIX Standard, §8.1

fmt — Command

Adjust the length of lines in a file of text
fmt [-*width***] [***textfile ... textfile***]**

The command **fmt** reads each *textfile* named on its command line, and adjusts it so that each line is approximately *width* characters long. It preserves indentation and word spacing.

If you name no *textfile* on its command line, **fmt** reads the standard input. If you do not name a *width* on the command line, **fmt** adjusts each line to be approximately 72 characters long.

See Also

commands, elvis

Notes

fmt is part of the **elvis** package. Users usually do not run it on its own.

fnkey — Command

Set/print function keys for the console
fnkey [*n* **[** *string* **]]**

The console keyboard of a COHERENT system includes ten programmable function keys, labeled **F1** through **F10**. Initially, these are programmed to send the escape sequences set by the **nkb** keyboard driver.

The command **fnkey** programs function key **F***n* to send *string,* where *n* is a number from one through ten. If no *string* is given, **fnkey** resets **F***n* to send nothing.

With no argument, **fnkey** prints the current string for each programmed function key.

fnkey also lets you change the default bindings for other special or function keys. See Lexicon articles **keyboard tables** and **nkb** for details.

Example

To set function key **F2** to execute the COHERENT command **date**, use the following command:

```
fnkey 2 'date
'
```

If you type **fnkey** without any arguments, it displays the binding of all function keys including the following:

```
F2:  date\n
```

Files

/dev/console

See Also

commands, keyboard, vtnkb

Diagnostics

fnkey prints

```
cannot open /dev/console
```

if you lack permission to open **/dev/console**.

fnmatch() — String Function (libc)

Match a string with a normal expression
#include <fnmatch.h>
int fnmatch(*pattern***,** *string***,** *flags***)**
const char **pattern***, ***string***; int** *flags***;**

The function **fnmatch()** checks whether the string to which *string* points matches the normal expression to which *pattern* points. A *normal expression* is one that uses wildcard characters to broaden the range of strings that it matches. For more information, see the Lexicon entry for **wildcards**.

flags is a bit map whose bits are defined in the header file **<fnmatch.h>**. **fnmatch()** recognizes any or all of following flags:

FNM_NOESCAPE
Disable recognizing the backslash as an escape character.

If this flag is not set, then prefixing a character in *pattern* with a backslash '\' matches that same character in *string*. For example, the pair '*' in *pattern* matches a literal '*' in *string*; and the pair '\\' in *pattern* matches '\' in *string*.

FNM_PATHNAME
A slash '/' in *string* matches only a slash in *pattern*. If this flag is set, then a '/' in *string* will not match a wildcard character in *pattern*.

FNM_PERIOD
A leading period '.' in *string* must be matched exactly by a period in *pattern*. If **FNM_PATHNAME** is set, then a "leading" period is one that occurs either at the beginning of *string* or immediately following a slash; if it is not set, then a "leading" period is one that appears at the beginning of *string*. If **FNM_PERIOD** is not set, then **fnmatch()** places no special restrictions on matching a period.

If *string* matches *pattern*, **fnmatch()** returns zero. If it does not match, **fnmatch()** returns **FN_NOMATCH**. If an error occurs, **fnmatch()** returns a value other than zero or **FN_NOMATCH**.

See Also

libc, pnmatch(), string.h, wildcards

fnmatch.h — Header File

Constants used with function fnmatch()
#include <fnmatch.h>

The header file **fnmatch.h** defines manifest constants used with the function **fnmatch()**.

See Also

fnmatch(), header files

fopen() — STDIO Function (libc)

Open a stream for standard I/O
#include <stdio.h>
FILE ***fopen* (*name, type***)**
char **name***, ***type***;**

fopen() allocates and initializes a **FILE** structure, or *stream*; opens or creates the file *name*; and returns a pointer to the structure for use by other STDIO routines. *name* refers to the file to be opened.

type is a string that consists of one or more of the characters "rwa", to indicate the mode of the string, as follows:

r Read; error if file not found
w Write; truncate if found, create if not found

a Append to end of file; no truncation, create if not found
r+ Read and write; no truncation, error if not found

w+ Write and read; truncate if found, create if not found
a+ Append and read; no truncation, create if not found

The modes that contain 'a' set the seek pointer to point at the end of the file; all other modes set it to point at the beginning of the file. Modes that contain '+' both read and write; however, a program must call **fseek** or **rewind** before it switches from reading to writing or vice versa.

Example

This example copies **argv[1]** to **argv[2]** using STDIO routines. It demonstrates the functions **fopen()**, **fread()**, **fwrite()**, **fclose()**, and **feof()**.

```
#include <stdio.h>
#include <stdlib.h>
/* BUFSIZ is defined in stdio.h */
char buf[BUFSIZ];

void fatal(message)
char *message;
{
      fprintf(stderr, "copy: %s\n", message);
      exit(1);
}

main(argc, argv)
int argc; char *argv[];
{
      register FILE *ifp, *ofp;
      register unsigned int n;

      if (argc != 3)
            fatal("Usage: copy source destination");
      if ((ifp = fopen(argv[1], "r")) == NULL)
            fatal("cannot open input file");
      if ((ofp = fopen(argv[2], "w")) == NULL)
            fatal("cannot open output file");

      while ((n = fread(buf, 1, BUFSIZ, ifp)) != 0) {
            if (fwrite(buf, 1, n, ofp) != n)
                  fatal("write error");
      }

      if (!feof(ifp))
            fatal("read error");
      if (fclose(ifp) == EOF || fclose(ofp) == EOF)
            fatal("cannot close");
      exit(0);
}
```

See Also

fclose(), fdopen(), freopen(), libc
ANSI Standard, §7.9.5.3
POSIX Standard, §8.1

Diagnostics

fopen() returns NULL if it cannot allocate a **FILE** structure, if the *type* string is nonsense, or if the call to **open()** or **creat()** fails.

The header file **stdio.h** defines the manifest constant **FOPEN_MAX**, which sets the maximum number of **FILE** structures that you can allocate per program, including **stdin**, **stdout**, and **stderr**. For release 4.2, **FOPEN_MAX** is set to 60.

Notes

Many operating systems recognize a 'b' modifier to the *type* argument; this indicates that the file contains binary information, and lets the operating system handle "funny characters" correctly. COHERENT has no need of such a modifier, so if you append 'b' to *type*, it will be ignored. This modifier, however, is recognized by numerous other operating systems, including MS-DOS, OS/2, and GEMDOS. If you expect to port developed code to any of these operating systems, files should append the 'b' to *type*.

for — Command

Execute commands for tokens in list
for *name* **[in** *token ...***] do** *sequence* **done**

The shell command **for** controls a loop. It assigns to the variable *name* each successive *token* in the list, and then executes the commands in the given *sequence*. If the **in** clause is omitted, **for** successively assigns *name* the value of each positional parameter to the current script ('$@'). Because the shell recognizes a reserved word only as the unquoted first word of a command, both **do** and **done** must either occur unquoted at the start of a command or be preceded by ';'.

The shell commands **break** and **continue** may be used to alter control flow within a **for** loop.

The shell executes **for** directly.

See Also

break, commands, continue, ksh, sh

for — C Keyword

Control a loop
for(*initialization***;** *endcondition***;** *modification***)**

for is a C keyword that introduces a loop. It takes three arguments, which are separated by semicolons ';'. *initialization* is executed before the loop begins. *endcondition* describes the condition that ends the loop. *modification* is a statement that modifies *variable* to control the number of iterations of the loop. For example,

```
for (i=0; i<10; i++)
```

first sets the variable **i** to zero; then it declares that the loop will continue as long as **i** remains less than ten; and finally, increments **i** by one after every iteration of the loop. This ensures that the loop will iterate exactly ten times (from **i==0** through **i==9**). The statement

```
for(;;)
```

will loop until its execution is interrupted by a **break**, **goto**, or **return** statement. Also, either or both of *initialization* and *modification* may consist of multiple statements that are separated by commas. For example,

```
for (i=0, j=0; i<10; i++, j++)
```

initializes both *i* and *j*, and increments both with each iteration of the loop.

See Also

break, C keywords, continue, while
ANSI Standard, §6.6.5.3

fork() — System Call (libc)

Create a new process
#include <unistd.h>
fork()

In the COHERENT system, many processes may be active simultaneously. **fork()** creates a new process; the new process is a duplicate of the requesting process. In practice, the new process often issues a call to execute yet another new program.

The process that issues the **fork()** call is termed the *parent* process, and the newly forked process is termed the *child* process. **fork()** returns the process id of the newly created child to the parent process, and returns zero to the child process. The parent may call **wait()** to suspend itself until the child terminates.

The following parts of the environment of a process are exactly duplicated by a **fork()** call:

- Open files and their seek positions
- Current working and root directories
- The file creation mask
- The values of all signals
- The alarm clock setting
- Code, data, and stack segments

The system normally makes a fresh copy of the code, data, and stack segments for the child process. One advantage of *shared text* processes is that they do not need to copy the code segment. It is write protected, and therefore may be shared.

Example

For examples of how to use this call, see **msgget()**, **pipe()**, and **signal()**.

See Also

alarm(), execl(), exit(), libc, sh, umask(), unistd.h, wait()
POSIX Standard, §3.1.1

Diagnostics

fork() returns -1 on failure, which usually involves insufficient system resources. On successful calls, **fork()** returns zero to the child and the process id of the child to the parent.

fortune — Command

Print randomly selected, hopefully humorous, text
/usr/games/fortune [*file* **]**

fortune prints a message that is randomly selected from the contents of a text file. **fortune** reads *file* if it is named on the command line; otherwise, it reads the default file **/usr/games/lib/fortunes**.

Files

/usr/games/lib/fortunes — Default fortunes

See Also

commands

Notes

The fortunes included in **/usr/games/lib/fortunes** were selected mainly because they would not give offense to anyone. We encourage you to update this file with your favorite witticisms.

.forward — System Administration

Set a forwarding address for mail

The file **$HOME/.forward** lets you automatically redirect your incoming mail. You can redirect mail to one or more other users, who are located either on your local machine or on a remote site; or you can redirect your mail to one or more programs on your local machine, for further processing; or both. As you can see, this feature of the mail system included with COHERENT gives you great flexibility in processing your mail.

For example, you may wish to forward to another user any mail that is sent to the superuser root, so you can handle it immediately. (If you don't, it will languish in **root**'s mailbox until someone logs in as **root**, which may not happen for days.) To forward **root**'s mail to user **fred**, place the following line into file **/.forward**:

```
fred
```

Thereafter, whenever mail is sent to **root**, it will be forwarded automatically to user **fred**.

For another example, suppose that you are going on vacation, and you want your mail to be forwarded both to user **fred** and to user **anne**. To do so, insert the following instruction into file **$HOME/.forward**:

```
fred, anne
```

Thereafter, the route-mail program **rmail** will send a copy of every mail message you receive to **fred** and to **anne**. Please note that **rmail** will *not* insert a copy into your mailbox: if you forward your mail, you will not see it.

For another example, suppose that user **fred** has an account on each of two systems: one called **acme.com** and the other **zenith.com**. Suppose, further, that he logs into **acme.com** regularly, but he logs into **zenith.com** only now and again. This user probably would want to route any mail he receives on **zenith.com** to **acme.com**, so he will see it immediately. To do so, he would put the following instruction into file **$HOME/.forward** on **zenith.com**:

```
fred@acme.com
```

Thereafter, all mail sent to address **fred@zenith.com** will be forwarded automatically to **fred@acme.com**.

Please note that it is illegal to include in **.forward** the name of the user whose mail is being forwarded, because it causes an infinite loop in the mail system. For example, writing

```
fred, anne, root
```

into **root**'s **.forward** file causes any message sent to **root** to be forwarded to **fred**, **anne**, and **root**; the copy forwarded to **root** is again forwarded to **fred**, **anne**, and **root**; and so on, *ad infinitum*.

You can also embed the name of a program with your **.forward** file. All mail sent to your account will be handed to this program for processing. For example, the **elm** mailer includes a program called **filter**, which a user can program to read his mail and throw away unwanted messages. If you have installed **elm** onto your system, you can turn on **filter** by embedding the following command into file **$HOME/.forward**:

```
"|/usr/local/bin/filter"
```

Note that the command must be preceded by a '|' symbol; this is because **filter** receives its input from the standard input, which is the standard method for programs that filter text or mail. Note, too, that the entire command must be enclosed within quotation marks.

See Also

Administering COHERENT, mail [overview], smail

fpathconf() — System Call (libc)

Get a file variable by file descriptor
#include <unistd.h>
long fpathconf(*fd***,** *fd***)**
int *fd***,** *name***;**

fpathconf() returns the value of a limit or option associated with the open file whose the file descriptor is *fd*. *name* is the symbolic constant (defined in **<unistd.h>**) that represents the limit or option to be returned. The value that **fpathconf()** returns depends upon the type of file that *fd* identifies.

fpathconf() can return information about the following constants:

_PC_LINK_MAX
The maximum value of a file's link count. If *fd* identifies a directory, the value returned applies to the directory itself.

_PC_MAX_CANON
The number of bytes in a terminal's canonical input queue. Behavior is undefined if *fd* does not identify a terminal file.

_PC_MAX_INPUT
The number of bytes for which space will be available in a terminal's input queue. Behavior is undefined if *fd* does not identify a terminal file.

_PC_NAME_MAX
The number of bytes in a file name. The behavior is refined if *fd* does not identify a directory. The value returned applies to the file names within the directory.

_PC_PATH_MAX
The number of bytes in a path name. Behavior is undefined if *fd* does not identify a directory. If *fd* identifies the current working directory, **fpathconf()** returns the maximum length of a relative path name.

_PC_PIPE_BUF
The number of bytes that can be written atomically when writing to a pipe. If *fd* identifies a pipe or FIFO, the value returned applies to the FIFO itself. If *fd* identifies a directory, the value returned applies to any FIFOs that exist or can be created within that directory. If *fd* identifies any other type of file, behavior is

undefined.

_PC_CHOWN_RESTRICTED
chown() can be used only by a process with appropriate privileges, and only to change the group ID of a file to either that process's effective group ID or one of its supplementary group IDs. If *fd* identifies a directory, the value returned applies to any file, other than a directory, that exists or can be created within the directory.

_PC_NO_TRUNC
Path-name components longer than **NAME_MAX** generate an error. The behavior isundefined if *fd* does not identify a directory. The value returned applies to the file names within the directory.

_PC_VDISABLE
If this value is defined, terminal-special characters can be disabled. Behavior is undefined if *fd* does not identify a terminal file.

The value of the system limit or option that *name* specifies does not change during the lifetime of the calling process.

fpathconf() fails and returns -1 if *name* is not set to a recognized constant. It fails, returns -1, and sets **errno** to an appropriate value if either of the following is true:

- *fd* is not a valid file descriptor. **fpathconf()** sets **errno** to **EBADF**.
- *name* is an invalid value. **fpathconf()** sets **errno** to **EINVAL**.

See Also

libc, pathconf(), unistd.h
POSIX Standard, §5.7.1

fperr.h — Header File

Constants used with floating-point exception codes
#include <fperr.h>

fperr.h declares constants used by routines that handle floating-point exceptions. It also defines the error messages they use.

See Also

header files

fprintf() — STDIO Function (libc)

Print formatted output into file stream
#include <stdio.h>
int fprintf(*fp, format,* **[***arg1, …. argN***])**
FILE ******fp***; char** ******format***;**
[*data type***]** *arg1, … argN***;**

fprintf() formats and prints a string. It resembles the function **printf()**, except that it writes its output into the stream pointed to by *fp*, instead of to the standard output.

fprintf() uses the *format* to specify an output format for *arg1* through *argN*.

See **printf()** for a description of **fprintf()**'s formatting codes.

If it wrote the formatted string correctly, **fprintf()** returns the number of characters written. Otherwise, it returns a negative number.

Example

For an example of this routine, see the entry for **fscanf()**.

See Also

libc, printf(), sprintf(), vfprintf()
ANSI Standard, §7.9.6.1
POSIX Standard, §8.1

Notes

Because C does not perform type checking, it is essential that an argument match its specification. For example, if the argument is a **long** and the specification is for a **short**, **fprintf()** will peel off the first word of that **long** and present it as an **short**.

fputc() — STDIO Function (libc)

Write character into file stream
#include <stdio.h>
int fputc(*c*, *fp***)**
char *c*; **FILE** **fp*;

fputc() writes the character *c* into the file stream pointed to by *fp*. It returns *c* if *c* was written successfully.

Example

The following example uses **fputc** to write the contents of one file into another.

```
#include <stdio.h>

void fatal(message)
char *message;
{
      fprintf(stderr, "%s\n", message);
      exit(1);
}

main()
{
      FILE *fp, *fout;
      int ch;
      int infile[20];
      int outfile[20];

      printf("Enter name to copy: ");
      gets(infile);
      printf("Enter name of new file: ");
      gets(outfile);

      if ((fp = fopen(infile, "r")) == NULL)
            fatal("Cannot write input file");

      if ((fout = fopen(outfile, "w")) != NULL)
            fatal("Cannot write output file");

      while ((ch = fgetc(fp)) != EOF)
            fputc(ch, fout);
}
```

See Also

libc
ANSI Standard, §7.9.7.3
POSIX Standard, §8.1

Diagnostics

fputc() returns EOF when a write error occurs, e.g., when a disk runs out of space.

fputs() — STDIO Function (libc)

Write string into file stream
#include <stdio.h>
int fputs(*string*, *fp***)**
char **string*; **FILE** **fp*;

fputs() writes *string* into the file stream pointed to by *fp*. Unlike its cousin **puts()**, it does not append a newline character to the end of *string*.

fputs() returns a nonnegative value on success and **EOF** if a write error occurs.

Example

For an example of this function, see the entry for **freopen()**.

See Also

libc, puts()
ANSI Standard, §7.9.7.4
POSIX Standard, §8.1

fputw() — STDIO Function (libc)

Write an integer into a stream
#include <stdio.h>
int fputw(*word***,** *fp***)**
int *word***; FILE** **fp***;*

fputw() writes *word* into the file stream pointed to by *fp*, and returns the value written.

Example

For an example of this function, see the entry for **ferror()**.

See Also

fgetw(), libc

Diagnostics

fputw() returns **EOF** when an error occurs. A call to **ferror()** or **feof()** may be needed to distinguish this value from a valid end-of-file signal.

fread() — STDIO Function (libc)

Read data from file stream
#include <stdio.h>
int fread(*buffer, size, n, fp***)**
char **buffer***;* **unsigned** *size, n***; FILE** **fp***;*

fread() reads *n* items, each being *size* bytes long, from file stream *fp* into *buffer*.

Example

For an example of how to use this function, see the entry for **fopen()**.

See Also

fwrite(), libc
ANSI Standard, §7.9.8.1
POSIX Standard, §8.1

Diagnostics

fread() returns zero upon reading EOF or on error; otherwise, it returns the number of items read.

free() — General Function (libc)

Return dynamic memory to free memory pool
#include <stdlib.h>
void free(*ptr***) char** **ptr***;*

free() helps you manage the arena. It returns to the free memory pool memory that had previously been allocated by **malloc()**, **calloc()**, or **realloc()**. **free()** marks the block indicated by *ptr* as unused, so the **malloc()** search can coalesce it with contiguous free blocks. *ptr* must have been obtained from **malloc()**, **calloc()**, or **realloc()**.

Example

For an example of how to use this routine, see the entry for **malloc()**.

See Also

libc
ANSI Standard, §7.10.3.2
POSIX Standard, §8.1

Diagnostics

free() prints a message and calls **abort()** if it discovers that the arena has been corrupted. This most often occurs by storing data beyond the bounds of an allocated block.

freemem — Device

Device that indicates amount of memory that is free
/dev/freemem

/dev/freemem is the device from which you can read the system's free memory at any given moment. It has major device 0, the same as **/dev/null** and **/dev/cmos**; and minor number 12.

This non-portable device node is used exclusively for tracking the amount of free memory in the system. Its driver recognizes the system calls **open()**, **close()**, **read()**, and **ioctl()**, but not **write()**.

Example

The following program prints the amount of free memory in your system.

```
#include <fcntl.h>
#include <sys/null.h>
#include <stdlib.h>

main()
{
      FREEMEM freemem;
      int fm_fd;

      fm_fd = open("/dev/freemem", O_RDONLY);

      if (fm_fd >= 0) {
            ioctl (fm_fd, NLFREE, &freemem);
            close (fm_fd);
            printf ("Available memory: %d kilobytes\n", freemem.avail_mem);
            printf ("Free memory: %d kilobytes\n", freemem.free_mem);
      } else
            printf("Cannot open /dev/freemem\n");
}
```

See Also

device drivers, hmon, ioctl(), null

freopen() — STDIO Function (libc)

Open file stream for standard I/O
#include <stdio.h>
FILE *freopen (*name***,** *type***,** *fp***)**
char ***name, ***type**; FILE ***fp**;**

freopen() reinitializes the file stream *fp*. It closes the file currently associated with it, opens or creates the file *name*, and returns a pointer to the structure for use by other STDIO routines. *name* names a file.

type is a string that consists of one or more of the characters "**rwa**" (for, respectively, read, write, and append) to indicate the mode of the stream. For further discussion of the *type* variable, see the entry for **fopen()**. **freopen()** differs from **fopen()** only in that *fp* specifies the stream to be used. Any stream previously associated with *fp* is closed by **fclose()**. **freopen()** is usually used to change the meaning of **stdin**, **stdout**, or **stderr**.

Example

This example, called **match.c**, looks in **argv[2]** for the pattern given by **argv[1]**. If the pattern is found, the line that contains the pattern is written into the file **argv[3]** or to **stdout**.

```
#include <stdio.h>
#define MAXLINE 128
char buffer[MAXLINE];
```

```
void fatal(message)
char *message;
{
      fprintf(stderr, "match: %s\n", message);
      exit(1);
}

main(argc,argv)
int argc; char *argv[];
{
      FILE *fpin, *fpout;

      if (argc != 3 && argc != 4)
            fatal("Usage: match pattern infile [outfile]");
      if ((fpin = fopen(argv[2], "r")) == NULL)
            fatal("Cannot open input file");

      fpout = stdout;
      if (argc == 4)
            if ((fpout = freopen(argv[3], "w", stdout)) == NULL)
                  fatal("Cannot open output file");

      while (fgets(buffer, MAXLINE, fpin) != NULL) {
            if (pnmatch(buffer, argv[1], 1))
                  fputs(buffer, stdout);
      }
      exit(0);
}
```

See Also

fopen(), libc
ANSI Standard, §7.9.5.4
POSIX Standard, §8.1

Diagnostics

freopen() returns NULL if the *type* string is nonsense or if the file cannot be opened. Currently, only 20 **FILE** structures can be allocated per program, including **stdin**, **stdout**, and **stderr**.

frexp() — General Function (libc)

Separate fraction and exponent
#include <math.h>
double frexp(*real***,** *ep***)**
double *real***; int** ******ep***;**

frexp() breaks double-precision floating point numbers into fraction and exponent. It returns the fraction *m* of its *real* argument, such that 0.5 <= *m* < 1 or *m*=0, and stores the binary exponent *e* in the location pointed to by *ep*. These numbers satisfy the equation *real* = *m* * 2*e*.

Example

This example prompts for a number, then uses **frexp()** to break it into its fraction and exponent.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
      double real, fraction;
      int ep;

      char string[64];

      for (;;) {
            printf("Enter number: ");
            if (gets(string) == NULL)
                  break;
```

```
            fraction = frexp(real, &ep);
            printf("%lf is the fraction of %lf\n",
                  fraction, real);
            printf("%d is the binary exponent of %lf\n",
                  ep, real);
      }
}
```

See Also

atof(), ceil(), fabs(), floor(), ldexp(), libc, modf()
ANSI Standard, §7.5.4.3
POSIX Standard, §8.1

from — Command

Generate list of numbers, for use in loop
from *start* **to** *stop* **[by** *incr* **]**

from prints a list of integers on the standard output, one per line. It prints beginning with *start*, and then prints successive numbers incrementing by *incr* (default, one) the previous number. It continues until the generated value matches or exceeds *stop*. Each of *start*, *stop*, and optional *incr* is a decimal integer with an optional leading '-' sign.

Typical uses of **from** include generating a file of numbers and generating a loop index for the shell. The following example creates special files for eight terminal ports:

```
for i in `from 0 to 7`
do
      /etc/mknod /dev/hs0$i c 7 $i
done
```

See Also

commands, ksh, sh

Diagnostics

from prints an error message if the generated list is empty.

fscanf() — STDIO Function (libc)

Format input from a file stream
#include <stdio.h>
int fscanf(*fp, format, arg1, … argN***)**
FILE **fp***;* **char** **format***;*
[*data type***]** **arg1, … *argN***;**

fscanf() reads the file stream pointed to by *fp*, and uses the string *format* to format the arguments *arg1* through *argN*, each of which must point to a variable of the appropriate data type.

fscanf() returns either the number of arguments matched, or EOF if no arguments matched.

For more information on **fscanf()**'s conversion codes, see **scanf()**.

Example

The following example uses **fprintf()** to write some data into a file, and then reads it back using **fscanf()**.

```
#include <stdio.h>

main ()
{
      FILE *fp;
      char let[4];

      /* open file into write/read mode */
      if ((fp = fopen("tmpfile", "wr")) == NULL) {
            printf("Cannot open 'tmpfile'\n");
            exit(1);
      }
```

```
      /* write a string of chars into file */
      fprintf(fp, "1234");

      /* move file pointer back to beginning of file */
      rewind(fp);

      /* read and print data from file */
      fscanf(fp, "%c %c %c %c",
            &let[0], &let[1], &let[2], &let[3]);
      printf("%c %c %c %c\n",
            let[3], let[2], let[1], let[0]);
}
```

See Also

libc, scanf(), sscanf()
ANSI Standard, §7.9.6.2
POSIX Standard, §8.1

Notes

Because C does not perform type checking, it is essential that an argument match its specification. For that reason, **fscanf()** is best used only to process data that you are certain are in the correct data format, such as data previously written out with **fprintf()**.

fsck — Command

Check and repair file systems interactively
/etc/fsck [**-fnqsSy**] [**-t** *tempfile*] [*filesystem …*]

fsck checks and interactively repairs file systems. If all is well, **fsck** merely prints the number of files used, the number of blocks used, and the number of blocks that are free. If the file system is found to be inconsistent in one of the aspects outlined below, **fsck** asks whether it should fix the inconsistency and waits for you to reply **yes** or **no**.

The following file system aspects are checked for consistency by **fsck**:

- If a block is claimed by more than one i-node, by an i-node and the free list, or more than once in the free list.
- Whether an i-node or the free list claims blocks beyond the file system's range.
- Link counts that are incorrect.
- Whether the directory size is not aligned for 16 bytes.
- Whether the i-node format is correct.
- Whether any blocks are not accounted for.
- Whether a file points to an unallocated i-node.
- Whether a file's i-node number is out of range.
- Whether the super block refers to more than 65,536 i-nodes.
- Whether the super block assigned more blocks to the i-nodes than the system contains.
- Whether the format of the free block list is correct.
- Whether the counts of the total free blocks and the free i-nodes are correct.

fsck prints a warning message when a file name is null, has an embedded slash '/', is not null-padded, or if '.' or '..' files do not have the correct i-node numbers.

When **fsck** repairs a file system, any file that is orphaned (that is, allocated but not referenced) is deleted if it is empty, or copied to a directory called **lost+found**, with its i-node number as its name. The directory **lost+found** must exist in the root of the file system being checked before **fsck** is executed, and it must have room for new entries without requiring that new blocks be allocated.

fsck recognizes the following options:

-f Fast check. **fsck** only checks whether a block has been claimed by more than one i-node, by an i-node and the free list, or more than once in the free list. If necessary, **fsck** will reconstruct the free list.

-n No option: a default reply of **no** is given to all of **fsck**'s questions.

-q Quiet option: run quietly. **fsck** automatically removes all unreferenced pipes, and automatically fixes list counts in the super block and the free list. File-name warning messages are suppressed, but **fsck** still prints the number of files used, the number of blocks used, and the number of blocks that remain free.

-s Sort the free lists, both free blocks and free i-nodes, based on the interleave number. This is useful in reducing fragmentation of a file system. This option ignores mounted file systems.

-S Same as **-s**, except that it also works on mounted file systems. Not recommended for the faint of heart.

-t Name the temporary file used by **fsck**.

-y Yes option: a default reply of **yes** is given to all of **fsck**'s questions.

If you do not name a file system in **fsck**'s command line, **fsck** checks the file systems named in the file **/etc/checklist**.

Files

/etc/checklist

See Also

clri, commands, icheck, ncheck, ram, sync, umount

Diagnostics

The following describes **fsck**'s error messages and questions. The error messages fall into two categories: *warnings*, which describe something possibly wrong with a file; and *fatals*, which indicate that something has gone wrong with a file system, or with **fsck** itself, with which **fsck** cannot cope. Each question describes the condition in question; here, it is followed by advice on what is probably the correct response.

Bad action in virtual system *(fatal)*
Bad block *number*, i-number = *number (warning)*
Number Bad blocks in Free List *(warning)*

Bad/Dup blocks in *i-node type file name* (Clear i-node) [yes/no] *(question)*
The given i-node contains bad or duplicately referenced blocks. You are asked if you would like to clear the i-node completely. If you answer yes, then the file will be lost forever.

Bad entry in block *number* in directory *name/i-node (warning)*

Bad Free List (SALVAGE) [yes/no] *(question)*
fsck is asking if you want it to salvage the free list automatically. This is almost certainly a good thing to do.

Bad or Dup blocks in *directory/file* (Remove) [yes/no] *(question)*
The given file's i-node references bad or duplicately referenced blocks. **fsck** is asking if you wish to remove *file* from the directory.

Bad Super Block: *number (warning)*
Number Blocks missing *(warning)*

***** BOOT Coherent (NO SYNC!) ***** *(message)*
Do as the message says: reboot COHERENT *without* running the command **sync**.

Cannot close Ram Disk Close /dev/rram1close *(fatal)*
Cannot create temp file *name (fatal)*
Cannot open Ram Disk Close /dev/rram1close *(fatal)*
Cannot open read/write Ram Disk /dev/rram1 *(fatal)*

Can not Read: Blk num: *number* (CONTINUE) [yes/no] *(question)*
The given action could not be performed. If you choose to not continue, **fsck** will abort. If you choose to continue, the results may be unpredictable.

LEXICON

Can not Seek: Blk num: *number* (CONTINUE) [yes/no] *(question)*
The given action could not be performed. If you choose to not continue, **fsck** will abort. If you choose to continue, the results may be unpredictable.

Can not Write: Blk num: *number* (CONTINUE) [yes/no] *(question)*
The given action could not be performed. If you choose to not continue, **fsck** will abort. If you choose to continue, the results may be unpredictable.

Can't access ram disk /dev/rram1, use the -t option *(fatal)*
Can't malloc memory, phase 2 *(fatal)*
Can't malloc space for interleave table. Free-block list is not rebuilt. *(warning)*
Can't open: *file system (warning)*
Can't open checklist file: /etc/checklist *(fatal)*
Can't stat: *file system (warning)*
Can't stat temp file *name (fatal)*

Count = *count*, should be *count* (Adjust) [yes/no] *(question)*
The given i-node claims to have a different number of links than was actually found in the file system. You are asked if you wish to adjust the count found in the i-node. If you answer yes, then **fsck** will correct the i-node count.

Directory Misaligned i-number = *number (warning)*
Dir i-number = *number* connected. Parent was i-number = *number (warning)*
Dir i-number = *number* connected. It has bad/dup blocks. *(warning)*
Dir i-number = *number* connected. It has no .. entry. *(warning)*

Dup/Bad blocks in root i-node (Continue) [yes/no] *(question)*
The root i-node has bad or duplicate blocks. This may require a guru to fix properly. **fsck** is asking whether you want it to continue. If not, then **fsck** will abort.

Dup Block *number*, i-number = *number (warning)*
Number Dup blocks in Free List *(warning)*

DUP Table Overflow (Continue) [yes/no] *(question)*
The table of duplicately referenced disk blocks has overflowed. You can continue with the **fsck** (as best as it is able), or abort.

Embedded slashes in entry in block *number* in directory *name/i-node (warning)*
Error seeking tmp file *(fatal)*
Error writing tmp file *(fatal)*
Error writing to tmp file *(fatal)*

Excessive Bad Blocks i-number = *number* (Continue) [yes/no] *(question)*
The specified i-node references an excessive number of bad blocks. You can continue with the **fsck** (at the next i-node), or abort.

Excessive Dup Blocks i-number = *number* (Continue) [yes/no] *(question)*
The specified i-node references an excessive number of duplicate blocks. You can continue with the **fsck** (at the next i-node), or abort.

Excessive *bad/dup* blocks in free list (Continue) [yes/no] *(question)*
This indicates that there are excessive bad or duplicately referenced blocks in the free list off of the superblock. This is a very bad condition. You should choose to continue, which will fall to phase 6 to salvage the free list. If you answer no, then **fsck** will abort.

Expect roughly *number* missing blocks next time fsck is run as a result of i-nodes being cleared. *(message)*

file is not a block or character device; OK? [yes/no]: *(question)*
You are attempting to **fsck** a file that is not a block or character device. If you are certain it is a file system, then answer yes to continue.

File System Read-Only (NO WRITE) *(fatal)*
***** File System *system* was modified ***** *(message)*
Number files *number* blocks *number* free *(message)*
Fixblock error. *(fatal)*

Free Block count wrong in superblock. (FIX) [yes/no] *(question)*
The free block count in the superblock is incorrect. You should allow **fsck** to repair it unless you are a guru and have reason to believe that **fsck** should not use the redundancy in the file system (via all previously reported messages) to repair this crucial piece of data in the superblock.

Free i-node count wrong in superblock. (FIX) [yes/no] *(question)*
The free i-node count in the superblock is incorrect. You should allow **fsck** to repair it unless you are a guru and have reason to believe that **fsck** should not use the redundancy in the file system (via all previously reported messages) to repair this crucial piece of data in the superblock.

Inconsistent . entry in block *number* in directory *name/i-node (warning)*
Inconsistent .. entry in block *number* in directory *name/i-node (warning)*
i-number = *number* is in a bad inode block. *(warning)*

I-number is out of range I=*file name* (Remove) [yes/no] *(question)*
file has an i-node number that is out of range. **fsck** is asking if you wish to remove the stated file (which, after all, does not exist).

I-node *number* is a multiply referenced directory i-node. *(warning)*
internal linktable corruption. *(fatal)*
Invalid interleave factors in superblock. Default free-block list spacing assumed. *(warning)*
Invalid Response *(fatal)*
Link count discrepancy in *i-node type file name*
file system mounted on *point* as of *time (message)*
Name too long. *(warning)*
Non null padded entry in block *number* in directory *name/i-node (warning)*
Null name entry in block *number* in directory *name/i-node (warning)*

Out of Range Block number: *number* (CONTINUE) [yes/no] *(question)*
The given action could not be performed. If you choose to not continue, **fsck** will abort. If you choose to continue, the results may be unpredictable.

Possible Directory Size Error i-number = *number (warning)*
Possible File Size Error i-number = *number (warning)*
Possible file system on ram disk /dev/rram1, use the -t option *(fatal)*
Ram disk close /dev/rram1close not mknoded properly *(fatal)*
Ram disk /dev/rram1 not mknoded properly *(fatal)*

Root i-node is not a directory (FIX) [yes/no] *(question)*
The root i-node must be a directory. **fsck** is asking whether you wish to fix this. If not, then **fsck** will abort.

Root i-node is unallocated. Terminating *(fatal)*
Size check: fsize *blocks* isize *first non-i-node block (warning)*
Sorry. No lost+found directory. *(warning)*
Sorry. No space in lost+found directory. *(warning)*
Temp File must not be on file system to fsck *(fatal)*
Too many file systems in checklist file: /etc/checklist *(fatal)*
Too large free block count *(warning)*
Too large free i-node count *(warning)*
Too many links in i-node *number (fatal)*
Tried to checkpath i-node *number* which is not dir. *(fatal)*

Unallocated *file* (Remove) [yes/no] *(question)*
file's i-node is unallocated. **fsck** is asking if you wish to remove the stated file (which, after all, does not exist).

Unknown File Type i-number = *number* (Clear) [yes/no]: *(question)*
The mode field in the specified i-node is unknown. If you wish, you can clear the named i-node.

file system unmounted. Last mounted on *point. (message)*

Unref Dir *name* (Reconnect) [yes/no] *(question)*
The given directory's i-node is unreferenced. You are asked if you would like to reconnect the stated directory. If you answer yes, then the directory will be reconnected in directory **/lost+found** in the given file system. If not, it will remain unreferenced and you will be asked later if you would like to remove it.

LEXICON

Unref *i-node type file name* (Reconnect) [yes/no] *(question)*
The given i-node is unreferenced. **fsck** is asking if you wish to reconnect it to the stated file. If you answer yes, then the file will be reconnected in directory **/lost+found** in the given file system. If not, it will remain unreferenced and you will be asked later if you would like to remove it.

Unref *i-node type file name* (Clear i-node) [yes/no] *(question)*
The given i-node is unreferenced. **fsck** asks if you wish to clear the i-node completely. If you answer yes, the file is lost forever. You have already decided not to reconnect it, so there seems to be no reason to keep it anyway.

Notes

The correction of file systems almost always involves the destruction of data.

You should run **fsck** only when the COHERENT system is in single-user mode.

fsck cannot modify a file system during its work. This rule was adopted to prevent **fsck** from attempting to modify a corrupt file system, and so making matters worse. However, this means that **fsck** cannot change the size of directory **lost+found**. Thus, if more files are detached from the file system than **lost+found** can hold, **fsck** must delete them outright. If you are running an application that uses large numbers of transient files (e.g., a news system), you should increase the size of **lost+found** so that it has a fighting chance of holding all detached files that **fsck** finds. To do so, use the command **/etc/mklost+found**. For details, see its entry in the Lexicon.

fseek() — STDIO Function (libc)

Seek on file stream
#include <stdio.h>
int fseek(*fp, where, how***)**
FILE **fp***;* **long** *where*; **int** *how*;

fseek() changes where the next read or write operation will occur within the file stream *fp*. It handles any effects the seek routine might have had on the internal buffering strategies of the system. The arguments *where* and *how* specify the desired seek position. *where* indicates the new seek position in the file. It is measured from the start of the file if *how* equals **SEEK_SET** (zero), from the current seek position if *how* equals **SEEK_CUR** (one), and from the end of the file if *how* equals two **SEEK_END** (two).

fseek() differs from its cousin **lseek()** in that **lseek()** is a COHERENT system call and takes a file number, whereas **fseek()** is a STDIO function and takes a **FILE** pointer.

Example

This example opens file **argv[1]** and prints its last **argv[2]** characters (default, 100). It demonstrates the functions **fseek()**, **ftell()**, and **fclose()**.

```
#include <stdio.h>
extern long atol();

void fatal(message)
char *message;
{
      fprintf(stderr, "tail: %s\n", message);
      exit(1);
}

main(argc, argv)
int argc; char *argv[];
{
      register FILE *ifp;
      register int c;
      long nchars, size;

      if (argc < 2 || argc > 3)
            fatal("Usage: tail file [ nchars ]");
      nchars = (argc == 3) ? atol(argv[2]) : 100L;
```

```
      if ((ifp = fopen(argv[1], "r")) == NULL)
            fatal("cannot open input file");
      /* Seek to end */
      if (fseek(ifp, 0L, 2) == -1)
            fatal("seek error");

      /* Find current size */
      size = ftell(ifp);
      size = (size < nchars) ? 0L : size - nchars;

      /* Seek to point */
      if (fseek(ifp, size, 0) == -1)
            fatal("seek error");
      while ((c = getc(ifp)) != EOF)
            /* Copy rest to stdout */
            putchar(c);
      if (fclose(ifp) == EOF)
            fatal("cannot close");
      exit(0);
}
```

See Also

fsetpos(), ftell(), libc, lseek()
ANSI Standard, §7.9.9.2
POSIX Standard, §8.1

Diagnostics

For any diagnostic error, **fseek()** returns -1; otherwise, it returns zero. If **fseek()** goes beyond the end of the file, it will not return an error message until the corresponding read or write is performed.

fsetpos() — STDIO Function (libc)

Set file-position indicator
#include <stdio.h>
int
fsetpos(*fp***,** *position***)**
FILE **fp***; **fpos_t** **position***;*

fsetpos() resets the file-position indicator. *fp* points to the file stream whose indicator is being reset. *position* is a value that had been returned by an earlier call to **fgetpos()**. It is of type **fpos_t**, which is defined in the header **stdio.h**.

Like the related function **fseek()**, **fsetpos()** clears the end-of-file indicator and undoes the effects of a previous call to **ungetc()**. The next operation on *fp* may read or write data.

fsetpos() returns zero if all goes well. If an error occurred, it returns nonzero and sets **errno** to an appropriate value.

Example

For an example of this function, see **fgetpos()**.

See Also

fgetpos(), fseek(), ftell(), libc, rewind()
ANSI Standard, §7.9.9.3

Notes

The ANSI Standard designed **fsetpos()** to be used with files whose file position cannot be represented within a **long**. Under COHERENT, it behaves the same as **fseek()**.

If you wish to use **fsetpos()**, you should first call the function **fgetpos()** to obtain value of the file-position indicator.

You can also use the functions **ftell()** and **fset()**, respectively, to read and set the file-position indicator. However, code that uses these function may not be portable to operating systems other than COHERENT or UNIX.

fstat() — System Call (libc)

Find attributes of an open file
#include <sys/stat.h>
int fstat(*fd, statptr***)**
int *fd***; struct stat** ******statptr***;**

fstat() examines the attributes of an open file. *fd* is the descriptor of the open file or pipe you wish to examine. *statptr* points to a structure of type **stat**, which is defined in the header file **<stat.h>**; **fstat()** writes into it the attributes of the file or pipe to which *fd* points, including protection information, file type, and file size.

fstat() returns zero if all goes well. If an error occurs (e.g., *fd* is not found or *statptr* is invalid), it returns -1.

Example

For an example of how to use this function, see the Lexicon entry for **pipe()**.

See Also

chmod(), chown(), libc, ls, open(), stat(), stat.h
POSIX Standard, §5.6.2

Notes

fstat() differs from the related function **stat()** mainly in that it accesses a file through its descriptor, which was returned by a successful call to **open()**, whereas **stat()** takes the file's path name and opens the file itself before it checks its status.

fstatfs() — System Call (libc)

Get information about a file system
#include <sys/types.h>
#include <sys/statfs.h>
int fstatfs (*filedes***,** *buffer***,** *length***,** *fstype***)**
int *filedes***;**
struct statfs ******buffer***;**
int *length***,** *fstype***;**

The COHERENT system call **fstatfs()** returns information about a file system, either mounted or unmounted.

buffer points to a structure of type **statfs**, which contains the following members:

```
short f_fstyp;              /* type of the file system */
short f_bsize;              /* block size */
short f_frsize;             /* fragment size */
long f_blocks;              /* number of blocks in the file system */
long f_bfree;               /* number of free blocks */
long f_files;               /* number of file nodes */
long f_ffree;               /* number of free file nodes */
char f_fname[6];            /* name of the volume */
char f_fpack[6];            /* name of the pack */
```

length is the length of the area into which **fstatfs()** can write its output. Always set this to **sizeof(struct statfs)**.

filedes and *fstype* identify the file system. If the file system is unmounted, then *filedes* should give the file descriptor for the device by which the file system is accessed, as returned by a call to **creat()**, **dup()**, **open()**, or **pipe()**; and *fstype* contains the type of the file system. If the file system is mounted, then *filedes* should give the file descriptor of a file on the file system in question, and *fstype* must be set to zero.

fstatfs() returns zero if all went well. If something went wrong, it returns -1 and sets **errno** to an appropriate value.

See Also

libc, mkfs, statfs(), statfs.h, types.h, ustat()

ft — Device Driver

Floppy-tape driver
/dev/ft

The device driver **ft** supports floppy-tape drives. It has major number 4. Minor-number assignments are documented in the header file **/usr/include/sys/ft.h**.

ft works with QIC-40 and QIC-80 drives from Colorado, Archive, Mountain, and Summit. It offers the following features:

- It uses the bad-block bitmap that is written into the first two 32-kilobyte segments of tape at format time.
- It uses standard QIC-40/QIC-80 Reed-Solomon error-correcting code (ECC). This technique uses three of every 32 blocks for error checking. A tape block is one kilobyte long.
- It supports no-rewind-on-close. This feature permits you to concatenate several archives onto a single tape cartridge.
- It performs auto-configuration for users who do not know if their drives use soft select A or soft select B, or hard select on unit 0, 1, 2, or 3, with manual override.
- It lets you configure the size of the tape buffer, from 64 through 4,064 kilobytes.
- It reads from and writes to buffer space rather than going to tape whenever possible.
- It works with partially formatted tapes. Some formatting utilities let you select the number of tape tracks to format, in case you do not want to take the time to format an entire cartridge.
- It recognizes both 205-foot and 307.5-foot tapes.
- It works with the COHERENT command **tape** with the following arguments: **rewind**, **retension**, **seek**, **status**, and **tell**.

Please note that release 1.0 of **ft** has the following limitations:

- It does not format tapes. For now, we suggest that you buy pre-formatted tapes, or use formatting utilities available under other operating systems.
- It does not support the QIC-80 formats for MS-DOS or UUCP file systems on tape. These features do not need to be part of the device driver, and can be implemented by user-level applications.
- It does not perform data compression, as documented in QIC-122. Other forms of data compression are presently available under COHERENT, such as the **-z** option supported by the tape-archive command **gtar**.
- The device driver is character-only: there is no corresponding block device for floppy tape.
- It does not support 1,100-foot tapes. Although the QIC-80 standard mentions this length, it is not in common use.
- You cannot access a floppy-disk drive from COHERENT while a floppy-tape drive is in use. Likewise, if a floppy disk is in use — for example, if it is mounted — you cannot access the floppy-tape drive.
- Although a QIC-80 drive can read a tape that was formatted for QIC-40, it cannot write to such a tape. The cartridge must be reformatted for QIC-80 before a QIC-80 drive can write to it.

See Also

device drivers, fd, ftbad, gnucpio, gtar, tape

Notes

ft reports any error that may affect integrity of the data. If the same block number appears repeatedly in **ft**'s warning messages, it is a problem on the tape and the block should be in the bad block list. Because the Reed-Solomon ECC used in **ft** allows the physical medium to spoil up to three of every 32 one-kilobyte blocks yet recover all data, your data set may still be recoverable despite these errors; but you should consider using the command **ftbad** to add such blocks to your cartridge's list of bad blocks before you again write data onto that cartridge.

The message:

```
Get Reference Burst Failed
```

can occur if you attempt to back up to an unformatted tape, or one who's format is unrecognizable. If a backup fails with this message, try using another, formatted cartridge.

Systems with a very slow CPU (e.g., a 16-megahertz 80386SX) may have trouble running **ft** in multi-user mode. The reason is that floppy-tape hardware does not have much intelligence built into it, so the driver must consume many CPU cycles. In such instances, we suggest that you back up your system while in single-user mode (which is a good idea in any case).

ftbad — Command

Manipulate bad-block list on a floppy-tape cartridge
ftbad [-rw] [*device***]**

The command **ftbad** lets you manipulate the list of bad blocks on a floppy-tape cartridge. It recognizes the following options:

-r Read the list of bad blocks from floppy-tape cartridge, and write them to the standard-output device. The output will appear something like the following:

```
557
1033
89640
```

-w Read a list of bad blocks from the standard-input device, and write it onto the floppy-tape cartridge.

device
The floppy-tape device to manipulate. If you do not name a device on **ftbad**'s command line, by default it uses **/dev/ft**, which rewinds the tape upon close. For a list of tape devices that you can use the Lexicon entry for **tape**.

Example

To modify the bad block list for a cartridge, do the following:

- First, use the command:

```
ftbad -r > badlist
```

This reads the list of bad blocks and writes it into file **badlist**.

- Second, edit **badlist**. Each line in this file will name only one bad block, in decimal notation.
- Finally, write the edited list back onto the tape cartridge with the command:

```
ftbad -w < badlist
```

See Also

commands, ft, tape

Notes

Do not change the bad block list of a tape that contains data you wish to retrieve. You should use **ftbad** only when you see repeated I/O errors at the same block on a tape and wish to mark that block as being bad before you reuse the tape. *Caveat utilitor!*

ftell() — STDIO Function (libc)

Return current position of file pointer
#include <stdio.h>
long ftell(*fp***) FILE** ***fp*;**

ftell() returns the current position of the seek pointer. Like its cousin **fseek()**, **ftell()** takes into account any buffering that is associated with the stream *fp*.

Example

For an example of how to use this function, see the entry for **fseek()**.

See Also

fgetpos(), fseek(), libc, lseek(), rewind()
ANSI Standard, §7.9.9.4
POSIX Standard, §8.1

ftime() — System Call (libc)

Get the current time from the operating system
#include <sys/timeb.h>
int ftime(*tbp***)**
struct timeb **tbp***;*

ftime() fills the structure **timeb**, which is pointed to *tbp*, with COHERENT's representation of the current time. Header file **timeb.h** defines **timeb** as follows:

```
struct timeb {
      time_t time;
      unsigned short millitm;
      short timezone;
      short dstflag;
}
```

The member **time** is the number of seconds since January 1, 1970, 0h00m00s GMT. **millitm** is a count of milliseconds. **timezone** and **dstflag** are obsolete; they have been replaced by the environmental variable **TIMEZONE**.

ftime() does not return a meaningful value.

See Also

date, libc, time, timeb.h, TIMEZONE, types.h

Notes

The ANSI Standard eliminates **ftime()** from the set of standard time functions. COHERENT includes it only to support existing software. Users are well advised to modify their time routines to eliminate **ftime()**.

ftok() — General Function (libc)

Generate keys for interprocess communication
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(*filename***,** *procid***)**
char **filename***;*
char *procid***;**

The COHERENT system implements three methods by which one process can communicate with another: semaphores, messages, and shared memory. In each case, a process must use a key of type **key_t** (which is defined in header file **<sys/types.h>**) to identify itself.

One problem is that each process generates its own key, by its own method. Therefore, two processes could independently generate the same key, which could create serious problems for interprocess communication.

The function **ftok()** generates keys for processes that perform interprocess communication. *filename* is the full path name of a file. This can be the full path name of the file in which the program resides on disk. The file named in *filename* must exist and be accessible for the system call **stat()**, or **ftok()** will fail. *procid* is a one-character identifier with which this process distinguishes itself from all other processes that are pegged to *filename*. How a process generates *procid* is up to the program itself.

For example, the program **myproc** can generate a unique key for itself with the call:

```
key_t mykey;
mykey = ftok("/usr/bin/myproc", 'A');
```

Note the following caveats:

- Because **ftok()** generates its key from a file's i-node major and minor numbers rather than its name, it generates the same key for two files that are linked. For example, if files **/usr/henry/foo** and **/usr/henry/bar** are linked to each other, then the calls

  ```
  ftok("/usr/henry/foo", 'A');
  ```

  and

  ```
  ftok("/usr/henry/bar", 'A');
  ```

will generate the same key.

- If the file named by *filename* is destroyed and then recreated, the call to **ftok()** generates a different key than it did before *filename* was destroyed.
- If the file named by *filename* does not exist, **ftok()** returns **(key_t) -1**.

Example

For an example of this function, see the entry for **msgget()**.

See Also

ipc.h, libc, msgget(), semget(), shmget()

function — Definition

A **function** is the C term for a portion of code that is named, can be invoked by name, and that performs a task. Many functions can accept data in the form of arguments, modify the data, and return a value to the statement that invoked it.

See Also

data types, library, portability, Programming COHERENT

fwrite() — STDIO Function (libc)

Write into file stream
#include <stdio.h>
int fwrite(*buffer, size, n, fp***)**
char **buffer***; **unsigned** *size, n*; **FILE** **fp*;

fwrite() writes *n* items, each of *size* bytes, from *buffer* into the file stream pointed to by *fp*.

Example

For an example of how to use this function, see the entry for **fopen()**.

See Also

fread(), libc
ANSI Standard, §7.9.8.2
POSIX Standard, §8.1

Diagnostics

fwrite() normally returns the number of items written. If an error occurs, the returned value will not be the same as *n*.

fwtable — Command

Build font-width table
fwtable [-ptv] [*infile* **[** *outfile* **]]**

For the typesetting program **troff** to use a font, it must know the width of each character in the font, and it must know how to tell the printer to select the font. All of this information is built into a *font-width table*, which **troff** reads when you run it.

COHERENT comes with font-width tables for a selected set of fonts: for a handful of scalable fonts that are included with standard PostScript cartridges, for a few bit-mapped fonts, and for some fonts that are built into the Hewlett-Packard LaserJet III. For a list of the font-width tables that are included with COHERENT, and for further information on how to manage fonts, see the Lexicon entry for **troff**.

The command **fwtable** can read a font, and build a new font-width table for it. It reads the font information from *infile* (or the standard input) and writes a font-width table for the font to *outfile* (or the standard output). It can understand fonts in the following formats:

- PCL (Printer Control Language) bitmap fonts, which have the suffix **.usp**.
- Fonts that are built into the Hewlett-Packard LaserJet III and IV, which have the suffix **.tfm**.
- AFM (Adobe Font Metric) descriptions of PostScript fonts, which have the suffix **.afm**.

fwtable recognizes the following command-line options:

-p *infile* is an AFM (Adobe Font Metric) description for a PostScript font. By default, **fwtable** assumes that *infile* is a a bit-mapped soft font (that is, a font with the suffix **.usp**).

Please note that if the AFM font you will be using is downloadable rather than built into a cartridge, you must also use the command **PSfont** to "cook" that font's **.pfb** file into downloadable form. For more information, see the Lexicon entry **PSfont**.

-t *infile* is a Hewlett-Packard **.tfm** file, which describes a font that is built into the Hewlett-Packard LaserJet III, rather than a bit-mapped soft font.

-v Print a brief font description to the standard error file.

Files

/usr/lib/roff/troff_pcl/fwt/ — Directory for PCL font-width tables
/usr/lib/roff/troff_ps/fwt/ — Directory for PostScript font-width tables

See Also

commands, hpr, PSfont, troff

Notes

fwtable does not understand Intellifont scalable fonts, or TrueType fonts.

gawk — Command

Pattern-scanning and -processing language
gawk [*POSIX or GNU style options* **] -f** *program-file* **[--]** *file ...*
gawk [*POSIX or GNU style options* **] [--]** *program-text file ...*

gawk is the GNU Project's implementation of the AWK programming language. It conforms to the definition of the language in the POSIX Standard 1003.2 *Command Language and Utilities Standard*. This version in turn is based on the description in *The AWK Programming Language*, by Aho, Kernighan, and Weinberger, with the additional features defined in the System V Release 4 version of **awk**. **gawk** also provides some GNU-specific extensions.

The command line consists of options to **gawk** itself, the AWK program text (if not supplied via the options **-f** or **--file**), and values to be made available in the predefined AWK variables **ARGC** and **ARGV**.

Command-line Options

gawk options may be either the traditional POSIX one-letter options, or the GNU style long options. POSIX Standard-style options begin with a single '-', whereas GNU long options begin with "--". GNU-style long options are provided for both GNU-specific features and for POSIX mandated features. Other implementations of the AWK language are likely to only accept the traditional one-letter options.

Following the POSIX Standard, **gawk**-specific options are supplied via arguments to the **-W** option. Multiple **-W** options may be supplied, or multiple arguments may be supplied together if they are separated by commas, or enclosed in quotation marks and separated by white space. Case is ignored in arguments to the **-W** option. Each **-W** option has a corresponding GNU style long option, as detailed below.

gawk recognizes the following command-line options:

-F *fs*
--field-separator=*fs*
Use *fs* for the input field separator (the value of the predefined variable **FS**).

-v *variable***=***value*
--assign=*variable***=***value*
Assign *value* to *variable* before executing the program. *value* is available to the **BEGIN** block of an AWK program.

-f *program-file*
--file=*program-file*
Read the AWK program's source from file *program-file*, instead of from the first command-line argument. The **awk** command line can contain more than one **-f** or **--file** options.

-W compat
--compat
Run in compatibility mode. In compatibility mode, **gawk** behaves identically to UNIX **awk**; it recognizes none of the GNU-specific extensions are recognized. These extensions are described below.

-W copyleft
-W copyright
--copyleft
--copyright
Print the short version of the GNU copyright information message on the standard error.

-W help

-W usage
--help
--usage Print a relatively short summary of the available options on the standard error.

-W lint
--lint Provide warnings about constructs that are dubious or non-portable to other implementations of AWK.

-W posix
--posix This turns on compatibility mode, with the following additional restrictions:

- The '\x' escape sequences are not recognized.
- The synonym **func** for the keyword function is not recognized.
- The operators "**" and "**=" cannot be used in place of '^' and "^=".

-W source=*program-text*

--source=*program-text*
Use *program-text* as the AWK program's source code. This option allows the easy intermixing of library functions (used via the options **-f** and **--file**) with source code entered on the command line. It is intended primarily for medium to large AWK programs used in shell scripts. The **-W source=** form of this option uses the rest of the command line argument for *program-text*; no other options to **-W** will be recognized in the same argument.

-W version
--version
Print version information for this particular copy of **gawk** on the standard error. This is useful mainly for knowing if your copy of **gawk** is up to date with what the Free Software Foundation is distributing.

-- Signal the end of options. This is useful to allow further arguments to the AWK program itself to start with a '-'. This is mainly for consistency with the argument parsing convention used by most other POSIX Standard programs.

All other options are flagged as illegal and ignored.

AWK Program Execution

An AWK program consists of a sequence of pattern/action statements, plus optional function definitions:

```
pattern { action statements }
function name(parameter list) { statements }
```

gawk first reads the program source from the program file (or files) if specified, or from the first non-option argument on the command line. The option **-f** may be used multiple times on the command line. **gawk** reads the program text as if all the program-files had been concatenated. This is useful for building libraries of AWK functions, without having to include them in each new AWK program that uses them. To use a library function in a file from a program typed in on the command line, specify **/dev/tty** as one of the program files, type your program, and end it with a **<ctrl-D>**.

The environment variable **AWKPATH** specifies a search path to use when finding source files named with the option **-f**. If this variable does not exist, the default path is:

```
.:/usr/lib/awk:/usr/local/lib/awk
```

If a file name given to the **-f** option contains a '/' character, **gawk** does not perform a path search.

gawk executes AWK programs in the following order:

1. **gawk** compiles the program into an internal form.
2. All variable assignments specified via the **-v** option are performed.
3. **gawk** executes the code in the **BEGIN** block (or blocks), should there be any.
4. **gawk** then proceeds to read each file named in the **ARGV** array. If no files are named on the command line, **gawk** reads the standard input.

If a file name on the command line has the form *variable***=***value*, **gawk** treats it as a variable assignment, and assigns *value* to *variable*. (This happens after every **BEGIN** block has been run.) Command-line assignment of variables is most useful when you wish to assign values dynamically to the variables AWK uses to control how

LEXICON

input is broken into fields and records. It is also useful for controlling the state of program execution if multiple passes are needed over a single data file.

If the value of a particular element of **ARGV** is empty (""), **gawk** skips it.

For each line in the input, **gawk** tests to see if it matches any pattern in the AWK program. It tests the patterns in the order they occur in the program. For each pattern that the line matches, **awk** executes action associtaed with that pattern.

Finally, after all the input is exhausted, **gawk** executes the code in every **END** block.

Variables and Fields

AWK variables are dynamic: they come into existence when they are first used. Their values are floating-point numbers, strings, or both, depending upon how they are used. AWK also has one dimensional arrays: multiply dimensioned arrays can be simulated. Several pre-defined variables are set as a program runs; these are described as needed and summarized below.

Fields

As it reads a line of input, **gawk** splits that line into fields. The variable **FS** defines how fields are separated:

- If **FS** is a single character, fields are separated by that character.
- If **FS** is longer than one character, it must be a regular expression. In this case, the value of variable **IGNORECASE** (described below) also affects how fields are split. **FS** is a regular expression.
- In the special case that **FS** is a single space character, fields are separated by a number of space characters or tab characters.

If variable **FIELDWIDTHS** is set to a space-separated list of numbers, each field is expected to have a fixed width: **gawk** splits up the record using the specified widths, and ignores the value of **FS**. Assigning a new value to **FS** overrides the use of **FIELDWIDTHS**, and restores the default behavior.

Each field in the input line can be referenced by its position: **$1**, **$2**, and so on. $0 is the whole line.

The value of a field may be assigned to as well. Fields need not be referenced by constants. For example, the AWK expression

```
n = 5
print $n
```

prints the fifth field in the input line. The variable **NF** holds the total number of fields in the input line.

References to non-existent fields (i.e., fields after **$NF**) produce the null string. However, assigning to a nonexistent field (e.g., **$(NF+2) = 5**) increases the value of **NF**; creates any intervening fields, with the null string as the value of each; and causes the value of **$0** to be recomputed, with the fields being separated by the value of **OFS**.

Built-in Variables

The following variables are built into AWK:

ARGC The number of command-line arguments. Note that this does not include options to **gawk**, or the program source.

ARGIND
The index in **ARGV** of the file now being processed.

ARGV Array of command-line arguments. The array is indexed from through to **ARGC** minus one. Dynamically changing the contents of **ARGV** can control the files used for data.

CONVFMT
The conversion format for numbers — by default, “%.6g”.

ENVIRON
An array containing the values of the current environment. The array is indexed by the environment variables, each element being the value of that variable (e.g., **ENVIRON["HOME"]** might be **/u/arnold**). Changing this array does not affect the environment seen by programs which gawk spawns via redirection or the function **system()**. (This may change in a future version of **gawk**.)

ERRNO
If a system error occurs while performing redirection for **getline()**, during a read for **getline()**, or during a close, **ERRNO** contains a string describing the error.

FIELDWIDTHS
A white-space separated list of fieldwidths. When set, **gawk** parses the input into fields of fixed width, instead of using the value of the variable **FS** as the field separator. The fixed field-width facility is still experimental; expect the semantics to change as **gawk** evolves over time.

FILENAME
The name of the current input file. If no files are specified on the command line, the value of **FILENAME** is '-'. However, **FILENAME** is undefined within the **BEGIN** block.

FNR The number of the record within the current input file that is now being processed.

FS The input field separator. By default, this is a blank.

IGNORECASE
Tell **gawk**'s pattern-matching features to ignore the case when they compare text with a pattern. When **IGNORECASE** is set to a nonzero function, the following features of **gawk** are affected:

- Pattern-matching within rules
- Fieldsplitting with **FS**.
- Regular expression matching with '~' and "!~".
- The operation of the pre-defined **gawk** functions **gsub()**, **index()**, **match()**, **split()**, and **sub()**.

Thus, if **IGNORECASE** is not equal to zero, pattern

```
/aB/
```

matches all of the following:

```
ab
aB
Ab
AB
```

As with all AWK variables, the initial value of **IGNORECASE** is zero, so all regular expression operations are normally case-sensitive.

NF The number of fields in the current input record.

NR The total number of input records seen so far.

OFMT The output format for numbers — by default "%.6g".

OFS The output-field separator — by default a space character.

ORS The output-record separator — by default a newline.

RS The input record separator — by default a newline. **RS** is exceptional in that only the first character of its string value is used to separate records. (This will probably change in a future release of **gawk**.) If **RS** is set to the null string, then records are separated by blank lines. When **RS** is set to the null string, then the newline character always acts as a field separator, in addition to whatever value **FS** may have.

RSTART
The index of the first character matched by the **gawk** function **match()**: zero if no match.

RLENGTH
The length of the string matched by **match()**: -1 if no match.

SUBSEP
The character used to separate multiple subscripts in array elements — by default " 34".

Arrays

Arrays are subscripted with an expression between square brackets ('[' and ']'). If the expression is an expression

list (expr, expr ...)

LEXICON

then the array subscript is a string consisting of the concatenation of the (string) value of each expression, separated by the value of the variablen **SUBSEP**. This facility simulates multi-dimensional arrays. For example,

```
i = "A" ; j = "B" ; k = "C"
x[i, j, k] = "hello, world\n"
```

assigns the string

```
"hello, world\n"
```

to the element of the array x which is indexed by the string

```
"A\034B\034C".
```

All arrays in AWK are associative, i.e., indexed by string values.

The special operator in may be used in an **if** or **while** statement to see if an array has an index that consists of a particular value:

if (val in array)
print array[val]

If the array has multiple subscripts, use **(i, j)** in array.

You can also use the construct **in** within a **for** loop to iterate through all the elements of an array.

An element can be deleted from an array using the statement **delete**.

Variable Typing And Conversion

Variables and fields can be floating-point numbers, strings, or both. How the value of a variable is interpreted depends upon its context. If a variable or field is used in a numeric expression, **gawk** treats it as a number; if used as a string, **gawk** treats it as a string. To force a variable to be treated as a number, add zero to it; to force it to be treated as a string, concatenate it with the null string.

When a string must be converted to a number, the conversion is accomplished by the library function **atof()**. A number is converted to a string by using the value of **CONVFMT** as a format string for **sprintf()**, with the numeric value of the variable as the argument. However, even though all numbers in AWK are floating point, integral values are always converted as integers. Thus, given

```
CONVFMT = "%2.2f"
a = 12
b = a ""
```

the variable **b** has a value of 12, not 12.00.

gawk performs comparisons as follows:

- If two variables are numeric, they are compared numerically.
- If one value is numeric and the other has a string value that is a "numeric string," then comparisons are also done numerically.
- Otherwise, the numeric value is converted to a string and a string comparison is performed.

Two strings are compared, of course, as strings. According to the POSIX Standard, even if two strings are numeric strings, a numeric comparison is performed; however, this is clearly incorrect, and **gawk** does not do this.

Uninitialized variables have the numeric value zero and the string value "" (the null, or empty, string).

Patterns and Actions

AWK is a line-oriented language: the pattern comes first, and then the action. Action statements are enclosed in '{' and '}'. Either the pattern may be missing, or the action may be missing, but (of course) not both. If the pattern is missing, AWK executes the action for every line of input. A missing action is equivalent to

```
{ print }
```

which prints the entire line.

Comments begin with the character '#', and continue to the end of the line. Blank lines can be used to separate statements. Normally, a statement ends with a newline; however, this is not the case for lines ending in any of the following characters:

```
,     {     ?     :     &&    ||
```

Lines that end in one of the above characters have their statements automatically continued on the following line. In other cases, a line can be continued by ending it with a '\', in which case the newline will be ignored.

Multiple statements may be put on one line by separating them with a ';'. This applies to both the statements within the action part of a pattern/action pair (the usual case), and to the pattern/action statements themselves.

Patterns

AWK patterns may be one of the following:

```
BEGIN
END
/regular expression/
relational expression
pattern && pattern
pattern || pattern
pattern ? pattern : pattern
(pattern)
! pattern
pattern1, pattern2
```

BEGIN and **END** are two special patterns that are not tested against the input. The action parts of all **BEGIN** patterns are merged as if all the statements had been written in a single **BEGIN** block. They are executed before any of the input is read. Likewise, **gawk** merges all the **END** patterns and executes them when all the input is exhausted (or when an **exit** statement is executed). **BEGIN** and **END** patterns cannot be combined with other patterns in pattern expressions. **BEGIN** and **END** patterns must have action parts.

For

```
/regular expression/
```

patterns, the associated statement is executed for each input line that matches the regular expression. Regular expressions are the same as those described in the Lexicon entry for the shell **sh**, and are summarized below.

A relational expression may use any of the operators defined below in the section on actions. These generally test whether certain fields match certain regular expressions.

The operators **&&**, **||**, and **!** are logical AND, logical OR, and logical NOT, respectively, as in C. They do short-circuit evaluation, also as in C, and are used for combining more primitive pattern expressions. As in most languages, parentheses may be used to change the order of evaluation.

The operator **?:** is like the same operator in C. If the first pattern is true then the pattern used for testing is the second pattern, otherwise it is the third. Only one of the second and third patterns is evaluated.

The

pattern1, *pattern2*

form of an expression is called a "range pattern". It matches all input records starting with a line that matches *pattern1*, and continues until it reads a record that matches *pattern2*, inclusive. It does not combine with any other sort of pattern expression.

Regular Expressions

Regular expressions are the extended kind found in the shell **sh**. They are composed of characters, as follows:

c Match the non-meta-character *c*.

*****c* Match the literal character *c*.

. Match any character except newline.

^ Match the beginning of a line or a string.

$ Match the end of a line or a string.

[*abc...***]** Character class: Match any of the characters *abc...*.

LEXICON

[^*abc...***]**
Negated character class: Match any character except *abc...* and newline.

*r1***|***r2* Alternation: match either *r1* or *r2*.

r1r2 Concatenation: Match *r1*, then *r2*.

*r***+** Match one or more *r*'s.

*r** Match zero or more *r*'s.

*r***?** Match zero or one *r*'s.

(*r***)** Grouping: match *r*.

The escape sequences that are valid in string constants (see below) are also legal in regular expressions.

Actions

Action statements are enclosed in braces, '{' and '}'. Action statements consist of the usual assignment, conditional, and looping statements found in most languages. The operators, control statements, and input/output statements available are patterned after those in C.

Operators

The following gives AWK's operators, in order of increasing precedence:

= += -=
*= /= %= ^= = (assignment)

Both absolute assignment (*var* **=** *value*) and operator-assignment (the other forms) are supported.

This has the form

expr1 **? expr2 :** *expr3*

If *expr1* is true, the value of the expression is *expr2*; otherwise it is *expr3*. Only one of *expr2* and *expr3* is evaluated.

| | — logical OR
&& — logical AND
~ — Regular expression match
!~ — Negated match
Do not use a constant regular expression (**/foo/**) on the left-hand side of a '~' or '!~'. Only use one on the right-hand side. The expression

/foo/ ~ *exp*

has the same meaning as:

(($0 ~ /foo/) ~ *exp***)**

This is usually not what was intended.

< >
<= >=
!=
== The regular relational operators.

<blank>
String concatenation.

+
- Addition and subtraction.

*
/
% Multiplication, division, and modulus.

+

-
! Unary plus, unary minus, and logical negation.

^ Exponentiation. The operator '**' may also be used, and '**=' for the assignment operator.

++
-- Increment and decrement, both prefix and suffix.

$ Field reference.

Control Statements

The control statements are as follows:

if (*condition***)** *statement* **[else** *statement* **]**
while (*condition***)** *statement*
do *statement* **while (***condition***)**
for (*expr1***;** *expr2***;** *expr3***)** *statement*
for (*var in array***)** *statement*
break
continue
delete array[*index***]**
exit [*expression*]
{ *statements* }

I/O Statements

AWK recognizes the following input/output statements:

close(*filename***)**
Close file or pipe.

getline Set **$0** from next input record. This statement also sets the built-in variables **NF**, **NR**, and **FNR**.

getline <*file*
Set **$0** from next record of file. This statement also sets the built-in variable **NF**.

getline *var*
Set *var* from next input record. This statment also sets the built-in variables **NF** and **FNR**.

getline *var* **<***file*
Set *var* from next record of *file*.

next Stop processing the current input record. The next input record is read and processing starts over with the first pattern in the AWK program. If the end of the input data is reached, each **END** block is executed.

next *file*
Stop processing the current input file. The next input record read comes from the next input file. **FILENAME** is updated, **FNR** is reset to one, and processing starts over with the first pattern in the AWK program. If the end of the input data is reached, every **END** is executed.

print Print the current record.

print *expr-list*
Print each expression in *expr-list*.

print *expr-list* **>***file*
Print expressions on file.

printf *fmt***,** *expr-list*
Format and print.

printf *fmt***,** *expr-list* **>***file*
Format and print into *file*.

system(*cmd-line***)**
Execute the command *cmd-line*, and return its exit status.

Other input/output redirections are also allowed. For **print** and **printf**, **>>***file* appends output onto *file*, whereas a '**|**' command writes onto a pipe. Likewise, command **|getline pipes into getline**. **getline returns zero when it reads EOF, and -1 if an error occurs.**

LEXICON

The printf Statement

The AWK statement **printf** and the function **sprintf()** (see below) accept the following conversion specification formats:

%c An ASCII character. If the argument used for **%c** is numeric, it is treated as a character and printed. Otherwise, the argument is assumed to be a string, and the only first character of that string is printed.

%d A decimal number (the integer part).

%i Just like **%d**.

%e A floating-point number of the form **[-]***d***.***dddddd***E[+-]***dd*.

%f A floating-point number of the form **[-]***ddd***.***dddddd*.

%g Use 'e' or 'f' conversion, whichever is shorter, with nonsignificant zeros suppressed.

%o An unsigned octal number (again, an integer).

%s A character string.

%x An unsigned hexadecimal number (an integer).

%X Like **%x**, but using "ABCDEF" instead of "abcdef".

%% A single '%' character; no argument is converted.

There are optional, additional parameters that may lie between the '%' and the control letter:

- The expression should be left-justified within its field.

width The field should be padded to this width. If the number has a leading zero, then the field will be padded with zeroes; otherwise, it is padded with blanks.

.*prec* A number that indicates the maximum width of the string or digit to the right of the decimal point.

The dynamic width and precision capabilities of the ANSI C **printf()** routines are supported. A '*' in place of either the width or precision specification causes AWK to take its value from the argument list to **printf** or **sprintf()**.

Special File Names

When doing I/O redirection from either **print** or **printf** into a file, or via **getline** from a file, **gawk** recognizes certain special file names internally. These file names allow access to open file descriptors inherited from **gawk**'s parent process (usually the shell). Other special file names provide access information about the running **gawk** process. The file names are as follows:

/dev/pid
Reading this file returns the identfier of the current process, in decimal, terminated with a newline.

/dev/ppid
Reading this file returns the identifier of the current's process's parent, in decimal, terminated with a newline.

/dev/pgrpid
Reading this file returns the current process's group identifier, in decimal, terminated with a newline.

/dev/user
Reading this file returns a single record terminated with a newline. The fields are separated with blanks. **$1** is the value of the system call **getuid()**; **$2** is the value of the system call geteuid() ; **$3** is the value of the system call **getgid()**; and **$4** is the value of the system call **getegid()**. If there are any additional fields, they are the group identifiers returned by **getgroups()**.

/dev/stdin
The standard input.

/dev/stdout
The standard output.

/dev/stderr
The standard error output.

/dev/fd/*n*
The file associated with the open-file descriptor *n*.

These are particularly useful for error messages. For example, these files let you use the statement

```
print "You blew it!" > "/dev/stderr"
```

where otherwise you would have had to say:

```
print "You blew it!" | "cat 1>&2"
```

These file names may also be used on the command line to name data files.

Numeric Functions

AWK contains the following pre-defined arithmetic functions:

atan2(*y***, *x*)**
Return the arctangent of *y*/*x*, in radians.

cos(*expr***)**
Returns the cosine, in radians.

exp(*expr***)**
The exponential function.

int(*expr***)**
Truncate to integer.

log(*expr***)**
The natural-logarithm function.

rand() Returns a random number between zero and one.

sin(*expr***)**
Return the sine in radians.

sqrt(*expr***)**
The square-root function.

srand(*expr***)**
Use *expr* as a new seed for the random number generator. If no *expr* is provided, the time of day will be used. The return value is the previous seed for the random number generator.

String Functions

AWK has the following pre-defined string functions:

gsub(*r***, *s*, *t*)**
For each substring matching the regular expression *r* in the string *t*, substitute the string *s* and return the number of substitutions. If *t* is not supplied, use **$0**.

index(*s***, *t*)**
Return the index of the string *t* in the string *s*, or zero if *t* is not present.

length(s)
Return the length of the string *s*, or the length of **$0** if *s* is not supplied.

match(*s***, *r*)**
Return the position in *s* where the regular expression *r* occurs, or zero if *r* is not present, and set the values of **RSTART** and **RLENGTH**.

split(*s***, *a*, *r*)**
Split the string *s* into the array *a* on the regular expression *r*, and return the number of fields. If *r* is omitted, use **FS** instead.

sprintf(*fmt***, *expr-list*)**
Print *expr-list* according to *fmt*, and return the resulting string.

sub(*r***, *s*, *t*)**
Just like **gsub()**, but only the first matching substring is replaced.

LEXICON

substr(*s***, *i*, *n*)**
Return the *n*-character substring of *s* starting at *i*. If *n* is omitted, the rest of *s* is used.

tolower(*str***)**
Return a copy of the string *str*, with all the upper-case characters in *str* translated to their corresponding lower-case counterparts. Non-alphabetic characters are left unchanged.

toupper(*str***)**
Return a copy of the string *str*, with all the lower-case characters in *str* translated to their corresponding upper-case counterparts. Non-alphabetic characters are left unchanged.

Time Functions

Because one of the primary uses of AWK programs is processing log files that contain time stamp information, **gawk** provides the following two functions for obtaining time stamps and formatting them.

systime()
Return the current time of day as the number of seconds since 00:00:00 hours on January 1, 1970 GMT.

strftime(*format***, *timestamp*)**
Format *timestamp* according to the specification within *format*. *timestamp* should be of the same form as returned by **systime()**. If *timestamp* is missing, the current time of day is used. See the Lexicon entry for **strftime()** for the format conversions that are guaranteed to be available.

String Constants

String constants in AWK are sequences of characters enclosed between quotation marks "". Within a string, the following escape sequences are recognized:

****	Literal backslash
\a	The BEL character
\b	Backspace
\f	Form-feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	vertical tab.
\x*XX*	Character with hexadecimal value *XX*
*****OOO*	Character represented by octal digits *OOO*
*****c*	The literal character *c*

The escape sequences may also be used within constant regular expressions (e.g., **/[\t\f\n\r\v]/** matches whitespace characters).

Functions

AWK defines a function as follows:

*function name***(***parameter list***) { *statements* }**

AWK executes a function when it is called from within the action part of a regular *pattern***/***action* statement. The parameters supplied in the function call are used to instantiate the formal parameters declared within the function. Arrays are passed by reference, other variables are passed by value.

Because functions were not originally part of the AWK language, the provision for local variables is rather clumsy: they are declared as extra parameters in the parameter list. The convention is to separate local variables from real parameters by extra spaces in the parameter list. For example:

```
function f(p, q, a, b) { # a & b are local
      .....
}

/abc/ { ...
; f(1, 2) ; ...
}
```

The left parenthesis in a function call is required to immediately follow the function name, without any intervening white space. This is to avoid a syntactic ambiguity with the concatenation operator. This restriction does not apply to the built-in functions listed above.

Functions may call each other and may be recursive. Function parameters used as local variables are initialized to the null string and the number zero upon function invocation.

The word **func** may be used in place of **function**.

Examples

Print and sort the login names of every user on your system:

```
BEGIN { FS = ":" }
{ print $1 | "sort" }
```

Count lines in a file:

```
{ nlines++ }
END { print nlines }
```

Precede each line by its number in the file:

```
{ print FNR, $0 }
```

Concatenate and line number (a variation on a theme):

```
{ print NR, $0 }
```

Compatibility

A primary goal for **gawk** is compatibility with the POSIX Standard, as well as with the latest version of UNIX **awk**. To this end, **gawk** incorporates the following user-visible features that are not described in the AWK book, but are part of **awk** in System V Release 4, and are in the POSIX Standard:

- The option **-v** for assigning variables before program execution starts is new. The book indicates that command line variable assignment happens when awk would otherwise open the argument as a file, which is after the **BEGIN** block is executed. However, in earlier implementations, when such an assignment appeared before any file names, the assignment would happen before the **BEGIN** block was run. Applications came to depend on this "feature." When **awk** was changed to match its documentation, this option was added to accomodate applications that depended upon the old behavior. (This feature was agreed upon by both the AT&T and GNU developers.)
- The option **-W** for implementation specific features is from the POSIX Standard.
- When processing arguments, gawk uses the special option "--" to signal the end of arguments, and warns about, but otherwise ignores, undefined options.
- The AWK book does not define the return value of **srand()**. The System V Release 4 version of UNIX awk (and the POSIX Standard standard) has it return the seed it was using, to allow keeping track of random number sequences. Therefore, **srand()** in **gawk** also returns its current seed.
- Other new features include the following: use of multiple **-f** options (from MKS **awk**); the **ENVIRON** array; the escape sequences **\a** and **\v** (done originally in **gawk** and fed back into AT&T's); the built-in functions **tolower()** and **toupper()** (from AT&T); and the ANSI-C conversion specifications in **printf** (done first in AT&T's version).

GNU Extensions

gawk has some extensions to POSIX Standard **awk**. They are described in this section. All the extensions described here can be disabled by invoking **gawk** with the command-line option **-W compat**. The following features of **gawk** are not available in POSIX Standard **awk**:

- The escape sequence **\x**.
- The functions **systime()** and **strftime()**.
- The special-file names available for I/O redirection.
- The variables **ARGIND** and **ERRNO** are not special.
- The variable **IGNORECASE** and its side-effects are not available.
- The variable **FIELDWIDTHS** and fixed-width field splitting.

LEXICON

- No path search is performed for files named via the option **-f**. Therefore, the environmental variable **AWKPATH** is not special.
- The use of next file to abandon processing of the current input file.

The AWK book does not define the return value of the function **close()**. **gawk**'s **close()** returns the value from **fclose()** or **pclose()** when closing a file or pipe, respectively. When **gawk** is invoked with the option **-W compat**, if the *fs* argument to option **-F** is 't', then **FS** will be set to the tab character. Because this is a rather ugly special case, it is not the default behavior. This behavior also does not occur if **-Wposix** has been specified.

Historical Features

There are two features of historical AWK implementations that **gawk** supports. First, it is possible to call the **length()** built-in function not only with no argument, but even without parentheses! Thus

```
a = length
```

is the same as either of

```
a = length()
a = length($0)
```

This feature is marked as "deprecated" in the POSIX Standard standard, and **gawk** will issue a warning about its use if option **-Wlint** is specified on the command line.

The other feature is the use of the **continue** statement outside the body of a **while**, **for**, or **do** loop. Traditional AWK implementations have treated such usage as equivalent to the next statement. **gawk** supports this usage if **-Wposix** has not been specified.

See Also

awk, commands, Programming COHERENT
Introduction to the awk Language, tutorial.
Aho, Alfred V.; Kernighan, Brian W.; Weinberger, Peter J.: *The AWK Programming Language.* Englewood Cliffs, NJ, Addison-Wesley, Inc., 1988 (ISBN 0-201-07981-X).
The GAWK Manual, ed 0.15. Boston, The Free Software Foundation, 1993.

Notes

The option **-F** option is not necessary given the command line variable assignment feature; it remains only for backwards compatibility.

If your system actually has support for **/dev/fd** and the associated **/dev/stdin**, **/dev/stdout**, and **/dev/stderr** files, you may get different output from **gawk** than you would get on a system without those files. When **gawk** interprets these files internally, it synchronizes output to the standard output with output to **/dev/stdout**, while on a system with those files, the output is actually to different open files. *Caveat utilitor.*

This man page documents gawk, version 2.15. Please note that with this version, **gawk** no longer recognizes the command-line options **-c**, **-V**, **-C**, **-a**, and **-e** that had been recognized by version 2.11.

The original version of UNIX awk was designed and implemented by Alfred Aho, Peter Weinberger, and Brian Kernighan of AT&T Bell Laboratories. Brian Kernighan continues to maintain and enhance it.

Paul Rubin and Jay Fenlason, of the Free Software Foundation, wrote **gawk** to be compatible with the original version of awk distributed in UNIX version 7. John Woods contributed a number of bug fixes. David Trueman, with contributions from Arnold Robbins, made **gawk** compatible with the new version of UNIX awk.

Brian Kernighan of AT&T Bell Laboratories provided valuable assistance during testing and debugging. The authors thank him.

Finally, please note that **gawk** and its associated documentation (including this manual page) is protected by the Free Software Foundation's "copyleft". For details on your rights and obligations, see the file **COPYING** in the source code for **gawk**, which is available through the Mark Williams BBS and other public-domain systems.

gcd() — Multiple-Precision Mathematics (libmp)

Set variable to greatest common divisor
#include <mprec.h>
void gcd(*a, b, c***)**
mint **a, *b, *c;**

gcd() sets *c* to the greatest common divisor of *a* and *b*.

See Also

libmp

gcvt() — General Function (libc)

Convert floating-point numbers to strings
char *
gcvt(*d***,** *prec***,** *buffer***)**
double *d***; int** *prec***; char** **buffer***;*

gcvt() converts floating-point number *d* into a NUL-terminated string. Its operation resembles that of **printf()**'s operator **%g**.

Argument *prec* gives the precision of the string i.e., the number of numerals to the right of the decimal point. Unlike its cousins **ecvt()** and **fcvt()**, **gcvt()** uses a buffer that is defined by the caller. *buffer* must point to a buffer large enough to hold the result; 64 characters will always be sufficient. When generating its output, **gcvt()** mimics **fcvt()** if possible. Otherwise, it mimics **ecvt()**.

gcvt returns *buffer*.

Example

For an example of this function, see the entry for **ecvt()**.

See Also

libc

gdbm.h — Header File

Header file for GDBM routines
#include <gdbm.h>

Header file **<gdbm.h>** declares functions, data types, and global variables used by the GDBM set of routines:

gdbm_close() . . . . . Close a GDBM data base
gdbm_delete(). . . . . Delete a record from a GDBM data base
gdbm_exists() . . . . . Check whether a GDBM data base contains a given record
gdbm_fetch() . . . . . Retrieve a record from a GDBM data base
gdbm_firstkey(). . . . . Return the first record from a GDBM data base
gdbm_nextkey() . . . . . Return the next record from a GDBM data base
gdbm_open(). . . . . Open a GDBM data base
gdbm_reorganize() . . . . . Reorganize a GDBM data base
gdbm_setopt(). . . . . Set GDBM options
gdbm_store() . . . . . Add records to a GDBM data base
gdbm_strerror(). . . . . Translate a GDBM error code into text
gdbm_sync(). . . . . Flush buffered GDBM data into its data base

This header file also defines two structures that the GDBM routines use. The first, **datum**, defines the structure of a data element, either a key or its associated data set:

```
typedef struct {
      char *dptr;
      int dsize;
} datum;
```

The other structure, **GDBM_FILE**, holds the information that the GDBM routines use to access a GDBM data base:

```
typedef struct {int dummy[10];} *GDBM_FILE;
```

Error codes are written into global variable **gdbm_errno**, and are defined in header file **<gdbmerrno.h>**.

See Also

Notes

For a statement of copyright and permissions on this header file, see the Lexicon entry for **libgdbm**.

LEXICON

gdbm_close() — GDBM Function (libgdbm)

Close a GDBM data base
#include <gdbm.h>
void gdbm_close (*database***)**
GDBM_FILE *database***;**

Function **gdbm_close()** closes the data base to which *database* points. *database* must have been returned by a call to **gdbm_open()**.

See Also

Notes

If *database* were opened into mode **GDBM_FAST**, **gdbm_close()** automatically calls **gdbm_sync()** to flush buffered data into the data base before it closes *database*.

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_delete() — GDBM Function

Delete a record from a GDBM data base
#include <gdbm.h>
int gdbm_delete (*database***,** *key***)**
GDBM_FILE *database***;**
datum *key***;**

Function **gdbm_delete()** deletes a the record with *key* from the data base to which *database* points. *database* must have been returned by a call to **gdbm_open()**.

If all goes well, **gdbm_delete()** returns zero. It returns -1 if *database* did not contain a record with *key*, or if *database* were opened into read-only mode.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_exists() — GDBM Function (libgdbm)

Check whether a GDBM data base contains a given record
#include <gdbm.h>
int gdbm_exists(*database***,** *key***)**
GDBM_FILE *database***;**
datum *key***;**

Function **gdbm_exists()** checks whether the GDBM data base to which *database* points contains a record with the key to which *key* points. *database* must have been returned by a call to **gdbm_open()**.

If *database* contains *key*, **gdbm_exists()** returns a value other than zero; otherwise, it returns zero.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_fetch() — GDBM Function (libgdbm)

Retrieve a record from a GDBM data base
#include <gdbm.h>
datum gdbm_fetch (*database***,** *key***)**
GDBM_FILE *database***;**
datum *key***;**

Function **gdbm_fetch()** retrieves the record with *key* from the database to which *database* points. *database* must have been returned by a call to **gdbm_open()**.

gdbm_fetch() returns the record that contains *key*. If *database* does not contains such a record, **gdbm_fetch()**

returns a record whose field **dptr** is set to NULL.

gdbm_fetch() calls **malloc()** to allocate the memory to hold the data it retrieves from *database*. It is your responsibility to free this memory; to do so, call **free()** and pass it field **dptr** in the record that **gdbm_fetch()** returns.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_firstkey() — GDBM Function (libgdbm)

Return the first record from a GDBM data base
#include <gdbm.h>
datum gdbm_firstkey(*database***)**
GDBM_FILE *database***;**

Function **gdbm_firstkey()** returns the first record from the data base to which *database* points. *database* must have been returned by a call to **gdbm_open()**.

gdbm_firstkey() returns the first record within *database*. Note that that the first record is dictated by the algorithm that the GDBM routines use to hash the keys within the data base, and so may not be what you expect. If *database* is empty, **gdbm_firstkey()** returns a record whose field **dptr** is set to NULL.

gdbm_firstkey() calls **malloc()** to allocate the memory to hold the data it retrieves from *database*. It is your responsibility to free this memory; to do so, call **free()** and pass it field **dptr** in the record that **gdbm_firstkey()** returns.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_nextkey() — GDBM Function (libgdbm)

Return the next record from a GDBM data base
#include <gdbm.h>
datum gdbm_nextkey(*database***,** *key***)**
GDBM_FILE *database***;**
datum *key***;**

Function **gdbm_nextkey()** retrieves the next record from the data base to which *database* points. If *database* contains no more records, it returns a record whose field **dptr** is set to NULL.

database must have been returned by a call to **gdbm_open()**. The call to **gdbm_nextkey()** must follow a call to **gdbm_firstkey()**.

Please note that **gdbm_nextkey()** returns records in the order dictated by the algorithm with which the GDBM routines hash the data base's keys. If called within a loop, it is guaranteed to retrieve every record within *database*, although the order in which the records are retrieved may not be what you expect.

gdbm_nextkey() calls **malloc()** to allocate the memory to hold the data it retrieves from *database*. It is your responsibility to free this memory; to do so, call **free()** and pass it field **dptr** in the record that **gdbm_nextkey()** returns.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_open() — GDBM Function (libgdbm)

Open a GDBM data base
#include <gdbm.h>
GDBM_FILE gdbm_open(*database***,** *block_size***,** *read_write***,** *mode***,** *bailout***)**
char **database***;*
int *block_size*, *read_write*, *mode*;
void (*bailout***)();**

Function **gdbm_open()** opens a GDBM data base. It takes the following parameters:

database
: This gives the complete name of the data base. Please note that a data base actually consists of two files: one, called *database*.**dir**, holds the hashed index; the other, called *database*.**pag**, holds the data. The GDBM routines manage the manipulation of these files; you need not worry about them yourself. (For more details on how GDBM works, see the Lexicon entry for **libgdbm**.)

block_size
: This gives the size of a single transfer from disk to memory. **gdbm_open()** ignores this parameter is unless *database* is new. The minimum size is 512. If you set *block_size* to less than 512, the GDBM routines use a block size of **BSIZE**. (This constant gives the size of a block under COHERENT; it is set in header file **<sys/buf.h>**.

read_write
: This parameter indicates whether you are opening the data base into read mode or write mode. If a process opens *database* only to read records within it, it is called a "reader". If, however, a process can also add records to *database*, remove record from it, or modify records within it, it is called a "writer". *database* can be opened by multiple readers simultaneously, or by a single writer; it cannot be opened by multiple writers simultaneously, or by a reader and a writer simultaneously. This rule prevents a writer from modifying a data base while it is being read, and so confusing the readers; and to prevent multiple writers from "clobbering" each other's changes.

  read_write can be one of the following values:

  GDBM_READER
  : The process opening *database* is a reader.

  GDBM_WRITER
  : The process opening *database* is a writer.

  GDBM_WRCREAT
  : The process opening *database* is a writer; if the data base *database* does not exist, create it.

  GDBM_NEWDB
  : The process opening *database* is a writer; create *database* as a new data base, regardless of whether it already exists.

  GDBM_FAST
  : If this constant is OR'd onto **GDBM_WRITER**, **GDBM_WRCREAT**, or **GDBM_NEWDB**, the GDBM routines write the data base without disk-file syncronization. This speeds writing to the data base; however, if the writer dies unexpectedly, some data may be lost. To flush buffered data to disk, call function **gdbm_sync()**.

mode This is a bitwise OR of the modes into which *database* is created. For a list of the flags that can be incorporated into this argument, see the Lexicon entry **stat.h**. **gdbm_open()** ignores this argument unless *read_write* is set to **GDBM_WRCREAT** or **GDBM_NEWDB**.

bailout This points to the function that **gdbm_open()** calls should a fatal error occur. This function must take only one argument, a string that holds an error message. If you set *bailout* to NULL, the GDBM routines use a default function.

If all goes well, **gdbm_open()** returns a pointer to a record of type **GDBM_FILE**. All other GDBM functions need this record to manipulate the data base in *database*. If an error occurs, **gdbm_open()** returns NULL and sets global variable **gdbm_errno** and **errno** to appropriate values. For information on interpreting the contents of **errno**, see the Lexicon entry for **errno.h**; for information on interpreting the contents of **gdbm_errno**, see the entry for **gdbmerrno.h**.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_reorganize() — GDBM Function (libgdbm)

Reorganize a GDBM data base
#include <gdbm.h>
int gdbm_reorganize(*database***)**
GDBM_FILE *database***;**

Function **gdbm_reorganize()** reorganizes the contents of the data base to which *database* points. *database* must have been returned by a call to **gdbm_open()**.

When you delete a record from a GDBM data base, the GDBM routines to not close up the space within the data base, because doing so would make the GDBM routines unacceptably slow. Thus, if you delete many records from within a data base, its file will be much larger than it need be. In this case, you should call **gdbm_reorganize()** to close up the "holes" in it.

gdbm_reorganize() returns zero if all went well. If something went wrong, it returns a value other than zero and sets the global variables **errno** and **gdbm_errno** to appropriate values. (For information on how to interpret the contents of these variables, see the Lexicon entries for **errno.h** and **gdbmerrno.h**).

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_setopt() — GDBM Function (libgdbm)

Set GDBM options
#include <gdbm.h>
int gdbm_setopt(*database***,** *option***,** *value***,** *size***)**
GDBM_FILE *database***;**
int *option***, ****value***,** *size***;**

Function **gdbm_setopt()** sets an option on an open GDBM data base. You should call **gdbm_setopt()** after you call **gdbm_open()**, but before you read the data base or write to it.

database points to the data base being manipulated; it must have been returned by a call to **gdbm_open()**.

value is the value to which *option* is being set. It is specified as a pointer to an integer.

option specifies the option to set, as follows:

GDBM_CACHESIZE
: Set the size of the internal bucket cache. This option may only be set once on each data base. Upon the first access to the data base, the GDBM routines by default set the cache size to 100. Set *value* to the size of the cache.

GDBM_FASTMODE
: Turn on or turn off fast mode of access. If fast mode is turned on, the GDBM routines do not synchronize disk updates with changes to the data base. This speeds modifications to the data base, but runs the risk of losing data should the "writer" process die unexpectedly. Set *value* to **TRUE** or **FALSE**.

size gives the size of the data to which *value* points.

For example, the following call sets a data base to use a cache of ten:

```
int value = 10;
ret = gdbm_setopt( dbf, GDBM_CACHESIZE, &value, sizeof(int));
```

If all goes well, **gdbm_setopt()** returns zero. If something goes wrong, it returns -1 and sets global variables **errno** and **gdbm_errno** to appropriate values. For information on how to interpret the contents of these variables, see the Lexicon entries **errno.h** and **gdbmerrno.h**.

See Also

Notes

The use of variables *value* and *size* may seem overly complex; however, this will permit the GDBM routines to recognize a larger range of options in the future.

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_store() — GDBM Function (libgdbm)

Add records to a GDBM data base
#include <gdbm.h>
int gdbm_store (*database***,** *key***,** *content***,** *flag***)**
GDBM_FILE *database***;**
datum *key***,** *content***;**
int *flag***;**

Function **gdbm_store()** writes data into a GDBM data base.

database points to the data base into which data are written. It must have been returned by a call to **gdbm_open()**.

key gives the key for the record being written. *content* gives the data to be associated with *key*.

flag indicates how data should be written; it can be either of the following:

GDBM_INSERT
Insert only. If *database* already contains a record with *key*, generate an error.

GDBM_REPLACE
Update. If *database* already contains a record with *key*, replace it with with *contents*.

If all goes well, **gdbm_store()** returns zero. If *database* was opened into read-only mode, it returns -1. If *flag* is set to **GDBM_INSERT** and *database* already contains *key*, it returns one.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_strerror() — GDBM Function (libgdbm)

Translate a GDBM error code into text
#include <gdbm.h>
#include <gdbmerror.h>
char *gdbm_strerror(*errno***)**
gdbm_error *errno***;**

Function **gdbm_strerror()** converts a GDBM error code into an error message that can be read by a human being.

errno is the error code. This usually is the global variable **gdbm_errno**, which a GDBM routine sets should an error occur while manipulating a GDBM data base.

If an error occurs, **gdbm_strerror()** returns NULL. Otherwise, it returns a pointer to the string that holds the message.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbm_sync() — GDBM Function (libgdbm)

Flush buffered GDBM data into its data base
#include <gdbm.h>
void gdbm_sync(*database***)**
GDBM_FILE *database***;**

Function **gdbm_sync()** flushes buffered data into its data base. It is the GDBM analogue of the system call **sync()**. You should call this function periodically if you are writing data into a data base that had been opened with flag **GDBM_FAST**.

database points to the data base being manipulated. It must have been returned by a call to **gdbm_open()**.

gdbm_sync() does not return until all the buffers are flushed onto disk. **gdbm_close()** automatically calls **gdbm_sync()** to flush data-base buffers before it closes a GDBM data base.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

gdbmerrno.h — Header File

Define error messages used by GDBM routines
#include <gdbmerrno.h>

Header file **<gdbmerrno.h>** defines the error codes that the GDBM routines can write into global variable **gdbm_errno**, as follows:

GDBM_NO_ERROR
: All is well.

GDBM_MALLOC_ERROR
: The GDBM routines call **malloc()** to allocate memory for each record that they retrieve from a data base. This message indicates that a call to **malloc()** failed.

GDBM_BLOCK_SIZE_ERROR
: You tried to set an illegal block size when you created a new data base.

GDBM_FILE_OPEN_ERROR
: A data-base file could not be opened, for whatever reason.

GDBM_FILE_WRITE_ERROR
: A process could not write into a data-base file. This probably indicates a problem with permissions.

GDBM_FILE_SEEK_ERROR
: A GDBM routine could not move a data-base file's seek pointer to a place where the data base's hash table indicates a given record was stored. The data base may well be corrupt; check this error seriously.

GDBM_FILE_READ_ERROR
: A process could not read a data-base file. This probably indicates a problem with permissions.

GDBM_BAD_MAGIC_NUMBER
: When the GDBM function **gdbm_open()** create a new data base, it stamps the file with a "magic number," which indicates that that file is, in fact, a GDBM data base. This error indicates that the file you're attempting to read is not a GDBM a data base.

GDBM_EMPTY_DATABASE
: The GDBM data base contains no data.

GDBM_CANT_BE_READER
: You failed in an attempt to open a GDBM data base into read mode. The data base may have already been opened into write mode.

GDBM_CANT_BE_WRITER
: You failed in an attempt to open a GDBM data base into write mode. The data base may have already been opened into write mode by another process.

GDBM_READER_CANT_DELETE
: You opened a GDBM data base into read mode, but then attempted to delete a record. This is illegal.

GDBM_READER_CANT_STORE
: You opened a GDBM data base into read mode, but then attempted to write a record into it. This is illegal.

GDBM_READER_CANT_REORGANIZE
: You opened a GDBM data base into read mode, but then attempted to reorganize it. This is illegal.

GDBM_UNKNOWN_UPDATE
: You attempted to update a record within a data base, but the data base does not contain a record with the given key.

GDBM_ITEM_NOT_FOUND
: You attempted to read a record from a data base, but the data base does not contain a record with the given key.

GDBM_REORGANIZE_FAILED
An attempted reorganization of a file failed. The data base may be corrupt.
GDBM_CANNOT_REPLACE
You attempted to write a new record into a data base, but the data base already contains a record with the given key.
GDBM_ILLEGAL_DATA
You attempted to write a record into a data base, but the record contains illegal data (e.g., the field **dptr** is NULL).
GDBM_OPT_ALREADY_SET
You called **gdbm_setopt()** to set an option on a data base, but that option is already set.
GDBM_OPT_ILLEGAL
You called **gdbm_setopt()** to set an option on a data base, but the requested option is illegal or unrecognized.

Function **gdbm_strerror()** translates a GDBM error code into a string that you can display.

See Also

Notes

For a statement of copyright and permissions on this header file, see the Lexicon entry for **libgdbm**.

getc() — STDIO Function (libc)

Read character from file stream
#include <stdio.h>
int getc(*fp***)**
FILE **fp***;*

getc() is a function that reads a character from the file stream *fp*, and returns an **int**.

Example

The following example creates a simple copy utility. It opens the first file named on the command line and copies its contents into the second file named on the command line.

```
#include <stdio.h>

void fatal(string)
char *string;
{
      printf("%s\n", string);
      exit (1);
}

main(argc, argv)
int argc; char *argv[];
{
      int foo;
      FILE *source, *dest;

      if (--argc != 2)
            fatal("Usage: copy [source][destination]");

      if ((source = fopen(argv[1], "r")) == NULL)
            fatal("Cannot open source file");
      if ((dest = fopen(argv[2], "w")) == NULL)
            fatal("Cannot open destination file");

      while ((foo = getc(source)) != EOF)
            putc(foo, dest);
}
```

See Also

fgetc(), getchar(), libc, putc()
ANSI Standard, §7.9.7.5
POSIX Standard, §8.1

Diagnostics

getc() returns EOF at end of file or on read fatal.

Notes

Because **getc()** is a macro, arguments with side effects probably will not work as expected. Also, because **getc()** is a complex macro, its use in expressions of too great a complexity may cause unforeseen difficulties. Use of the function **fgetc()** may avoid this.

getchar() — STDIO Function (libc)

Read character from standard input
#include <stdio.h>
int getchar()

getchar() reads a character from the standard input. It is equivalent to **getc(stdin)**.

Example

The following example gets one or more characters from the keyboard, and echoes them on the screen.

```
#include <stdio.h>

main()
{
      int foo;
      while ((foo = getchar()) != EOF)
            putchar(foo);
}
```

See Also

getc(), libc, putchar()
ANSI Standard, §7.9.7.6
POSIX Standard, §8.1

Diagnostics

getchar() returns **EOF** at end of file or on read error.

If you wish to receive characters from the keyboard immediately, without waiting for the enter key, see the example in the entry for **pipe()**.

getcwd() — General Function (libc)

Get current working directory name
#include <unistd.h>
char ***getcwd(***_buffer_**,** _size_**)**
char **_buffer_**;**
int _size_**;**

The current working directory is the directory from which file-name searches commence when a path name does not begin with '/'. **getcwd()** returns the name of the current working directory. It is useful for processes like spoolers and daemons, which must generate full path names for files.

If *buffer* is not NULL, **getcwd()** writes the path of the current working directory into it. The expected path name must not be longer than two characters less than *size*. In this case, **getcwd()** returns *buffer*.

If *buffer* is NULL, **getcwd() malloc()**'s *size* bytes. **getcwd()** returns a pointer to this block of memory. You can **free()** it later.

If you do not have permission to search all levels of the directory hierarchy above the current directory, **getcwd()** cannot obtain the directory name for you.

See Also

chdir(), libc, pwd, unistd.h
POSIX Standard §5.2.2

Diagnostics

getcwd() returns NULL and sets **errno** to an appropriate value if an error occurs. Possible errors include the following:

EPERM Could not read one of the parent directories.

EINVAL *size* is zero.

ENOMEM Memory could not be **malloc()**'d for the buffer.

ERANGE The path name is too long to fit into *size* minus two bytes.

Notes

If **getcwd()** fails, the working directory cannot be restored to its initial value.

getdents() — System Call (libc)

Read directory entries
#include <dirent.h>
int getdents (*fd***,** *buffer***,** *num***)**
int *fd***;**
char **buffer***;*
unsigned *num***;**

The COHERENT system call **getdents()** is one of a set of COHERENT routines that manipulate directories in a device-independent manner. It reads an entry from a directory file and writes it into a structure of type **dirent**.

fd is the file descriptor for the directory file; it must be a file descriptor opened by a call to **open()** or **dup()**. *buffer* points to the area where **getdents()** writes its output. *num* gives the size of the area pointed to by *buffer*; **getdents()** returns no more than *num* bytes of information.

getdents() writes its output into a structure of type **dirent**, which is defined in the header file **dirent.h**. It has the following structure:

```
struct dirent {
      long d_ino;
      long d_off;
      unsigned short d_reclen;
      char d_name[1];
};
```

Field **d_name** is a NUL-terminated string of indefinite length. Because this structure does not have a fixed size, you must tell **getdents()** the maximum number of bytes it can output.

getdents() automatically increments the offset pointer associated with *fd* to point to the next entry within the directory file. This lets you within a loop to read the entire contents of a directory file.

If all goes well, **getdents()** returns the number of bytes it wrote into *buffer*. It returns zero if it has reached the end of the directory file. If something went wrong (for example, you tried to use it to read a file other than a directory file), it returns -1 and sets **errno** to an appropriate value.

See Also

dirent.h, closedir(), libc, opendir(), readdir(), rewinddir(), telldir()

Notes

This system call is designed to support directory-access library routines. It should not be called by user programs.

The COHERENT implementation of **getdents()** was written by D. Gwynn.

getdtablesize() — Sockets Function (libsocket)

Get the number of files a process can open
int getdtablesize()

Function **getdtablesize()** returns the number of file descriptors (and hence, the number of files) that a process can have open at any one time. It is meant to be an operating-system independent means of determining this value; under COHERENT, it returns the value of the manifest constant **OPEN_MAX**.

See Also

libsocket

getegid() — System Call (libc)

Get effective group identifier
#include <unistd.h>
getegid()

Every process has two different versions of its *group identifier*, called the *real* group identifier and the *effective* group identifier. The group identifiers determine eligibility to access files and use system privileges. Normally, these two identifiers are identical. However, for a *set group identifier* load module (see **exec**), the real group identifier is that of the group's current group, whereas the effective group identifier is that of the load module owner. This distinction allows system programs to use files which are protected from groups that invoke the program.

getegid() returns the effective group identifier.

See Also

access, exec, geteuid(), getgid(), getuid(), libc, login, setuid(), unistd.h
POSIX Standard, §4.2.1

getenv() — General Function (libc)

Read environmental variable
#include <stdlib.h>
char ***getenv(***VARIABLE**) char** **VARIABLE**;**

A program may read variables from its *environment*. This allows the program to accept information that is specific to it. The environment consists of an array of strings, each having the form *VARIABLE*=*VALUE*. When called with the string *VARIABLE*, **getenv()** reads the environment, and returns a pointer to the string *VALUE*.

Example

This example prints the environmental variable **PATH**.

```
#include <stdio.h>
#include <stdlib.h>

main()
{
      char *env;
      extern char *getenv();

      if ((env = getenv("PATH")) == NULL) {
            printf("Sorry, cannot find PATH\n");
            exit(1);
      }
      printf("PATH = %s\n", env);
}
```

See Also

environmental variables, envp, exec, libc, putenv(), sh, stdlib.h
ANSI Standard, §7.10.4.4
POSIX Standard, §4.6.1

Diagnostics

When *VARIABLE* is not found or has no value, **getenv()** returns NULL.

geteuid() — System Call (libc)

Get effective user identifier
#include <unistd.h>
geteuid()

Every process has two different versions of its *user id*, called the *real* user id and the *effective* user id. The user ids determine eligibility to access files or employ system privileges. Normally, these two ids are identical. However, for a *set user id* load module (see **exec**), the real user id is that of the user, whereas the effective user id is that of the load module owner. This distinction allows system programs to use files which are protected from the user who invokes the program.

geteuid() returns the effective user identifier

Example

For an example of this call, see the entry for **getpwent()**.

See Also

access(), exec, getegid(), getgid(), getuid(), libc, login, setuid(), unistd.h
POSIX Standard, §4.2.1

getgid() — System Call (libc)

Get real group identifier
#include <unistd.h>
getgid()

Every process has two different versions of its *user id*, called the *real* user id and the *effective* user id. The user ids determine eligibility to access files or employ system privileges. Normally, these two ids are identical. However, for a *set user id* load module (see **exec**), the real user id is that of the user, whereas the effective user id is that of the load module owner. This distinction allows system programs to use files which are protected from the user who invokes the program.

getgid() returns the real group id.

See Also

access(), exec, getegid(), geteuid(), getuid(), libc, login, setuid(), unistd.h
POSIX Standard §4.2.1

getgrent() — General Function (libc)

Get group file information
#include <grp.h>
struct group *getgrent();

getgrent() returns the next entry from file **/etc/group**. It returns NULL if an error occurs or if the end of file is encountered.

Files

/etc/group
<grp.h>

See Also

group, initgroups(), libc

Notes

All structures and information returned are in a static area internal to **getgrent()**. Therefore, information from a previous call is overwritten by each subsequent call.

getgrgid() — General Function (libc)

Get group file information, by group id
#include <grp.h>
struct group *getgrgid(*gid***);**
int *gid***;**

getgrgid() searches file **/etc/group** for the first entry with a numerical group id of *gid*. It returns a pointer to the entry if found; it returns NULL if an error occurs or if the end of file is encountered.

Files

/etc/group
<grp.h>

See Also

group, libc
POSIX Standard, §9.2.1

Notes

All structures and information returned are in a static area internal to **getgrgid()**. Therefore, information from a previous call is overwritten by each subsequent call.

getgrnam() — General Function (libc)

Get group file information, by group name
#include <grp.h>
struct group ***getgrnam(***gname***);**
char **gname***;*

getgrnam() searches file **/etc/group** for the first entry with a group name of *gname*. It returns a pointer to the entry for *gname* if it is found; it returns NULL for any error or if the end of the file is encountered.

Files

/etc/group
<grp.h>

See Also

group, libc
POSIX Standard, §9.2.1

Notes

All structures and information returned are in a static area internal to **getgrnam()**. Therefore, information from a previous call is overwritten by each subsequent call.

getgroups() — System Call (libc)

Read the supplemental group-access list
#include <unistd.h>
int getgroups(*gidsetsize***,** *grouplist***)**
int *gidsetsize***; gid_t** ***grouplist***;**

The "supplemental group-access list" is the list of group identifiers that are used in addition to the effective group identifier when determining the level of access that a process has to a file. **getgroups()** reads the identifiers from the current process's supplemental group-access list, and writes them into the array to which *grouplist* points.

grouplist has *gidsetsize* entries, and must be large enough to contain every entry from the list. The list cannot have more than **NGROUPS_MAX** entries. If *gidsetsize* equals zero, **getgroups()** returns the number of groups to which the calling process belongs without modifying the array to which *grouplist* points.

If all goes well, **getgroups()** returns the number of supplementary-group identifiers set for the calling process. It fails and returns -1 if *gidsetsize* is greater than zero but less than the number of supplementary-group identifiers set for the calling process, or if *grouplist* points to an illegal address. In the former instance, it sets **errno** to **EINVAL**; in the latter, it sets **errno** to **EFAULT**.

See Also

libc, setgroups(), unistd.h
POSIX Standard, §4.2.3

gethostbyaddr() — Sockets Function (libsocket)

Retrieve host information by address
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
struct hostent ***gethostbyaddr(***addr***,** *len***,** *type***)**
char ***host***;**
int *len***,** *type***;**

Function **gethostbyaddr()** interrogates file **/etc/hosts** and returns information about a given host on a network.

addr gives the address at which the host's Internet address resides in memory. *length* gives the number of characters in its name. *type* gives the type of address this is. If it is anything other than type **AF_INET**, **gethostbyaddr()** returns NULL.

If it could find information about the host in question, **gethostbyaddr()** returns the address in a instance of structure **hostent**, which is defined in header file **<netdb.h>**. If it could not, it returns **NULL**.

See Also

endhostent(), gethostbyname(), libsocket, sethostent()

Page 2

gethostbyname() — Sockets Function (libsocket)

Retrieve a host IP address by name

#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
struct hostent ***gethostbyname(***host***)**
char **host***;*

Function **gethostbyname()** interrogates file **/etc/hosts** for information about a host on a network. *host* gives the address where the name of the host resides in memory.

If it could find the address of *host*, **gethostbyname()** returns the address in a instance of structure **hostent**, which is defined in header file **<netdb.h>**. If it could not, or if *host* points to a spurious host name, it returns **NULL**.

See Also

endhostent(), gethostbyaddr(), libsocket, sethostent()

gethostname() — Sockets Function (libsocket)

Get the name of the local host

#include <sys/utsname.h>
int gethostname (*name***,** *length***)**
char **name***;*
int *length***;**

Function **gethostname()** reads the name of your local host.

name points to the chunk of memory into which **gethostname()** is to write the name of the local host. *length* gives the length of that chunk of memory.

gethostname() returns -1 if it could not read the name of the local host. Otherwise, it returns zero.

See Also

libsocket

Notes

name must point to enough memory to hold the name of your local host. If it does not, the behavior of this function is undefined (and probably unwelcome). *Caveat utilitor.*

getlogin() — General Function (libc)

Get login name
#include <unistd.h>
char *getlogin()

The name corresponding to the current user id is not always the same as the name under which a user logged into the COHERENT system. For example, the user may have issued a **su** command, or there may be several login names associated with a user id. **getlogin()** returns the login name found in the file **/etc/utmp.**

In cases where **getlogin()** fails to produce a result, **getpwuid()** (described in **getpwent()**) is normally used to determine the user name for a process.

Files

/etc/utmp login names

See Also

getpwent(), getuid(), libc, su, ttyname(), unistd.h, utmp.h, who
POSIX Standard, §4.2.4

Diagnostics

getlogin() returns NULL if the login name cannot be determined.

Notes

getlogin() stores the returned name in a static area that is destroyed by subsequent calls.

getmap — Command

De-archive Usenet map articles
/usr/lib/mail/getmap [-b *batchfile***] [-m** *mapdir***] [-n** *newsgroup***] [-u** *username***] [-w** *workdir***]**

The script **getmap** de-archives Usenet map articles. The articles must be in the form of a shell archive (or "shar" file). De-archived articles are copied into directory **/usr/spool/uumaps**.

getmap recognizes the following command-line arguments:

-b *batch*
De-archive *batch*, which is a shell archive of file names. If *batch* is '-', **getmap** reads the standard input. By default, **getmap** reads file **/usr/spool/uumaps/work/batch**.

-m *mapdir*
Copy articles into *mapdir*, instead of the default directory **/usr/spool/uumaps**.

-n *newsgroups*
Read articles from *newsgroup*.

-u *user* Mail errors to *user*. If *user* is '-', write errors to the standard output. By default, **getmap** mails errors to user **postmaster**.

-w *workdir*
Keep logs and batch files in *workdir*. By default, logs and batch files are kept in directory **/usr/spool/uumaps/work**.

See Also

commands, mail [overview]

getmsg() — System Call (libc)

Get the next message from a stream
#include <stropts.h>
int getmsg (*fd***,** *ctlptr***,** *dataptr***,** *flagsp***)**
int *fd***; struct strbuf ***ctlptr***,** *dataptr***; int ***flagsp***;**

getmsg() retrieves a message from a STREAMS file, and writes it into the buffer or buffers that you specify. The message must contain a data part, a control part, or both. **getmsg()** writes each part into its own buffer, as described below. The STREAMS module that generated the message defines the semantics of each part.

fd gives the file descriptor that references the stream whose message is being retrieved. *ctlptr* and *dataptr* each point to a structure of type **strbuf**, which contains the following members:

```
int maxlen; Maximum buffer length
int len;    Length of data
void  *buf; Pointer to buffer
```

ctlptr holds the message's control part, and *dataptr* its data part. **buf** points to the buffer into which the data or control information is to be written, and **maxlen** gives the maximum number of bytes the buffer can hold. **getmsg()** initializes **len** to the number of bytes of data or control information that it actually wrote into **buf**. It sets **len** to zero if the part in question has a length of zero; and it sets **len** to -1 if the message does not contain the part in question.

flagsp points to an integer that indicates the type of messages you can receive; this is discussed in detail below.

getmsg() has special behaviors, corresponding to the settings of *ctlptr* and *dataptr*, and of the structures to which they point:

- If either *ctlptr* or *dataptr* is NULL, or if **maxlen** equals -1, **getmsg()** does not process the corresponding part of the message. The message is left on the stream head's read queue.
- If *ctlptr* or *dataptr* is not NULL, but the message does not have a corresponding part, **getmsg()** sets **len** to -1.
- If **maxlen** equals zero and there is a zero-length control or data part, **getmsg()** removes the zero-length part from the read queue and sets **len** to zero If **maxlen** equals zero and the corresponding section contains more than zero bytes of information, **getmsg()** leaves that information on the read queue and sets **len** to zero.
- If **maxlen** is less than than the corresponding part of the message (the control part for *ctlptr* and the data part for *dataptr*), **getmsg()** retrieves **maxlen** bytes. It leaves the remainder of the message on the stream head's read queue and returns and a non-zero return value. Details are given below.

Flags

The following summarizes what flags are available, and what they mean.

- By default, **getmsg()** processes the first available message on the stream head's read queue. However, you can choose to retrieve only a high-priority message: just insert **RS_HIPRI** into the integer to which *flagsp* points. In this case, **getmsg()** processes the next message only if it is a high-priority message.
- If the integer to which *flagsp* points equals zero, **getmsg()** retrieves any message available on the stream head's read queue. In this case, if **getmsg()** retrieves a high-priority message, it sets to the integer to which *flagsp* points to **RS_HIPRI**; if the message does not have high priority, it sets that integer to zero.
- If flags **O_NDELAY** and **O_NONBLOCK** are not set as part of the global settings for *fd* (for details, see the Lexicon entry for **open()**), **getmsg()** blocks execution of your program until a message of the type specified by *flagsp* is available on the stream head's read queue. If either **O_NDELAY** or **O_NONBLOCK** has been set and a message of the specified type is not at the top of the queue, **getmsg()** fails and sets **errno** to **EAGAIN**.

If a hangup occurs on the stream from which messages are to be retrieved, **getmsg()** operates normally until the stream head's read queue is empty. Thereafter, it returns zero in the **len** fields of both *ctlptr* and *dataptr*.

Return Values

If all goes well, **getmsg()** returns a non-negative value. Zero indicates that a full message was read successfully.

MORECTL and **MOREDATA** indicate, respectively, that more control information or more data are awaiting retrieval; whereas **MORECTL | MOREDATA** indicates that more of both types information remain in the queue, to be retrieved by subsequent calls to **getmsg()**. However, if a message of higher priority has come into the stream head's read queue, the next call to **getmsg()** retrieves that higher-priority message and the information remaining from the partially retrieved message remains on the queue.

Errors

getmsg() fails if any of the following conditions are true:

- Either of the flags **O_NDELAY** or **O_NONBLOCK** is set but no message is available. **getmsg()** sets **errno** to **EAGAIN**.

- *fd* is not a valid file descriptor. **getmsg()** sets **errno** to **EBADF**.
- The next message in the read queue is not valie for **getmsg()** to read. **getmsg()** sets **errno** to **EBADMSG**.
- *ctlptr*, *dataptr*, or *flagsp* contains an illegal address. **getmsg()** sets **errno** to **EFAULT**.
- A signal was caught as **getmsg()** was executing. **getmsg()** sets **errno** to **EINTR**.
- *flagsp* holds an unrecognized value, or the stream referenced by *fd* is linked under a multiplexor. **getmsg()** sets **errno** to **EINVAL**.
- *fd* does not describe a stream. **getmsg()** sets **errno** to **ENOSTR**.

getmsg() also fails if the stream header receives a STREAMS error message before **getmsg()** tries to read it. **getmsg()** then returns the value in the STREAMS error message.

See Also

libc, putmsg(), STREAMS, stropts.h

getnetbyaddr() — Sockets Function (libsocket)

Get a network entry by address
#include <netdb.h>
struct netent ***getnetbyaddr(***network***,** *type***)**
long *network***; int** *type***;**

getnetbyaddr() fetches a network entry. It opens and searches file **/etc/network**, which describes all entities on your local network, for the entry with *address*. **/etc/networks** must have been opened by function **setnetent()**. *type* is the type of network; at present, **getnetbyaddr()** recognizes only type **AF_INET**.

getnetbyaddr() returns a pointer to an object of type **netend**, which is defined in header file **<netdb.h>**:

```
struct netent {
      char *n_name;     /* official name of net */
      char **n_aliases; /* alias list */
      int n_addrtype;   /* net number type */
      unsigned long n_net;    /* net number */
};
```

The following describes the members:

n_name
The official name of the network.

n_aliases
This points to a zero-terminated list of alternate names for the network.

n_addrtype
The type of the network number returned; currently, only type **AF_INET** is recognized.

n_net The network's number. Network numbers are returned in the machine's byte order.

getnetent() returns a pointer to the **netent** structure it built. It returns NULL if something went wrong or if it cannot find an entry with *address*.

See Also

endnetent(), getnetent(), getnetbyname(), libsocket, netdb.h, setnetent()

getnetbyname() — Sockets Function (libsocket)

Get a network entry by address
#include <netdb.h>
struct netent ***getnetbyname(***name***)**
char **name***;*

getnetbyname() fetches a network entry. It opens and searches file **/etc/networks**, which describes all entities on your local network, for the entry with *name*. **/etc/networks** must have been opened by function **setnetent()**.

getnetbyname() returns a pointer to an object of type **netend**, which is defined in header file **<netdb.h>**:

```
struct netent {
      char *n_name;     /* official name of net */
      char **n_aliases; /* alias list */
      int n_addrtype;   /* net number type */
      unsigned long n_net;    /* net number */
};
```

The following describes the members:

n_name
The official name of the network.

n_aliases
This points to a zero-terminated list of alternate names for the network.

n_addrtype
The type of the network number returned; currently, only type **AF_INET** is recognized.

n_net The network's number. Network numbers are returned in the machine's byte order.

getnetent() returns a pointer to the **netent** structure it built. It returns NULL if something went wrong or if it cannot find an entry with *address*.

See Also

endnetent(), getnetent(), getnetbyaddr(), libsocket, netdb.h, setnetent()

getnetent() — Sockets Function

Fetch a network entry
#include <netdb.h>
struct netent *getnetent();

getnetent() fetches a network entry. It reads the next line of file **/etc/network**, which describes all entities on your local network; if necessary, it opens this file.

getnetent() returns a pointer to an object of type **netend**, which is defined in header file **<netdb.h>**:

```
struct netent {
      char *n_name;     /* official name of net */
      char **n_aliases; /* alias list */
      int n_addrtype;   /* net number type */
      unsigned long n_net;    /* net number */
};
```

The following describes the members:

n_name
The official name of the network.

n_aliases
This points to a zero-terminated list of alternate names for the network.

n_addrtype
The type of the network number returned; currently, only type **AF_INET** is recognized.

n_net The network's number. Network numbers are returned in the machine's byte order.

getnetent() returns a pointer to the **netent** structure it built. It returns NULL if something went wrong or if it has reached the end of **/etc/networks**. You must call function **endnetent()** to close **/etc/networks**.

See Also

getnetbyaddr(), getnetbyname(), endnetent(), libsocket, netdb.h, setnetent()

getopt() — General Function (libc)

Get option letter from argv
#include <unistd.h>
int getopt(*argc***,** *argv***,** *optstring***)**
int *argc***;**
char ***argv***;**
char **optstring***;**
extern char **optarg***;**
extern int *optind***;**

getopt() returns the next option letter in **argv** that matches a letter in *optstring*. *optstring* is a string of recognized option letters. If a letter is followed by a colon, the option must have an argument, which may or may not be separated from it by white space. *optarg* points to the start of the option argument on return from **getopt()**.

getopt() writes into *optind* the **argv** index of the next argument to be processed. Because *optind* is external, it is normally initialized to one automatically before the first call to **getopt()**.

When all options have been processed (i.e., up to the first non-option argument), **getopt()** returns EOF. The special option "--" may be used to delimit the end of the options: **getopt()** returns EOF and skip "--".

See Also

libc

Diagnostics

getopt() prints an error message and returns a question mark when it encounters an option letter not included in *optstring*.

Notes

It is not obvious how '-' standing alone should be treated. This version treats it as a non-option argument, which is not always right.

Option arguments are allowed to begin with '-'. This is reasonable, but reduces the amount of error checking possible.

getopt() returns the parsed letter option in the external **int optopt**, which is overwritten by each call to **getopt()**. When **getopt()** returns '?', it can be helpful to examine the contents of this variable.

getopts — Command

Parse command-line options
getopts *optstring name [opt]*

The command **getopts** parses a command's options and check their legality. *optstring* must contain the options letters that the command using **getopts** will recognize. If a letter is followed by a colon ':', that option must have an argument that is separated from it by whitespace.

Each time it is invoked, **getopts** places the next option into the shell variable *name* and the index of the next argument to be processed into the shell variable **OPTIND**, which is initialized by default to one. When an option requires an argument, **getopts** copies it into the shell variable **OPTARG**. If **getopts** encounters an error, it initializes variable *name* to **?**.

When it encounters the end of the options, **getopts** exits with non-zero status. The special option "--" can be used to delineate the end of options.

Example

The following example processes a command that takes options **a**, **b**, and **o**; the last option requires an argument:

```
while getopts abo: c
do
      case $c in
            a|b)  FLAGS=$FLAGS$c;;
            o)    OARG=$OPTARG;;
            \?)   echo $USAGE 1>&2
                  exit 2;;
      esac
done
shift OPTIND-1
```

This code will accept any of the following as equivalent:

```
cmd -a -b -o"xxx z yy" file
cmd -a -b -o"xxx z yy" -- file
cmd -ab -o"xxx z yy" file
cmd -ab -o"xxx z yy" -- file
```

Note that no space is required between **-o** and its argument.

See Also

commands, getopt(), ksh

getpass() — General Function (libc)

Get password with prompting
char ***getpass(***prompt**)**
char **prompt**;**

getpass() first prints the *prompt*. Then it disables echoing of input characters on the terminal device (either the file **/dev/tty** or the standard input), reads a password from it, and restores echoing on the terminal. It returns the given password.

Files

/dev/tty

See Also

crypt(), libc, login, passwd, su

Notes

The password is stored in a static location that is overwritten by successive calls. This static buffer is 50 characters long; any password longer than that can cause problems of one sort or another.

getpeername() — Sockets Function (libsocket)

Get name of connected peer
int getpeername(*socket***,** *name***,** *namelen***)**
int *socket***,** ***namelen***; struct sockaddr** ***name***;**

getpeername() returns the name of the "peer socket" that is connected to *socket*.

name points to the space into which **getpeername()** writes the name of the peer. *namelen* points to an integer that gives the amount of space to which **name** points. **getpeername()** re-initializes it to the length, in bytes, of the peer name that it has written at *name*.

If all goes well, **getpeername()** returns zero. If an error occurs, it returns -1 and sets **errno** to an appropriate value. The following lists the errors that can occur, by the value to which **getpeername()** sets **errno**:

EBADF *socket* is not a valid descriptor.

ENOTSOCK
socket describes a file, not a socket.

ENOTCONN
socket is not connected.

ENOBUFS
The system lack resources to perform the operation.

EFAULT
name contains an illegal address.

See Also

accept(), bind(), getsockname(), libsocket, socket()

getpgrp() — System Call (libc)

Get process-group identifier
#include <sys/types.h>
#include <unistd.h>
pid_t getpgrp();

getpgrp() returns the identifier of the calling process's process group. It always succeeds.

See Also

libc, types.h, unistd.h
POSIX Standard, §4.3.1

getpid() — System Call (libc)

Get process identifier
#include <unistd.h>
getpid()

Every process has a unique number, called its *process id*. **fork()** returns the process id of a created child process to the parent process.

getpid() returns the process id of the requesting process. Typically a process uses **getpid()** to pass its process id to another process which wants to send it a signal, or to generate a unique temporary file name.

Example

For an example of using this system call in a C program, see **signal()**.

See Also

fork(), getppid(), kill, libc, mktemp, unistd.h
POSIX Standard, §4.1.1

getppid() — System Call (libc)

Get process identifier of parent process
#include <unistd.h>
getppid()

Every process has a unique number, called its *process id*. **fork()** returns the process id of a created child process to the parent process.

getppid() returns the process id of the requesting process's parent process. In this way, a wayward child process can discover the identity of its parent.

See Also

fork(), getpid(), kill, libc, mktemp, unistd.h
POSIX Standard, §4.1.1

getprotobyname() — Sockets Function (libsocket)

Get protocol entry by protocol name
#include <netdb.h>
struct protoent ***getprotobyname(***name***);**
char **name***;*

getprotobyname() searches file **/etc/protocols**, which holds information about all protocols recognized by your local network, for the protocol named *name*. **/etc/protocols** has to have been opened by a call to **setprotoent()**.

getprotobyname() returns a pointer to an object of type **protoent**, which is defined in header file **netdb.h**:

```
struct protoent {
      char *p_name;     /* official name of protocol */
      char **p_aliases; /* alias list */
      int p_proto;      /* protocol number */
};
```

The following details each member:

p_name
The official name of the protocol.

p_aliases
This points to a zero-terminated list of alternate names for the protocol.

p_proto
The number of the protocol.

getprotobyname() returns NULL if an error occurs, or if it encounters the end of the file.

See Also

endprotoent(), getprotobynumber(), getprotoent(), libsocket, netdb.h, setprotoent()

Notes

This function uses a static data space. If your application needs to save these data, it must copy them before any subsequent calls overwrite them.

At present, only the Internet protocols are understood.

getprotobynumber() — Sockets Function (libsocket)

Get protocol entry by protocol number
#include <netdb.h>
struct protoent ***getprotobynumber(***protocol**);**
int *protocol***;**

getprotobynumber() searches file **/etc/protocols**, which holds information about all protocols recognized by your local network, for the protocol identified by *number*. **/etc/protocols** has to have been opened by a call to **setprotoent()**.

getprotobynumber() returns a pointer to an object of type **protoent**, which is defined in header file **netdb.h**:

```
struct protoent {
      char *p_name;     /* official name of protocol */
      char **p_aliases; /* alias list */
      int p_proto;      /* protocol number */
};
```

The following details each member:

p_name
The official name of the protocol.

p_aliases
This points to a zero-terminated list of alternate names for the protocol.

p_proto
The number of the protocol.

getprotobynumber() returns NULL if an error occurs, or if it encounters the end of the file.

See Also

endprotoent(), getprotobyname(), getprotoent(), libsocket, netdb.h, setprotoent()

Notes

This function uses a static data space. If your application needs to save these data it must copy them before any subsequent calls overwrite them.

At present, only the Internet protocols are understood.

getprotoent() — Sockets Function (libsocket)

Get protocol entry
#include <netdb.h>
struct protoent *getprotoent();

getprotoent() reads the next entry from file **/etc/protocols**, which holds information about all protocols recognized by your local network. If necessary, it opens the file. It returns a pointer to an object of type **protoent**, which is defined in header file **<netdb.h>**:

```
struct protoent {
      char *p_name;     /* official name of protocol */
      char **p_aliases; /* alias list */
      int p_proto;      /* protocol number */
};
```

The following details each member:

p_name
The official name of the protocol.

p_aliases
This points to a zero-terminated list of alternate names for the protocol.

p_proto
The number of the protocol.

To close **/etc/protocols**, call function **endprotoent()**.

getprotoent() returns NULL if an error occurs, or if it encounters the end of the file.

See Also

endprotoent(), getprotobyname(), getprotobynumber(), libsocket, netdb.h, setprotoent()

Notes

This function uses a static data space. If your application needs to save these data, it must copy them before any subsequent calls overwrite them.

At present, only the Internet protocols are understood.

getpw() — General Function (libc)

Search password file
getpw(*uid, line***)**
short *uid***;**
char **line***;*

getpw() searches the password file **/etc/passwd** for the first entry with numerical user id *uid*. If found, *line* receives the corresponding line from the password file.

Files

/etc/passwd

See Also

getpwent(), getuid(), libc, passwd

Diagnostics

getpw() returns a nonzero value on error.

getpwent() — General Function (libc)

Get password file information
#include <pwd.h>
struct passwd *getpwent()

The COHERENT system has five routines that search the file **/etc/passwd**, which contains information about every user of the system. The returned structure **passwd** is defined in the header file **pwd.h**. For a description of this structure, see **pwd.h**.

getpwent() returns the next entry from **/etc/passwd**.

Example

The following example demonstrates **getpwent()**, **getpwnam()**, **getpwuid()**, **setpwent()**, and **endpwent()**.

```
#include    <pwd.h>
#include    <stdio.h>
#include    <unistd.h>

main()
{
        int euid,                       /* Effective user id */
            ruid;                       /* Real user id */
        struct passwd *pstp;
        int i;

        /* Print out all users and home directories */
        i = 0;
        setpwent();                     /* Rewind file /etc/passwd */
        while ((pstp = getpwent()) != NULL)
             printf("%d: user name is %s, home directory is %s.\n",
                     ++i, pstp->pw_name, pstp->pw_dir);

        /* Find real user name.
         * NOTE: functions getpwuid and getpwnam rewind /etc/passwd
         * by calling setpwent().
         */
        ruid = getuid();
        if ((pstp = getpwuid(ruid)) == NULL) {
                /* If this message appears, something's wrong */
                fprintf(stderr, "Cannot find user with id number %d\n", pstp);
                exit (EXIT_FAILURE);
        } else
                printf("User's real name is %s\n", pstp->pw_name);

        /* Find the user id for superuser root */
        ((pstp = getpwnam("root")) == NULL) ?
                fprintf(stderr, "Do you have user root on your system?\n") :
                printf("root id is  %d\n", pstp->pw_uid);

      /* Check if the effective process id is the superuser id.
       *
       * NOTE: if you wish to see how to enable the root
       * privileges, you can run this command:
       * cc pwfun.c
       * su root chown root pwfun
       * su root chmod 4511 pwfun
       */

        euid = geteuid();               /* Get effective user id. */
        printf("Process ");
        (euid == pstp->pw_uid) ? printf("has ") : printf("doesn't have ");
        printf("the root privileges\n");
        exit(EXIT_SUCCESS);
}
```

Files

/etc/passwd
pwd.h

See Also

endpwent(), getpwnam(), getpwuid(), libc, pwd.h, setpwent()

Diagnostics

getpwent() returns NULL for any error or on end of file.

Notes

All structures and information returned are in static areas internal to **getpwent()**. Therefore, information from a previous call is overwritten by each subsequent call.

If your system has implemented shadow passwords, you must use the shadow-password routine **getspent()** to retrieve records that contain passwords. For details, see this function's entry in the Lexicon.

getpwnam() — General Function (libc)

Get password file information, by name
#include <pwd.h>
struct passwd *getpwnam(*uname***)**
char ***uname*;**

The COHERENT system has five routines that search the file **/etc/passwd**, which contains information about every user of the system. The returned structure **passwd** is defined in the header file **pwd.h**. For a description of this structure, see **pwd.h**.

getpwnam() attempts to find the first entry with a name of *uname*.

Example

For an example of this function, see the entry for **getpwent()**.

Files

/etc/passwd
pwd.h

See Also

libc
POSIX Standard, §9.2.2

Diagnostics

getpwnam() returns NULL for any error or on end of file.

Notes

All structures and information returned are in static areas internal to **getpwnam()**. Therefore, information from a previous call is overwritten by each subsequent call.

If your system has implemented shadow passwords, you must use the shadow-password routine **getspnam()** to retrieve records that contain passwords. For details, see this function's entry in the Lexicon.

getpwuid() — General Function (libc)

Get password file information, by id
#include <pwd.h>
struct passwd *getpwuid(*uid***)**
int *uid***;**

The COHERENT system has five routines that search the file **/etc/passwd**, which contains information about every user of the system. The returned structure **passwd** is defined in the header file **pwd.h**. For more information on this structure, see **pwd.h**.

getpwuid() attempts to find the first entry with a numerical user id of *uid*.

Example

For an example of this function, see the entry for **getpwent()**.

Files

/etc/passwd
pwd.h

See Also

libc
POSIX Standard, §9.2.2

Diagnostics

getpwuid() returns NULL for any error or on end of file.

Notes

All structures and information returned are in static areas internal to **getpwuid()**. Therefore, information from a previous call is overwritten by each subsequent call.

gets() — STDIO Function (libc)

Read string from standard input
#include <stdio.h>
char ***gets(***buffer**)**
char **buffer***;*

gets() reads characters from the standard input into a buffer pointed at by *buffer*. It stops reading as soon as it detects a newline character or EOF. **gets()** discards the newline or EOF, appends NUL onto the string it has built, and returns another copy of *buffer*.

Example

The following example uses **gets()** to get a string from the console; the string is echoed twice to demonstrate what **gets()** returns.

```
#include <stdio.h>

main()
{
      char buffer[80];

      printf("Type something: ");

      fflush(stdout);
      printf("%s\n%s\n", gets(buffer), buffer);
}
```

See Also

buffer, fgets(), getc(), libc
ANSI Standard, §7.9.7.7
POSIX Standard, §8.1

Diagnostics

gets() returns NULL if an error occurs or if EOF is seen before any characters are read.

Notes

gets() stops reading the input string as soon as it detects a newline character. If a previous input routine left a newline character in the standard input buffer, **gets()** will read it and immediately stop accepting characters; to the user, it will appear as if **gets()** is not working at all.

For example, if **getchar()** is followed by **gets()**, the first character **gets()** will receive is the newline character left behind by **getchar()**. A simple statement will remedy this:

```
      while (getchar() != '\n')
            ;
```

This throws away the newline character left behind by **getchar()**; **gets()** will now work correctly.

getservbyname() — Sockets Function (libsocket)

Get a service entry by name
#include <netdb.h>
struct servent ***getservbyname(***name**,** *protocol);*
char ***name***,** ****protocol***;*

Function **getservbyname()** searches file **/etc/services**, which describes the services offered by TCP/IP on your local network, for the services offered by *name*. If *protocol* is not NULL, the search must also match the protocol it names. **/etc/services** must first have been opened by a call to **setservent()**.

getservbyname() returns a pointer to a structure of type **servent**, which is defined in header file **<netdb.h>**:

```
struct servent {
      char *s_name;     /* official name of service */
      char **s_aliases; /* alias list */
      int s_port; /* port service resides at */
      char *s_proto;    /* protocol to use */
};
```

The following details each member:

s_name
The official name of the service.

s_aliases
This points to a zero-terminated list of alternate names for the service.

s_port The port number at which the service resides. Port numbers are returned in network byte order.

s_proto
The name of the protocol to use when contacting the service.

To close **/etc/services**, call function **endservent()**.

getservbyname() returns NULL if an error occurs, or if it encounters the end of the file.

See Also

endservent(), getservent(), getservbyport(), libsocket, netdb.h, setservent()

Notes

This function uses a static data space. If your application needs to save these data, it must copy them before any subsequent calls overwrite them.

getservbyport() — Sockets Function (libsocket)

Get a service entry by port number
#include <netdb.h>
struct servent ***getservbyport(***port**,** *protocol);*
int *port***; char** **protocol***;*

Function **getservbyport()** searches file **/etc/services**, which describes the services offered by TCP/IP on your local network, for the services offered by *port*. If *protocol* is not NULL, the search must also match the protocol it names. **/etc/services** must first have been opened by a call to **setservent()**.

getservbyport() returns a pointer to a structure of type **servent**, which is defined in header file **<netdb.h>**:

```
struct servent {
      char *s_name;     /* official name of service */
      char **s_aliases; /* alias list */
      int s_port; /* port service resides at */
      char *s_proto;    /* protocol to use */
};
```

The following details each member:

s_name
The official name of the service.

s_aliases
This points to a zero-terminated list of alternate names for the service.

s_port The port number at which the service resides. Port numbers are returned in network byte order.

s_proto
The name of the protocol to use when contacting the service.

To close **/etc/services**, call function **endservent()**.

getservbyport() returns NULL if an error occurs, or if it encounters the end of the file.

See Also

endservent(), getservbyname(), getservent(), libsocket, netdb.h, setservent(),

Notes

This function uses a static data space. If your application needs to save these data, it must copy them before any subsequent calls overwrite them.

getservent() — Sockets Function (libsocket)

Get a service entry
#include <netdb.h>
struct servent *getservent();

Function **getservent()** reads the next entry from file **/etc/services**, which describes the services offered by TCP/IP on your local network. If necessary, it opens the file. It returns a pointer to a structure of type **servent**, which is defined in header file **<netdb.h>**:

```
struct servent {
      char *s_name;     /* official name of service */
      char **s_aliases; /* alias list */
      int s_port; /* port service resides at */
      char *s_proto;    /* protocol to use */
};
```

The following details each member:

s_name
The official name of the service.

s_aliases
This points to a zero-terminated list of alternate names for the service.

s_port The port number at which the service resides. Port numbers are returned in network byte order.

s_proto
The name of the protocol to use when contacting the service.

To close **/etc/services**, call function **endservent()**.

getservent() returns NULL if an error occurs, or if it encounters the end of the file.

See Also

endservent(), getservbyname(), getservbyport(), libsocket, netdb.h, setservent()

Notes

This function uses a static data space. If your application needs to save these data, it must copy them before any subsequent calls overwrite them.

getsockname() — Sockets Function (libsocket)

Get the name of a socket
int getsockname(*socket***,** *name***,** *namelen***)**
int *socket*, ***namelen*; struct sockaddr ***name*;**

Function **getsockname()** returns the current name that is bound to *socket*.

socket is a file descriptor that identifies the socket in question. *name* points to a space into which **getsockname()** can write the socket name. *namelen* points to an integer that holds the number of bytes to which *name* points. **getsockname()** re-initializes this integer to the number of bytes in the name that it writes at address *name*.

If all goes well, **getsockname()** returns zero. If a problem occurs, it returns -1 and sets **errno** to an appropriate value. The following lists the errors that can occur, by the value to which **getsockname()** sets **errno**:

EBADF *socket* is not a valid file descriptor.

ENOTSOCK
socket identifies a file, not a socket.

ENOBUFS
The system lacks sufficient resources to perform the operation.

EFAULT
name contains an illegal address.

See Also

bind(), libsocket, socket()

getsockopt() — Sockets Function (libsocket)

Read a socket option
#include <sys/types.h>
#include <sys/socket.h>
int getsockopt(*socket***,** *level***,** *option***,** *buffer***,** *length***)**
int *socket***,** *level***,** *option***;**
char **buffer***;*
int **length***;*

Function **getsockopt()** reads the options that are set on a socket.

socket gives the identifier of the socket, as returned by the function **socket()**.

level gives the level at which the options are set. To retrieve options set on the socket level, set *level* to **SOL_SOCKET** whereas to retrieve options set the TCP level, set *level* to the number of the TCP protocol.

option gives the number of the option whose setting interests you. For a list of options that are recognized at the socket level, see header file **<sys/socket.h>**. Options at other levels are set by their respective protocols.

buffer gives the address of the buffer into which the retrieve information will be written. *length* gives the address of an integer that gives the length of *buffer*, in bytes. If **getsockopt()** succeeds in retrieving the value of the requested option, it writes the option into *buffer* and re-initializes the **int** to which *length* points to give the length of the material it wrote into *buffer*.

If all goes well, **getsockopt()** returns zero. If something goes wrong, it returns -1 and sets **errno** to one of the following values:

EBADF *socket* does not identify a valid socket.

ENOMEM
The available user memory was insufficient to complete the operation.

ENOPROTOOPT
option gives an unknown option.

ENOTSOCK
socket identifies a file, not a socket.

See Also

libsocket, setsockopt()

getspent() — General Function (libc)

Get a shadow-password record
#include <shadow.h>
struct spwd *getspent()

The COHERENT system has four routines that search the file **/etc/shadow**, which contains the password of every user of the system. **getspent()** returns a record from this file. If a program has already read entries from **/etc/shadow**, **getspent()** returns the next entry; otherwise, it returns the first entry.

If an error occurs, **getspent()** returns NULL. Otherwise, it returns the address of an object with the structure **spwd** which is defined in header file **<shadow.h>**. For a description of this structure, see the Lexicon entry for **shadow.h**.

See Also

endspent(), libc, setspent(), shadow, shadow.h

Notes

All structures and information returned are in static areas internal to **getspent()**. Therefore, information from a previous call is overwritten by each subsequent call.

getspnam() — General Function (libc)

Get a shadow-password record, by user name
#include <shadow.h>
struct spwd ***getspnam(***uname**)**
char ***uname*;**

The COHERENT system has four routines that search the shadow-password file **/etc/shadow**, which contains the password of every user of your system. **getspnam()** returns the first entry for the user with a given login identifier. *uname* points to the login identifier of the user whose password you wish to retrieve.

If an error occurs, **getspnam()** returns NULL. Otherwise, it returns the address of an object with the structure **spwd**, which is defined in the header file **<shadow.h>**. For a description of this structure, see the Lexicon entry for **shadow.h**.

Files

/etc/shadow
shadow.h

See Also

getspent(), libc, shadow, shadow.h
POSIX Standard, §9.2.2

Notes

All structures and information returned are in static areas internal to **getspnam()**. Therefore, information from a previous call is overwritten by each subsequent call.

gettimeofday() — Sockets Function (libsocket)

Berkeley time function
#include <sys/time.h>
#include <time.h>
void gettimeofday (*timeval***,** *zone***)**
struct timeval ***timeval*;**
char ***zone*;**

Function **gettimeofday()** writes the current system time (i.e., the number of seconds since January 1, 1970 GMT) into *timeval***->tv_sec**. It also initializes field *timeval***->tv_usec** to zero.

gettimeofday() ignores argument *zone*. It returns nothing.

See Also

libsocket, time [overview]

getty — System Administration

Terminal initialization
/etc/getty *type*

The initialization process **init** invokes **getty** for each device indicated in the file **/etc/ttys**. **getty** tries to read a user name from the terminal which is the standard input, adapting its mode settings accordingly. Then **getty** invokes **login** with the name read. This process may set delays, mapping of upper to lower case, speed, and whether the terminal normally uses carriage return or linefeed to terminate input.

If the terminal baud rate is wrong, the login message printed by **getty** will appear garbled. If the specified *type* indicates variable speeds, as described below, hitting BREAK will try the next speed.

init passes the third character in a line of the file **/etc/ttys** as the *type* argument to **getty**. *type* conveys information about the terminal port. An upper-case letter in the range **A** to **S** specifies a hard-wired baud rate, as indicated in the header file **<sgtty.h>**. Other characters specify a range of speeds suitable to a dial-in modem. The following variable-speed settings are recognized:

0 Cycles through speeds 300, 1200, 150, and 110 baud, in that order. This is a good default setting for dial-in ports.

- Teletype model 33, fixed at 110 baud.

1 Teletype model 37, fixed at 150 baud.

2 9600 baud with delays (e.g., Tektronix 4104).

3 Cycles between 2400, 1200, and 300 baud. This is used with 2400-bps modems.

4 DECwriter (LA36) with delays.

5 Like **3**, but starts at 300 baud.

getty recognizes the following fixed-speed settings, for hard-wired terminals:

A 50 baud
B 75 baud
C 110 baud
D 134 baud
E 150 baud
F 200 baud
G 300 baud
H 600 baud
I 1200 baud
J 1800 baud
K 2000 baud
L 2400 baud
M 3600 baud
N 4800 baud
O 7200 baud
P 9600 baud
Q 19200 baud
R EXT
S EXT

Files

/etc/tty
<sgtty.h>

See Also

Administering COHERENT, init, ioctl(), login, sgtty.h, stty, ttys

getuid() — System Call (libc)

Get real user identifier
#include <unistd.h>
int getuid()

Every process has two different versions of its *user id*, called the *real* user id and the *effective* user id. The user ids determine eligibility to access files or employ system privileges. Normally, these two ids are identical. However, for a *set user id* load module (see **exec()**), the real user id is that of the user, whereas the effective user id is that of the load module owner. This distinction allows system programs to use files which are protected from the user who invokes the program.

getuid() returns the real user id.

Example

For an example of this call, see the entry for **getpwent()**.

See Also

access(), exec, getegid(), geteuid(), getgid(), libc, login, setuid(), unistd.h
POSIX Standard, §4.2.1

getutent() — General Function (libc)

Read an entry from a login logging file
#include <utmp.h>
struct utmp *getutent()

getutent() reads the next entry from the file that holds login information. If the file is not open, **getutent()** opens it. By default, **getutent()** reads file **/etc/utmp**. To change this, call **utmpname()**.

getutent() returns the address of the record it has read from the login file. This object has type **utmp**, which is defined in header file **<utmp.h>**; for a detailed description of this structure, see the Lexicon entry for **utmp.h**. **getutent()** returns NULL if it cannot open the login file, or when it attempts to read past the end of the file.

See Also

libc, utmp.h

Notes

getutent() writes its **utmp** record into a static portion of memory, which it overwrites the next time it is called. Therefore, if you wish to save **utmp** record, you must copy it into a portion of memory that you define before you again call **getutent()**.

getutid() — General Function (libc)

Find a record in login logging file by login identifier
#include <utmp.h>
struct utmp *getutid(*id***)**
struct utmp *id;

Function **getutid()** searches a login file for a record with a given type, or for a user with a given login identifier.

id is the address of an object type **utmp**, which is a structure whose fields describe a login event. (For a detailed description of this structure, see the Lexicon entry for **utmp.h**). Before you call **getutid()**, initialize *id*'s fields as follows:

- Set field *id***.ut_type** to the type of record you wish to retrieve. The type can be one of the following:

EMPTY	An empty entry
RUN_LVL	Run level
BOOT_TIME	Boot time
OLD_TIME	
NEW_TIME	
INIT_PROCESS	Process spawned by **init**
LOGIN_PROCESS	A **getty** waiting for a login
USER_PROCESS	A user process
DEAD_PROCESS	
ACCOUNTING	

- If you initialize field *id***.ut_type** to **INIT_PROCESS**, **LOGIN_PROCESS**, **USER_PROCESS**, or **DEAD_PROCESS**, initialize field *id***.ut_id** to the identifier of the user whose login event you are seeking. Note that this must be the identifier as set by **/etc/init**, *not* the login identifier that the user types to log into your system.

If you initialize field **ut_type** to **INIT_PROCESS**, **LOGIN_PROCESS**, **USER_PROCESS**, or **DEAD_PROCESS**, **getutid()** seeks the first record that matches both the type and the identifier that you set in **id**. If you initialize field **ut_type** to any other type, it seeks the first record that matches the type you requested.

If it finds a record that matches your request, **getutid()** copies it into a static portion of memory and returns the address of that memory. It returns NULL if it fails to find a record of the type you requested, or if it cannot open the login file.

By default, **getutid()** reads records from **/etc/utmp**. If you wish to read records from another file, call **utmpname()** before you call **getutid()**.

See Also

libc, utmp.h

getutline() — General Function (libc)

Find a record in login logging file by device
#include <utmp.h>
struct utmp ***getutline(***line***)**
struct utmp **line***;*

Function **getutline()** seeks a record for a login that occurred for a given line.

line points to an object of type **utmp**, which is a structure whose fields describe a login event. (For a detailed description of this structure, see the Lexicon entry for **utmp.h**.) Before you call **getutline()**, you must initialize field *line***.ut_line** to the name of the device that interests you.

If it finds a record that matches your request, **getutline()** copies it into a static portion of memory and returns the address of that memory. It returns NULL if it fails to find a record for the device you named, or if it cannot open the login file.

By default, **getutline()** reads records from **/etc/utmp**. If you wish to read records from another file, call **utmpname()** before you call **getutid()**.

See Also

libc, utmp.h

getw() — STDIO Function (libc)

Read word from file stream
#include <stdio.h>
int getw(*fp***) FILE** **fp***;*

getw() reads a word (an **int**) from the file stream *fp*.

getw() differs from **getc()** in that **getw()** gets and returns an **int**, whereas **getc()** returns either a **char** promoted to an **int**, or EOF. To detect EOF while using **getw()**, you must use **feof()**.

See Also

canon, getc(), libc

Notes

getw() returns EOF on errors.

getw() assumes that the bytes of the word it receives are in the natural byte ordering of the machine. This means that such files might not be portable between machines.

GMT — Definition

GMT is an abbreviation of Greenwich Mean Time, the time recorded at the Greenwich Observatory in England, where by international convention the Earth's zero meridian is fixed.

By definition, COHERENT fixes system time in GMT. It calculates local time as an offset of GMT; for example, the time zone for Chicago is six hours (360 minutes) behind Greenwich, so the local time for Chicago is calculated by subtracting 360 minutes from GMT.

See Also

gmtime(), localtime, Programming COHERENT, time, time.h, TIMEZONE

Notes

The ANSI Standard replaces GMT with UTC (*universel temps coordonne* or universal coordinated time) for C programming. The change is mainly one of terminology rather than substance, as some signatories to international conventions object to naming the standard for global time after a suburb of London.

Under international convention, there are two UTC standards: UTC1 is based on solar time and is identical to current GMT; and UTC2, which uses atomic clocks that are corrected by comparison with pulsars. These standards drift apart as the earth's rotation slows; thus, "leap seconds" are inserted periodically into UTC1 to bridge the difference.

gmtime() — Time Function (libc)

Convert system time to calendar structure
#include <time.h>
#include <sys/types.h>
tm *gmtime(*timep***)**
time_t ***timep*;**

gmtime() converts the internal time from seconds since midnight January 1, 1970 GMT, into fields that give integer years since 1900, the month, day of the month, the hour, the minute, the second, the day of the week, and yearday. It returns a pointer to the structure **tm**, which defines these fields, and which is itself defined in the header file **time.h**. Unlike its cousin, **localtime()**, **gmtime()** returns Greenwich Mean Time (GMT).

Example

For an example of how to use this function, see **asctime()**.

See Also

GMT, libc, localtime(), time [overview], TIMEZONE
ANSI Standard, §7.12.3.3
POSIX Standard, §8.1

Notes

gmtime() returns a pointer to a statically allocated data area that is overwritten by successive calls.

gnucpio — Command

Archiving/backup utility
Copy-in mode: **cpio {-o|--create} [-0acvABLV] [-C** *bytes***] [-H** *format***] [-M** *message***] [-O [[***user***@]***host***:]***archive***] [-F [[***user***@]***host***:]***archive***]** [--file=[[*user*@]*host*:]*archive*] **[--format=***format***] [--message=***message***] [--null] [--reset-access-time] [--verbose] [--dot] [--append] [--block-size=blocks] [--dereference] [--io-size=***bytes***] [--version] <** *name-list* **[>** *archive***]**

Copy-out mode: **cpio {-i|--extract} [-bcdfmnrtsuvBSV] [-C** *bytes***] [-E** *file***] [-H** *format***] [-M** *message***] [-R [***user***][:.][***group***]] [-I [[***user***@]***host***:]***archive***] [-F [[***user***@]***host***:]***archive***] [--file=[[***user***@]***host***:]***archive***] [--make-directories] [--nonmatching] [--preserve-modification-time] [--numeric-uid-gid] [--rename] [--list] [--swap-bytes] [--swap] [--dot] [--unconditional] [--verbose] [--block-size=***blocks***] [--swap-halfwords] [--io-size=***bytes***] [--pattern-file=***file***] [--format=***format***] [--owner=[***user***][:.][***group***]] [--no-preserve-owner] [--message=***message***] [--version] [***pattern...***] [<** *archive***]**

Copy-through mode: **cpio {-p|--pass-through} [-0adlmuvLV] [-R [***user***][:.][***group***]] [--null] [--reset-access-time] [--make-directories] [--link] [--preserve-modification-time] [--unconditional] [--verbose] [--dot] [--dereference] [--owner=[***user***][:.][***group***]] [--no-preserve-owner] [--version]** *destination-directory* **<** *name-list*

gnucpio is the GNU version of the archive utility **cpio**. It copies files into or out of a **cpio** or **tar** archive, which is a file that contains other files plus information about them, such as their pathname, owner, timestamps, and access permissions. The archive can be another file on the disk, a magnetic tape, or a pipe.

gnucpio has three operating modes.

Copy-out Mode
: **gnucpio** copies files into an archive. It reads a list of file names, one per line, from the standard input, and writes the archive onto the standard output.

Copy-in Mode
: **gnucpio** copies files from an archive or lists the archive's contents. It reads the archive from the standard input. Any non-option command-line arguments are shell wild-card patterns; only files in the archive whose names match one or more of those patterns are copied from the archive. Unlike in the shell, an initial '.' in a file name does match a wildcard at the start of a pattern, and a '/' in a file name can match wildcards. If the command line contains no pattern, **gnucpio** extracts all files.

Copy-pass Mode
: **gnucpio** copies files from one directory tree to another. This combines the copy-out and copy-in steps without actually using an archive. It reads the list of files to copy from the standard input; the directory into which it copies them is given as a non-option argument.

gnucpio supports the following archive formats: binary, old ASCII, new ASCII, **crc**, old **tar**, and POSIX.1 **tar**. The binary format is obsolete because it encodes information about the files in a way that is not portable between different machine architectures. The old ASCII format is portable between different machine architectures, but should not be used on file systems with more than 65536 i-nodes. The new ASCII format is portable between different machine architectures and can be used on any size file system, but is not supported by all versions of **cpio**; currently, it is only supported by GNU and UNIX System V R4. The **crc** format resembles the new ASCII format, but also contains a checksum for each file that **gnucpio** calculates when creating an archive and verifies when the file is extracted from the archive.

tar format is provided for compatability with the command **tar**. It can not be used to archive a file whose name exceeds 100 characters, and cannot be used to archive block or character devices. The POSIX.1 **tar** format can not be used to archive a file whose name exceeds 255 characters (less unless it has a '/' in just the right place).

By default, **gnucpio** creates binary archives, for compatibility with older **cpio** programs. When extracting from archives, **gnucpio** automatically recognizes the kind of archive it is reading, and can read archives created on machines with a different byte-order.

Options

gnucpio recognizes the following command-line options. Not every option applies to every mode. You can prefix the long-named options with an '+' as well as with an '--', for compatibility with previous releases. Eventually, support for '+' will be removed, because it is incompatible with the POSIX Standard.

-0
--null In copy-out and copy-pass modes, read a list of file names terminated by a null character instead of a newline. This permits **gnucpio** to archive files whose names contain newlines.

-a
--reset-access-time
Reset the access times of files after reading them, so that it does not look like they have just been read.

-A
--append
Append to an existing archive. Only works in copy-out mode. The archive must be a disk file specified with the options **-O** or **-F**.

-b
--swap In copy-in mode, swap both halfwords of words and bytes of halfwords in the data. Equivalent to the option **-sS**. Use this option to convert 32-bit integers between big-endian and little-endian machines.

-B Set the I/O block size to 5,120 bytes. Initially, the block size is 512 bytes.

--block-size=*blocks*
Set the block size to *blocks*×512 bytes.

-c Use the old portable (ASCII) archive format.

-C *size*

--io-size=*size*
Set the I/O block size to *size* bytes.

-d
--make-directories
Create leading directories where needed.

-E file
--pattern-file=*file*
In copy-in mode, read from *file* additional patterns that specify file names to extract or list. **gnucpio** treats the lines of *file* as if they had been non-option arguments to **gnucpio**.

-f
--nonmatching
Copy only the files that do *not* match any of the given patterns.

-F

LEXICON

--file=*archive*
Read to or write from *archive* instead of the standard input or output. When you use this option, you do not have to specify the output device for each volume of a multi-volume backup.

--force-local
With options **-F**, **-I**, or **-O**, take the archive file name to be a local file even if it contains a colon (which ordinarily names a remote host).

-H *format*
--format=*format*
Use archive format *format*. The valid formats are listed below; **gnucpio** also recognizes these names if given in capital letters. The default in copy-in mode is to detect automatically the archive format, and in copy-out mode is **bin**.

bin The obsolete binary format.

odc The old (POSIX.1) portable format.

newc The new (SVR4) portable format, which supports file systems that have more than 65536 i-nodes.

crc The new (SVR4) portable format with a checksum added.

tar The old **tar** format.

ustar The POSIX.1 **tar** format. Also recognizes GNU **tar** archives, which are similar but not identical.

-i
--extract
Run in copy-in mode.

-I *archive*
Archive file name to use instead of standard input.

-k This option exists only for compatibility with other versions of **cpio**. It is ignored.

-l
--link Whenever possible, link files instead of copying them.

-L
--dereference
Dereference symbolic links — that is, copy the files that they point to instead of copying the links.

-m
--preserve-modification-time
Retain previous file-modification times when creating files.

-M *message*
--message=*message*
Print *message* when **gnucpio** reaches the end of a volume of the back-up medium (such as a tape or a floppy disk), to prompt the user to insert a new volume. If *message* contains the string **%d**, **gnucpio** replaces that string with the number of the current volume (starting at one).

-n
--numeric-uid-gid
In the verbose table of contents listing, show the numeric UID and GID instead of translating them into names.

--no-preserve-owner
In copy-in and copy-pass modes, do not change the ownership of the files: leave them owned by the user who extracts them. This is the default for non-root users, so that users on System-V UNIX do not inadvertantly give away files.

-o
--create
Run in copy-out mode.

-O *archive*
Write output into *archive* instead of to the standard output.

-p
--pass-through
Run in copy-pass mode.

-r
--rename
Interactively rename files.

-R [*user***][:.][***group***]**
--owner [*user***][:.][***group***]**
In copy-out and copy-pass modes, set the ownership of all files created to *user* and *group*. Either the user or the group, or both, must be present. If the group is omitted but the ':' or '.' separator is given, **gnucpio** uses the user's login group. Only the super-user can change files' ownership.

-s
--swap-bytes
In copy-in mode, swap the bytes of each halfword (pair of bytes) in the files.

-S
--swap-halfwords
In copy-in mode, swap the halfwords of each word (four bytes) in the files.

-t
--list Print a table of contents of the input.

-u
--unconditional
Replace all files, without asking whether to replace existing newer files with older files.

-v
--verbose
List the files processed. When used with the option **-t**, give a listing that resembles the output of the command **ls -l**. In a verbose table of contents of a **ustar** archive, user and group names in the archive that do not exist on the local system are replaced by the names that correspond locally to the numeric UID and GID stored in the archive.

-V --dot
Print a '.' for each file processed.

--version
Print the number of the version of **gnucpio** that you are now running, and exit.

Examples

The following command copies all files and directories listed by the command **find** and copies them into the archive **newfile.cpio**:

```
find . -print | cpio -oc > ../newfile.cpio
```

The following command reads the **cpio** archive **newfile.cpio** and extracts all files whose names match the patterns **memo/al** or **memo/b***:

```
cpio -icdv "memo/al" "memo/b*" <../newfile.cpio
```

Note that the **-d** option forces **cpio** to create the sub-directory **memo** and write the files into it. Otherwise, the files would have been written into the current directory. Option **-v** causes **cpio** to display each file name as it is extracted from the archive.

The following commands perform a multi-volume backup of all files on mounted filesystem **/v** to the character-special (i.e., "raw") floppy device **/dev/rfha0**:

```
su root
cd /v
find . -print | cpio -ocv >/dev/rfha0
```

If the **cpio** archive exceeds one floppy disk, you will be prompted to insert another.

See Also

commands, cpio, gtar

LEXICON

Notes

COHERENT does not yet support networking. The above descriptions of host addressing do not yet apply.

gnucpio is released under the conditions of the Free Software Foundation's "copyleft". Full source code is available through the Mark Williams bulletin board.

goto — C Keyword

Unconditionally jump within a function

A **goto** command jumps to the area of the program introduced by a label. A program can **goto** only within a function; to jump across function boundaries, you must use the functions **setjmp()** and **longjmp()**.

In the context of C programming, the most common use for **goto** is to exit from a control block or go to the top of a control block. It is used most often to write "ripcord" routines, i.e., routines that are executed when a major error occurs too deeply within a function for the program to disentangle itself correctly. Note that in most instances, **goto** is a bad solution to a problem that can be better solved by structured programming.

Example

The following example demonstrates how to use **goto**.

```
#include <stdio.h>

main()
{
      char line[80];

getline:
      printf("Enter line: ");
      fflush(stdout);
      gets(line);

/* a series of tests often is best done with goto's */
      if (*line == 'x') {
            printf("Bad line\n");
            goto getline;
      } else if (*line == 'y') {
            printf("Try again\n");
            goto getline;
      } else if (*line == 'q')
            goto goodbye;
      else
            goto getline;


goodbye:
      printf("Goodbye.\n");
      exit(0);
}
```

See Also

C keywords
ANSI Standard, §7.6.6.1

Notes

The C Programming Language describes **goto** as "infinitely-abusable": *caveat utilitor*.

grep — Command

Pattern search

grep searches each *file* for occurrences of the *pattern* (sometimes called a regular expression). If no *file* is specified, **grep** searches the standard input. The *pattern* is given in the same manner as to **ed**. Normally, **grep** prints each line matching the *pattern.*

grep recognizes the following command-line options:

-b With each output line, print the block number in which the line started (used to search file systems).

-c Print the count of matching lines rather than the lines.

-e The next argument is *pattern* (useful if the pattern starts with '-').

-f The next argument is a file that contain a list of patterns separated by newlines; there is no *pattern* argument.

-h When more than one *file* is specified, output lines are normally accompanied by the file name; **-h** suppresses this.

-i Ignore case when matching letters in *pattern*. For example, an 'a' in *pattern* matches either 'a' or 'A' in *file*; likewise, an 'A' in *pattern* matches either 'a' or 'A'.

-l Print the name of each file containing matching lines rather than the lines.

-n The line number in the file accompanies each line printed.

-s Suppress all output, just return status.

-v Print a line if the pattern is *not* found in the line.

-x Print the line only if it is exactly the same as the pattern; treat wildcards in the pattern as plain text.

-y Lower-case letters in *pattern* match only upper-case letters within the input lines.

Limits

The COHERENT implementation of **grep** sets the following limits on input and output:

Characters per input record	512
Characters per output record	512
Characters per field	512

See Also

cgrep, commands, ed, egrep, zgrep

Diagnostics

grep returns an exit status of zero for success, one for no matches, two for error.

Notes

cgrep is a version of **grep** that is optimized for handling C-style expressions.

egrep is an extended and faster version of **grep**.

group — System Administration

Define groups of users

The group file **/etc/group** describes the user groups that have been defined on your COHERENT system. This allows users to control the access that members of their group have to certain files. **/etc/group** contains the information to map any ASCII group name to the corresponding numerical group identifier, and vice versa. It also contains, in ASCII, the names of the members of each group. This information is used by, among others, the command **newgrp**.

Each group has an entry in the file **/etc/group** one line per entry. Each line consists of four colon-separated ASCII fields, as follows:

group_name **:** *password* **:** *group_number* **:** *member***[,***member***...]**

Passwords are encrypted with **crypt**, so the group file is generally readable.

The COHERENT system has five system calls that manipulate **/etc/group**, as follows:

endgrent() Close **/etc/group**.

getgrent() Return the next entry from **/etc/group**.

getgrnam() Return the first entry with a given group name.

getgrgid() Return the first entry with a given group identifier.

setgrent() Rewind **/etc/group**, so that searches can begin again from the beginning of the file.

The calls **getgrent()**, **getgrid()**, and **getgrnam()** each return a pointer to structure **group**, which the header file **grp.h** defines as follows:

```
struct group {
        char    *gr_name;       /* Group name */
        char    *gr_passwd;     /* Group password */
        int     gr_gid;         /* Numeric group id */
        char    **gr_mem;       /* Group members */
};
```

A user can belong to more than one group. His "main" group, however, is the one that is named is in his entry in the file **/etc/passwd**. When a user creates a file, that file by default is "owned" by the user's main group.

For example, consider user **joe**, who has the following entry in **/etc/passwd**:

```
joe:*:10:5:Joe Smith:/usr/joe:/usr/bin/ksh
```

The fourth field, which in this example has the value **5**, gives the number of the user's main group. (For details on what the other fields mean, see the Lexicon entry for **passwd**.) Looking in **/etc/group**, we see the following entry for group 5:

```
user::5:
```

Thus, whenever **joe** creates a file, by default it will be "owned" by group **user**. Any member of group **user** will be granted that file's group-level permissions on that file.

A user can use the command **chmod** to change the group-level permissions on any file he owns. The superuser **root** can use the command **chgrp** to changes the group ownership for any file. For details on how to use these commands, see their entries in the Lexicon.

Files

/etc/group

See Also

Administering COHERENT, **chgrp(), chmod, chown, endgrent(), getgrent(), getgrgid(), getgrnam(), grp.h, newgrp, passwd, setgrend()**

Notes

At present the group password field cannot be set directly (no command similar to **passwd** exists for groups). One alternative is to set the password in the **/etc/passwd** file for a user with the **passwd** command, then transcribe the password into the group file manually.

grp.h — Header File

Declare group structure
#include <grp.h>

The header file **grp.h** declares the structure **group**, which is composed as follows:

```
struct group (
        char *gr_name;          /* group name */
        char *gr_passwd;        /* group password */
        int  gr_gid;            /* numeric group id */
        char **gr_mem;          /* group members */
};
```

This structure holds information about the group to which a given user belongs, as defined in the file **/etc/group**. It is used by the functions **endgrent()**, **getgrent()**, **getgrgid()**, **getgrnam()**, and **setgrent()**.

See Also

header files
POSIX Standard, §9.2.1

gtar — Command

Archiving/backup utility
gtar *options*

gtar is the GNU version of the archiving utility **tar**. It copies files into or out of a **tar** archive, reads the contents of a **tar** archive, and replaces files within an archive. It can also perform additional tasks such as compressing files as they are added to an archive, or uncompressing them as they are read out.

gtar works in either of two modes:

Copy-in Mode
: **gtar** copies files from an archive or lists the archive's contents. By default, it reads the archive from the standard input; you can also use the option **-f** (described below) to name the file or device that holds the archive you want read.

  gtar regards any non-option argument as a shell wild-card pattern; and it copies from the archive only those files whose names match one or more of those patterns. Unlike the shell, an initial '.' in a file name matches a wildcard at the start of a pattern, and a '/' in a file name can match a wildcard. If the command line contains no pattern, **gtar** extracts all files.

Copy-out Mode
: **gtar** copies files into an archive. By default, **gtar** reads a list of file names, one per line, from the standard input. However, if the command line contains non-option arguments, **gtar** regards each as a shell wild-card pattern that names one or more files to copy into the archive. If an argument names a directory, then **gtar** recursively copies all files within that directory into the archive.

  By default, **gtar** writes its newly built archive to the standard output. However, you can use the option **-f** (described below) to name the file or device into **gtar** writes the new archive.

  gtar normally writes into the local directory all files that it reads from an archive. If files were archived using absolute path names, **gtar** by default drops the leading '/' from the path name; to suppress this behavior, use the option **-P**, described below. If a file being extracted resides within a directory that does not exist in the current directory, **gtar** will create that directory. **gtar** will fail, of course, if you do not have write permission in the current directory.

Options

gtar recognizes the following options. Please note that not every option applies to both modes.

Please note, too, that some options have more than one name. Every option has a multi-character name that begins with with two hyphens **--**; some commonly used options also have a one-character name that begins with a single hyphen. This convention may appear clumsy, but it does permit option names to have hyphens embedded within them.

The following command-line options govern the mode in which **gtar** works:

-A
--catenate
--concatenate
: Append files onto an archive.

-c
--create
: Create a new archive.

-d
--diff
--compare
: Find the differences between the files in an archive and the identically named files in the file system. This is very useful in verifying that a new archive was built correctly.

--delete
: Delete files from the archive. Do not for use this option with an archive that is on a magnetic tape.

-r

--append
Replace files within an archive. If a file does not exist within an archive, append it onto the archive.

-t
--list List the contents of an archive.

-u
--update
Append a file onto an archive only if it is younger than the identically named file within the archive.

--use-compress-program
Specify the compression program to use. By default, **gtar** invokes **gzip** to compress files.

-x
--extract
--get Extract files from the archive.

The following options modify other aspects of **gtar**'s behavior:

--atime-preserve
Do not change the access times on files, whether copying into or out of an archive.

-b *N*
--block-size *N*
Use a block size of *N*×512 bytes. By default, **gtar** uses an *N* of 20 — that is, a block size of ten kilobytes.

-B
--read-full-blocks
Tell **gtar** to reblock as it reads. This is required for reading pipes under Berkeley UNIX release 4.2, and does not apply to COHERENT.

--block-compress [compress | gzip]
Block the output of the compression program for tapes. You must name one of the compression options to use: either **compress** or **gzip**.

-C *directory*
--directory *directory*
Change to *directory*.

--checkpoint
Print directory names while reading the archive.

--exclude *file*
Do not include *file* when archiving or de-archiving files. *file* can be a regular expression.

-f *file*
--file *file*
Read the input from, or write the output to, *file*. *file* can name an ordinary file or a device. File name '-' indicates the standard input or standard output (depending upon whether an archive is being read or written). When this option is not used, **gtar** by default reads from the standard input and writes to the standard output.

--force-local
The archive file is local even if its name contains a colon. **gtar** usually interprets a file name that contains a colon as naming a file on a remote system that is connected via a network.

-F *script*
--info-script *script*
--new-volume-script *script*
At the end of each tape (or disk), run *script*. Note that this option implies that you are also using option **-M**.

-G [*file ...***]**
--incremental
Create, list, or extract every *file* that is in an archive written in the format of the old GNU incremental backup. If no *file* is named, all **gtar** extracts all files.

-g
--listed-incremental
Create, list, or extract files that are in an archive written in the format of the new GNU incremental backup. create/list/extract new GNU-format incremental backup

-i
--ignore-zeros
Ignore blocks of zeros in archive.

--ignore-failed-read
gtar normally exits with non-zero status when it encounters an unreadable file. With this option, **gtar** ignores the unreadable file and continues to work.

-k
--keep-old-files
If a file being extracted from an archive has an identically named analogue in the file system, **gtar** normally overwrites the file in the file system with the file withdrawn from the archive. This option tells **gtar** to rename the file that is in the file system, rather than overwrite it.

-K *file*
--starting-file *file*
Keep option: begin work with *file* in the archive.

-l
--one-file-system
Stay in the local file system when creating an archive.

-L *N*
--tape-length *N*
Change tapes after writing *N*×1,024 bytes. **gtar** normally reads or writes until it reaches the end of the medium, then prompts for the name of the next device. This option, of course, normally does not apply to archives being written to or read from disk.

-m
--modification-time
Do not extract file modified time.

-M
--multi-volume
Create, list, or extract a multi-volume archive. You can use this option with multiple **-f** options. **gtar** uses the output devices in sequence, then wraps around to the beginning. This lets you, say, write output to two different tape drives or floppy-disk drives; you can loading blank media into one while **gtar** is writing to the other. Note that if you are using this option to create an archive, be *very* careful to label disks or tapes correctly to note the order in which they were written.

-N *date*
--after-date *date*
--newer *date*
Only store files newer than *date*.

-o
--old-archive
--portability
Write a V7-format archive, rather than an ANSI-format archive.

-O
--to-stdout
Write files to the standard output.

-p
--same-permissions
--preserve-permissions
Preserve the permissions that the file had originally.

-P

--absolute-paths
Do not strip leading '/'s from file names.

--preserve
This option is identical to **-p** plus **-s**.

-R
--record-number
Show record number within archive with each message.

--remove-files
Remove files after adding them to the archive.

-s
--same-order
--preserve-order
Sort the list of names to extract to match their order within the archive.

--same-owner
Create extracted files with the same ownership they had within the archive.

-S
--sparse
Handle sparse files efficiently. For a description of what a *sparse file* is, see the Lexicon entry for **chsize()**.

--show-omitted-dirs
Print the names of directories omitted from the archive.

-T file
--files-from *file*
Read from *file* the names of all files to archive or extract.

--null Modify option **-T** so that it reads null-terminated names. This option disables option **-C**.

--totals Print the number of bytes written with option **-c**.

--use-compress-program *program*
Filter the archive through *program*. Note that *program* must accept option **-d**.

-v
--verbose
Write the names of all files archived or extracted. When you also use the option **-f**, **gtar** writes the names to the standard output; however, when you do not use **-f**, it writes them to the standard error.

-V *name*
--label *name*
Name the archive *name*. When used with the option **--extract**, *name* can be a regular expression.

--version
Print the version of **gtar** that you are using.

--volno-file *file*
Read from *file* the volume number used when prompting the user. Note that **gtar** does not use the contents of *file* when it records volume identifiers on the archive.

-w
--interactive
--confirmation
Ask the user to confirm every action.

-W
--verify Attempt to verify the archive after writing it.

-X *file*
--exclude-from *file*
Do *not* archive or de-archive all of the files named in *file*.

-Z
--compress
--uncompress
Filter files being archived or de-archived through **compress**.

-z
--gzip
--ungzip
Filter files being archived or de-archived through **gzip**.

Examples

The first example archives **piggy**, into archive **piggy.tar**:

```
gtar -cf piggy.tar piggy
```

To simultaneously compress **piggy** with the utility **gzip**, use the command:

```
gtar -czf piggy.gtz piggy
```

Note that the suffix **.gtz** is used by convention to mark archives whose contents are compressed. This is not required, but it is a good idea to use this or some similar suffix to mark compressed archives: if you do not remember to use the **-z** option to de-archive a compress archive, **gtar** will fail. So, to extract file **piggy** from its compressed archive, use the command:

```
gtar -xzf piggy.gtz piggy
```

The **-z** is recommended: it speeds archiving of large files or file systems, and increases their accuracy — because the archives are smaller, there are fewer opportunities for errors to occur.

To write an archive onto a device, use the option **-f** to name that device instead of a file. You must, of course, have write permission on that device. If you are writing onto a floppy disk, the disk must have been formatted with the command **fdformat**, but does not need to have a COHERENT file system on it; in fact, **gtar** will overwrite all file-system information that may reside on a disk. For example, to write file **piggy** onto a high-density, 5.25-inch, formatted floppy disk in drive 0, use the following command:

```
gtar -czf /dev/fha0 piggy
```

To copy **piggy** back from this archive, use the command:

```
gtar -xzf /dev/fha0
```

As noted above, you must remember to use the **-z** option to de-archive files from a compressed archive.

As noted above, if you name a directory on **gtar**'s command line, **gtar** will archive or de-archive that directory and all files that it contains, including its sub-directories and their contents. For example, to archive all of your personal files, use the command:

```
gtar -cvzf backup.gtz $HOME
```

The option **-v** tells **gtar** to name every file that it is copying into its archive. Note, too, that **gtar** is smart enough not to copy an archive into itself, so you can execute the above command while still within your home directory.

The following backs up your personal files onto a high-density, 3.5-inch disk in drive 0:

```
gtar -cvzf /dev/fva0 $HOME
```

NB, if you are backing up a directory that will require more than one floppy disk, you should consider using the utility **cpio** instead: it is somewhat easier to use when you are handling multiple-volume archives.

To copy directory **src** to the SCSI tape device with SCSI identifier 2, use the command:

```
tar cvzf /dev/rStp2 src
```

To archive **src** to a tape and then confirm it, use the command

```
tar cvzf /dev/rStp2 src ; tar dvzf /dev/rStp2 src
```

Note that this can be time consuming, but will confirm the integrity of backups of vital files. To restore **src** from its tape, use the command:

```
tar xvzf /dev/rStp2
```

LEXICON

gtar by default saves files with their original ownerships and permissions; however, when it restores files, it may modify them. To restore files with their original permissions, use the option **-p**. For example, to restore **src** and restore the original ownership and permissions of its files, use the command:

```
tar xvpzf /dev/rStp2
```

See Also

commands, compression, gnucpio, tar
POSIX Standard, §10.1.1

Notes

COHERENT does not yet support networking. The above descriptions of host addressing do not yet apply.

gtar is released under the conditions of the Free Software Foundation's "copyleft". Full source code is available through the Mark Williams bulletin board.

gtty() — System Call (libc)

Device-dependent control
#include <sgtty.h>
int gtty(*fd, sgp***)**
int *fd***; struct sgttyb** **sgp***;*

gtty() gets attributes of a terminal. It is shorthand notation for **ioctl** calls with a *command* argument of **TIOCGETP**.

Example

For examples of this system call, see **pipe()** and **stty()**.

See Also

exec, libc, ioctl(), open(), read(), sgtty.h, stty(), write()

guess — Command

Extraordinarily amusing guessing game
/usr/games/guess

The COHERENT game **guess** plays a guessing game with you. When you first invoke it, it will ask you to think of an object. As you go through the guessing game, it will ask you for questions by which that object can be distinguished from other objects. **guess** gets "smarter" over time (assuming you don't lie to it), so it over time develops a fighting chance of actually guessing something.

See Also

commands

Notes

guess is not for the impatient.

gunzip — Command

GNU utility to uncompress files
gunzip [-cfhLrtvV] [*file ...* **]**

gunzip is the GNU command that uncompresses each *file* named on its command line.

Whenever possible, **gunzip** replaces each *file* whose name ends with **.z** or **.Z** (and which begins with the correct magic number) with an uncompressed file without the original suffix. **gunzip** also recognizes the special extensions **.tgz** and **.taz** as shorthands for **.tar.z** or **.tar.Z**.

gunzip can currently decompress files created by the COHERENT commands **gzip** or **compress**, or by the UNIX commands **zip** or **pack**. It automatically detects the format by which the file is compressed and applies the correct algorithm to uncompress it.

When uncompressing the formats used by **gzip** and **zip**, **gunzip** checks a 32-bit CRC. For files compressed by **pack**, **gunzip** checks the uncompressed length.

The format used by **compress** was not designed to allow consistency checks. However, **gunzip** can sometimes

detect a corrupted **.Z** file. If you get an error when uncompressing a **.Z** file, do not assume that the **.Z** file is correct simply because the COHERENT command **uncompress** does not complain. This generally means that most implementations of **uncompress** do not check their input, and happily generate garbage output.

Command-Line Options

gunzip recognizes the following command-line options:

-c Write output to standard output, and do not change the original *file*. If the command line names more than one *file*, **gzip** writes to the standard output a sequence of independently compressed members. To obtain better compression, concatenate the *file*s before compressing them.

-f force compression or decompression, even if *file* has multiple links or the corresponding file already exists. Without this option, and when not running in the background, **gzip** prompts to verify whether it should overwrite an existing file.

-h Help: display a screenful of information on how to run this program.

-L Display the **gzip** license.

-r Recurse: if a *file* is a directory, compress or uncompress all files within it.

-t Test: check the integrity of a compressed file.

-v Verbose: display the name and percentage reduction for each *file* as it is compressed.

-V Display the version of this command, and the options by which it was compiled.

See Also

commands, compress, compression, gzip, unpack

Diagnostics

gunzip returns zero if all went well. It returns one if an error occurred and it returns two if it had to issue a warning message.

gunzip can issue the following warning messages:

*file***: not in gzip format**
A *file* named on the command line was not compressed.

The compressed file has been damaged. If the data were compressed by the program **compress**, they can be recovered up to the point of damage by using the program **zcat** to concatenate the file into another file.

*file***: compressed with** *XX* **bits, can only handle** *YY* **bits**
file was compressed by a program that could deal with more bits than the decompress code on this machine. Recompress the file with **gzip**, which compresses better and uses less memory.

*file***: already has z suffix -- no change**
file has the suffix **.z** or **.Z**; therefore, **gunzip** assumes that it is compressed already.

file **already exists; do you wish to overwrite (y or n)?**
Respond 'y' if you want the output file to be replaced; 'n' if not.

gunzip: corrupt input
gunzip detected a **SIGSEGV** violation, which usually means that the input file has been corrupted.

Notes

gzip is released under the conditions of the Free Software Foundation's "copyleft". Full source code is available through the Mark Williams bulletin board.

gzip — Command

GNU utility to compress files
gzip [-cdfhLrtvV19] [*file ...* **]**

The command **gzip** is the GNU command that compresses *file* named on its command line. It will only attempt to compress regular files.

Whenever possible, **gzip** replaces each *file* with one that has the suffix **.gz**, while preserving its ownership and times of last access and last modification. If the name of *file* is longer than 12 characters (which prevents **gzip**

from attaching the suffix **.gz**), **gzip** truncates it and keeps its original name within the compressed file.

If its command line names no *file*, **gzip** compresses what it reads from from the standard input, and writes it to the standard output.

To restore a compressed file, filter it thorugh the command **gunzip**.

gzip uses the Lempel-Ziv algorithm to perform compression. Under most circumstances, this algorithm compresses files more tightly than do most other commonly used techniques, such as the LZW algorithm, Huffman coding, or adaptive Huffman coding. The amount of compression obtained depends upon the size of the input and the distribution of common substrings; in general, it reduces text by 60% to 70%.

gzip always compresses its input, even if the compressed file is slightly larger than the original. The worst-case expansion is a few bytes for the **gzip** file header, plus five bytes for every 32-kilobyte block.

Command-Line Options

gzip recognizes the following command-line options:

-c Write output to standard output, and do not change the original *file*. If the command line names more than one *file*, **gzip** writes to the standard output a sequence of independently compressed members. To obtain better compression, concatenate the *file*s before compressing them.

-d Decompress each *file*.

-f force compression or decompression, even if *file* has multiple links or the corresponding file already exists. Without this option, and when not running in the background, **gzip** prompts to verify whether it should overwrite an existing file.

-h Help: display a screenful of information on how to run this program.

-L Display the **gzip** license.

-q Suppress all warning messages.

-r Recurse: if a *file* is a directory, compress or uncompress all files within it.

-t Test: check the integrity of a compressed file.

-v Verbose: display the name and percentage reduction for each *file* as it is compressed.

-V Display the version of this command, and the options by which it was compiled.

1-9 Regulate the speed of compression, on a scale of from **1** to **9**. **1** performs the fastest but most superficial compression, whereas **9** performs the slowest but most thorough compression. **-fast** is a synonym for **-1**, whereas **-best** is a synonym for **-9**. The default compression level is **-5**.

Advanced Usage

You can concatenate multiple compressed files. In this case, **gunzip** extracts all members at once. For example:

```
gzip -c file1 > foo.gz
gzip -c file2 >> foo.gz
gunzip -c foo
```

is equivalent to:

```
cat file1 file2
```

In case of damage to one member of a **.gz** file, other members can still be recovered (if the damaged member is removed). However, you can get better compression by compressing all members at once:

```
cat file1 file2 | gzip > foo.gz
```

compresses better than:

```
gzip -c file1 file2 > foo.gz
```

If you want to recompress concatenated files to get better compression, type:

```
zcat old.gz | gzip > new.gz
```

Environment

gzip reads the environment variable **GZIP** for its default options. It interprets these options first; you can override them by setting other options on the **gzip** command line.

See Also

commands, compress, compression, gunzip, uncompress, unpack, zcmp, zdiff, zforce, zgrep, zmore, znew

Diagnostics

If all went well, **gzip** returns zero upon exiting. If an error occurred, it returns one; if it issued a warning message, it returns two.

gzip can issue the following warning messages:

Usage: gzip [-cdfhLrtvV19] [file ...]
The **gzip** command line contained an option that **gzip** does not recognize.

*file***: already has z suffix -- no change**
file already has the suffix **.gz**; therefore, **gzip** assumes that it already is compressed.

file **not a regular file or directory: ignored**
A *file* is not a regular file or directory. **gzip** does not attempt to compress devices, pipes, or other special files.

file **has** *XX* **other links: unchanged**
file has more than one link. By default, **gzip** does not compress a file that has multiple links.

Notes

gzip is released under the conditions of the Free Software Foundation's "copyleft". Full source code is available through the Mark Williams bulletin board.

hai — Device Driver

Host adapter-independent SCSI driver

hai is a host adapter-independent device driver that supports various SCSI devices. It supports the Adaptec 154x host adapter, and compatibles; and all host adapters built around the Future Domain TMC-950/9C50 chip set. With a supported host adapter, **hai** can support any mix of up to seven SCSI hard disks (either fixed or removable media), CD-ROM drives, and tape drives.

hai has major-device number 13. It can access devices either in block mode or character mode. The minor number specifies the device and partition number for disk-type devices; this allows the use of up to eight SCSI identifiers (SCSI-ID's), with up to four logical unit numbers (LUNs) per SCSI-ID and up to four partitions per LUN. Tape and other special devices decode the minor number to perform special operations such as "rewind on close" or "no rewind on close". The first **open()** call on a SCSI disk device allocates memory for the partition table and reads it into memory.

hai is a modular driver that you can configure to suit your system's suite of SCSI hardware. To build the driver, you must link the main **hai** module with the appropriate module for your system's SCSI host-adapter card, and a module for each type of SCSI device you have (hard disk, CD-ROM, or tape). Each of **hai** modules is described below. Usually, you will configure **hai** when you install COHERENT onto your system, but you can reconfigure **hai** at any time should you wish to add or modify your system's suite of SCSI devices. The script **/etc/conf/hai/mkdev** walks you through this process. Once you have reconfigured **hai**, run the program **/etc/conf/bin/idmkcoh** to build a new kernel; then reboot your system to invoke the newly built kernel and you're back in business.

Extending hai

hai is designed to be extendable to other host adapters and other SCSI devices. It is easy to extend **hai** to work with new hardware. It is possible to extend **hai** either to support a new host adapter, or to support new peripheral device, or both.

To adapt **hait** to a new a host adapter, you must write a handful of routines to initialize and access the host adapter. A host-adapter module must be able to do the following:

- Initialize the host adapter and ready it for future requests.
- Start a SCSI command and call a notification function when that command completes or times out.
- Abort a SCSI command in progress.
- Reset a device on the SCSI bus.
- Reset the SCSI bus.

It is easier to write a module for a peripheral device: you only need to send the appropriate SCSI commands to access the device as required by the COHERENT device-driver interface — i.e., **open()**, **close()**, **read()**, **write()**, and **ioctl**(). To do this, use the routines provided by the host-adapter module, when neccessary, to access the SCSI bus and the device.

The following sections of this article discuss each of **hai**'s constituent modules.

hai154x — Adaptec Host-Adapter Module

hai154x is the **hai** host-adapter module for the Adaptec 154x and compatible host adapters. This module lets you run any combination of SCSI hard disks, tape drives, or CD-ROM drives through any Adaptec AHA-154x host adapter.

The Adaptec AHA-154x is an intelligent ISA bus mastering SCSI host adapter. Its on-board processor and DMA controllers handle the SCSI bus protocol and the DMA transfer of SCSI data into the PC's main memory. **hai154x** uses port I/O, a DMA channel, and an interrupt line, which are configured through the following tunable variables:

HAI154X_BASE Base port
HAI154X_INTR Interrupt level
HAI154X_DMA DMA channel

The following tunable parameters let you set the DMA transfer speed, the bus-on time, and the bus-off time on the SCSI bus:

HAI154X_XFERSPEED DMA transfer speed, from the table below
HAI154X_BUSOFFTIME Host-adapter bus-on time for DMA transfers
HAI154X_BUSONTIME Host-adapter bus-off time for DMA transfers

Variable **HAI154X_XFERSPEED** must be set to one of the values given in the following table.

Setting	*Speed, megabytes/second*
0	5.0
1	6.7
2	8.0
3	10.0
4	5.7

The default setting is '4'.

You can use these parameters to tune the performance of the SCSI bus for your system. For most installations, the default settings should be work well.

haiss — Seagate Host-Adapter Module

haiss is the host-adapter module for host adapters built around the Future Domain TMC-950 or TMC-9C50 chip sets. It works with the following controllers:

Seagate ST01 or ST02
Future Domain TMC-845, 850, 860, 875, or 885
Future Domain TMC-840, 841, 880, or 881

Through this host-adapter module, you can run any combination of SCSI hard disks, tape drives, or CD-ROM drives through any of the above host adapters.

These host adapters map the SCSI bus data and signal lines onto memory addresses on the PC bus. **haiss** then uses standard memory-read and -write operations to access the state of the SCSI bus and the data on it. By default, this controller uses the Intel block-move instruction to transfer data between the device's buffer and the SCSI data-address range. This mode of transfer may be too fast for certain SCSI devices, in which case data can be transferred byte by byte. You can set how **haiss** transfers data; this is described below.

haiss can be used through the following tunable kernel variables:

HAISS_TYPE
The type of the card, as follows:

Type	*Controller*
0	Seagate ST01/02
1	Future Domain TMC-845/850/860/875/885
2	Future Domain TMC-840/841/880/881

HAISS_INTR
The interrupt vector to which the card is set. Although MS-DOS permits you to use this card without interrupts, COHERENT requires that you use interrupts.

HAISS_BASE
The real-mode segment address for the start of the card's RAM area. On all Future Domain and Seagate host adapters with an eight-kilobyte ROM, this is also the base address that is jumpered onto the card. On Seagate host adapters with a 16-kilobyte ROM, this is the base address jumpered on the card plus 0x0200.

HAISS_SLOWDEV
A bitmask of the target identifiers of all SCSI devices whose rate of data transfer is slower than the default transfer mode that the host adapter supports.

These variables are set automatically by the script **/etc/conf/hai/mkdev**, when you use it to configure **hai** for your system; or you can use the command **/etc/conf/bin/idtune** to tune them individually.

haict — Tape Device Module

haict is the device module that controls SCSI tape drives. It works a number of popular quarter-inch and DAT SCSI tape drives

SCSI tape-drive configuration is controlled by the tunable variables **HAI_TAPE_SPEC** and **HAICT_CACHE**.

HAI_TAPE_SPEC is a bitmap of the SCSI identifiers that identify tape drives on your system. For example, if a system has only one SCSI tape drive, and it is assigned SCSI identifier two, then you would set **HAI_TAPE_SPEC** to 0x04, which flips on bit two of that mask. (If you are versed in converting binary values into bit masks, note that the script **/etc/conf/hai/mkdev** handles that conversion for you — all you have to do is tell it what SCSI identifiers are set to which devices, and it does the rest.)

Variable **HAICT_CACHE** sets the size of block of memory that **hai** uses to buffer data that it writes to or reads from the tape drive. You can set this anywhere from zero to 256 kilobytes. The default is 16 kilobytes, which should works well with most tape drives. To tune this variable, use either the command **/etc/conf/bin/idtune** or the script **/etc/conf/hai/mkdev**. Please note that larger tape caches may not necessarily improve tape performance. For example, the program **cpio** for example uses a 5,120-byte buffer that limits the effectiveness of any tape-buffering scheme.

haicd — CD-ROM Device Module

haicd is the device module that controls SCSI CD-ROM drives. It permits you to read data from both audio CDs and CD-ROM that hold an ISO 9660 file system.

Configuration of **haicd** is controlled by the variable **HAI_CDROM_SPEC**, which is a bitmap of the SCSI identifiers that identify CD-ROM drives on your system. For example, if a system has only one SCSI CD-ROM drive, and it is assigned SCSI identifier three, then you would set **HAI_TAPE_SPEC** to 0x08, which flips on bit three of that mask. (If you are versed in converting binary values into bit masks, note that the script **/etc/conf/hai/mkdev** handles that conversion for you — all you have to do is tell it what SCSI identifiers are set to which devices, and it does the rest.)

As of this writing (September 1994), **haicd** has been tested with SCSI CD-ROM drives from Toshiba and NEC. The CD-ROM functions work with both makes of CD-ROM. Please note, however, that the audio functions of the NEC CDR-74 and CDR-84 CD-ROM drives deviate from the SCSI-2 specification considerably; therefore, the audio functions of **haicd** do not work on these drives.

haisd — Hard Disk Device Module

haisd is the **hai** device module that controls SCSI disk drives. Because **hai** allows multiple, overlapping, simultaneous access to the system's SCSI host adapter, the disk drives that **hai** controls operate independently of each other. **haisd** also chains "like" requests for multiple contiguous sectors, which reduces the overhead of starting SCSI commands and thereby inproves performance.

haisd is configured through the tunable kernel variable **HAI_DISK_SPEC**, which is a bitmap of the SCSI identifiers that identify hard-disk drives on your system. For example, if a system has two SCSI disk drives, one with SCSI identifier zero and the other with SCSI identifier one, **HAI_DISK_SPEC** to 0x03, which flips on bits 0 and 1 of that mask. (If you are versed in converting binary values into bit masks, note that the script **/etc/conf/hai/mkdev** handles that conversion for you — all you have to do is tell it what SCSI identifiers are set to which devices, and it does the rest.)

haisd determines partitioning information from the device's minor number as follows:

Bit: 7 6 5 4 3 2 1 0
S I-I-I L-L P-P

The 'S' field is the "special" bit: it distinguishes SCSI disk drives from tape drives. The 'P' fields are a binary value of the partition-table entry for this device, from 0 through 3. If the special bit is set and the partition fields are not 0, then **haisd** assumes that this device is *not* a disk drive and will not allow access to the device. The 'I' fields give the binary value of the SCSI identifier for this device, from zero through seven. This convention is used for all SCSI devices. Finally, the 'L' fields set the logical-unit number field, from 0 through 3. (If you are not skilled at setting bit maps by hand, do not despair: the configuration script **/etc/conf/hai/mkdev** automatically builds an

appropriate device node for each SCSI disk.)

Files

/dev/sd* — Block-special SCSI-disk devices
/dev/Stp* — Block-special SCSI-tape devices
/dev/Scdrom* — Block-special SCSI CD-ROM devices
/dev/rsd* — Character-special SCSI-disk devices
/dev/rStp* — Character-special SCSI-tape devices
/dev/rScdrom* — Character-special SCSI CD-ROM devices

See Also

CD-ROM, device drivers, hai154x, haiss, haicd, haict, haisd, hard disk, tape

Notes

For a list of the block-special files via which you can access the devices that **hai** supports, see the Lexicon entries for **hard disk** and **tape**.

If you are using an Adaptec AHA-1540, AHA-1542C, or AHA-1542CF SCSI host adapter with a drive larger than one gigabyte and extended BIOS support turned on, then you must override the default number of heads to 255 and the number of sectors per track to *63*. Note that when you run the script **/etc/conf/hai/mkdev** (or install COHERENT onto your system), "255" appears as the default choice for the number of heads; however, the default choice for number of sectors is 32. Therefore, when you run **/etc/conf/hai/mkdev** or install COHERENT for a system that has one of the above-named SCSI controllers, you must select the default setting for the number of heads, but you must type "63" when asked for the number of sectors per track.

hai supercedes the older COHERENT device drivers **aha** and **ss**, which were specific to the Adaptec and Future-Domain controllers, and which controlled only SCSI disk drives.

hai was written by Chris Hilton (hilton@mwc.com).

hard disk — Technical Information

The hard disk is the primary means of storing and accessing data under the COHERENT system. This article introduces some aspects of the COHERENT system that affect the care and feeding of your hard disk.

Device Drivers

The COHERENT system comes with two drivers for hard disks: the **at** drivers, for AT-style hard disks (i.e., IDE, ESDI, MFM, or RLL disks); and **hai**, for the SCSI family of hard disks. **hai** is a host adapter-independent SCSI driver and also supports SCSI devices other than hard disks, e.g., SCSI tape. which is the old-style driver for Adaptec SCSI devices. For details on each driver, see its entry in the Lexicon.

The following describes how to enable or disable a given hard-disk driver in your kernel. To disable a hard-drive controller, log in as the superuser **root** and then execute the following commands:

```
cd /etc/conf
bin/idenable -d disk_driver
bin/idmkcoh -o /kernel_name
```

where *kernel_name* is the name you wish to give to the new kernel, and *disk_driver* is one of **at**, **aha**, **ss**, or **hai**.

To enable a hard disk, again log in as **root**; then type the following commands:

```
cd /etc/conf
bin/idenable -e disk_driver
# if you are installing the hai driver:
# hai/mkdev
bin/idmkcoh -o /kernel_name
```

where *disk_driver* is one of **at**, **aha**, **ss**, or **hai**.

Partitioning

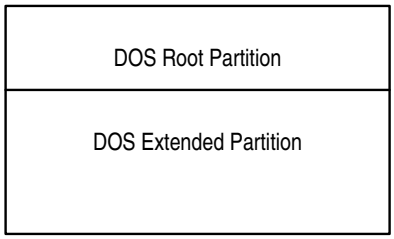
The COHERENT command **fdisk** displays information about how your hard disk is currently configured. You can also use it to repartition your hard disk and reassign partitions from MS-DOS to COHERENT, or vice versa.

This is an extremely powerful command, with which you can create much mayhem on your system. Like any powerful tool, it should be treated carefully and with respect. See the article on **fdisk** in the Lexicon for details on how to use this command.

Partitioning your hard drive can be an uncomplicated procedure. We offer these guidelines in an effort to make it as simple as possible. Before attempting any partitioning you should first back-up all the data currently on your hard drive. If you do not do this you risk losing data permanently. You should also know the correct physical parameters of your hard drive. This information can be obtained from your machine documentation or from the drive manufacturer. It is best not to rely on the parameters given in the BIOS: these may be translation parameters.

If your drive is formatted for MS-DOS, it is advisable to run MS-DOS **fdisk** before you start to install COHERENT. If the whole drive is taken up by DOS partitions, you must use MS-DOS **fdisk** to create a non-DOS area on the drive. It is not sufficient to have an empty MS-DOS logical drive set aside for COHERENT. COHERENT does not recognize MS-DOS logical drives, it only sees the whole partition. The following diagram shows the way the MS-DOS **fdisk** sees your drive:

And the following diagram shows the way the COHERENT **fdisk** sees your drive:



If you use COHERENT **fdisk** to repartition MS-DOS space, you risk causing MS-DOS **fdisk** to hang. One further word of warning. If you have an automated disk formatting and partitioning utility on your MS-DOS partition such as Disk Manager or Speedstor, you should operate it in "manual" mode, not in "automatic".

Some hard drives have more than 1,024 cylinders. COHERENT can only recognize a drive up to this limit. You may have a utility such as Speedstor that allows you to place MS-DOS partitions beyond that boundary. COHERENT will not see those partitions, but you can still access them as usual through MS-DOS.

When partitioning a drive with more than 1,024 cylinders, be sure to run the partitioning utility before you start to install COHERENT. You should create a non-DOS partition that falls completely within the 1,022-cylinder boundary. Your next MS-DOS partition should start no sooner than the 1,026th cylinder.

Adding a COHERENT Partition

The following describes how to add a new COHERENT partition on your hard disk.

During your initial installation of COHERENT, the installation program handled the details of preparing your hard disk for COHERENT. Adding a partition after the system is installed is not difficult, but it requires that you understand the operation of the following commands: **badscan**, **chmod**, **chown**, **fdisk**, **fsck**, **mkfs**, and **mount**. See the Lexicon articles for each of these commands for further information before you attempt to add a partition.

In general, the following steps are required when creating a partition for use by COHERENT. Please note that you must not change the size of your existing root partition, or you may no longer be able to boot COHERENT from the hard disk.

1. Completely back up all partitions on your hard disk. Be sure to back up the COHERENT partitions, as well as any non-COHERENT partitions (e.g., those for MS-DOS or OS/2). Verify that your backups are *readable* and *correct.*
2. Log in as the superuser **root**. Make sure all other users are off the system; then invoke the command **/etc/shutdown**. This shuts down COHERENT and returns the system to single-user mode. Type the command **sync** to flush all buffers.
3. Invoke the COHERENT command **fdisk** and add the COHERENT partition to your disk, as described above. Be sure to write down the device name associated with your new partition (e.g., **/dev/at0c**) and its size.
4. The command **badscan** checks the device for bad blocks. If your partition resides on a non-SCSI device, run the command **badscan** as follows:

```
/etc/badscan -v -o /conf/proto.device raw_device xdevice
```

where *device* specifies the four-character block-special device name for the partition (e.g., **at0c**), *raw_device* is the full device path name for the character-special device associated with the partition (e.g., **/dev/rat0c**), and *xdevice* names the partition-table device for the disk drive (e.g., **/dev/at0x**).

5. Invoke the command **mkfs** to create a COHERENT file system on the new partition, as follows:

```
/etc/mkfs /dev/device /conf/proto.device
```

This invocation forces **mkfs** to use the contents of the "proto" file that **badscan** created when it built the *bad_block* list for the new partition.

6. If need be, use the command **mkdir** to create a directory to use as a *mount point* for the newly created file system. The mount point is the directory onto which this directory's file system will be appended. Usually, this directory is located under '/', also called the *root directory*. You can, however, mount a file system onto any directory that already exists. If you create a new directory (e.g., **/w** or **/mydir**), use the commands **chown** and **chmod** to set an appropriate ownership and mode for for the directory.

7. Edit the file **/etc/mount.all** and add a line of the following form:

```
/etc/mount device /mount_point
```

where *device* is the full path name of the device that specifies your new partition (e.g., **/dev/at0c**), and *mount_point* is the name of the directory that you created in the earlier step.

8. Finally, edit the file **/etc/checklist** and add the character special device name (e.g., **/dev/rat0c**) of the new COHERENT partition to it. This will ensure that COHERENT will automatically run **fsck** on that partition's file system whenever you boot the system. This can be vital in recovering from a system crash.

Adding Another Hard Disk

If you wish to add another hard disk to your system, you may have to run some low-level routines that are hardware specific. See the documentation that accompanies your hardware for details.

In brief, when you install the hard disk, you must partition it, as you did your original hard disk when you first installed COHERENT. If you wish to add non-COHERENT operating systems to one or more partitions, do so first; then add COHERENT to the remaining partitions, as described above.

Changing the Size of the Root Partition

Changing the size of your **root** file system requires that you reinstall COHERENT. It is strongly advised that you back up *all* partitions of your system before you attempt to do this. In addition, to reduce the time involved in restoring your data files, make an additional backup of all directories and files that have changed form your original COHERENT installation. The command **find** will help you locate all such files; see its Lexicon entry for details.

You should then follow the directions given in the release notes for installing COHERENT. Note that when you attempt to install COHERENT over an existing COHERENT partition, COHERENT will ask you if you are sure you know what you're doing before the installation procedure creates a new file system on the partition. Be sure to request that a new file system be created, or the installation will fail.

After installing the COHERENT distribution onto your new root partition, restore any data files and directories from the second set of backups that you performed.

See Also

Administering COHERENT, at, badscan, chmod, chown, fdisk, fsck, hai, ideinfo, mkfs, mount

Notes

For information on how an IDE drive is configured, use the command **ideinfo**. For details on how to use this command, see its entry in the Lexicon.

Some users have attempted to use Norton Utilities or similar tools to rearrange the partition table, only to find that COHERENT no longer boots. That is because the kernel has embedded within it the name of the partition on which it and its root file system live. By using Norton Utilities to shuffle the partition table, the kernel will no longer be able to find any of the files or utilities it needs to boot your system. If you still wish to shuffle your disk's partition table, be sure to change the name of the root device within the kernel *before* you change the partition table.

LEXICON

hash — Command

Add a command to the shell's hash table
hash [-r] [*command ...* **]**

The command **hash** lets you manipulate the Korn shell's hashing facility. A hashed command can be accessed instantly by the shell, without the delay of searching the directories in the **PATH** environmental variable.

When called with an argument, **hash** prints all hashed commands. When called with one or more *command* arguments, it adds *command* to its hash table. The option **-r** removes *command* from the hash table.

Note that before you can use hashing, you must use the **set** command to turn it on. For more information on the Korn shell's hashing feature, see the Lexicon entry for **ksh**.

See Also

commands, ksh

hdioctl.h — Header File

Control hard-disk I/O
#include <sys/hdioctl.h>

Header file **<sys/hdioctl.h>** declares constants and structures used to control hard-disk I/O.

Structure **ide_info** is used by the command **ideinfo** to hold information about IDE drives. It is defined as follows:

```
typedef struct ide_info {
        unsigned short ii_config;            /* Configuration */
        unsigned short ii_cyl;               /* Cylinders (default translation mode) */
        unsigned short ii_reserved;          /* reserved */
        unsigned short ii_heads;             /* heads (default translation mode */
        unsigned short ii_bpt;               /* bytes per track (unformatted) */
        unsigned short ii_bps;               /* bytes per sector (unformatted) */
        unsigned short ii_spt;               /* sectors per track (default translation mode) */
        unsigned short ii_vendor1[3];        /* vendor's unique data */
        unsigned short ii_serialnum[10];     /* serial number in ASCII */
        unsigned short ii_buffertype;        /* buffer type */
        unsigned short ii_buffersize;        /* buffer size in 512-byte sectors */
        unsigned short ii_eccbyteslong;      /* ecc bytes for r/w long */
        unsigned short ii_firmrev[4];        /* firmware revision in ASCII */
        unsigned short ii_modelnum[20];      /* model number in ASCII */
        unsigned short ii_doublewordio;      /* double word transfer flag */
        unsigned short ii_capabilities;      /* capabilities */
        unsigned short ii_reserved2;         /* reserved */
        unsigned short ii_piomode;           /* PIO data transfer timing mode */
        unsigned short ii_dmamode;           /* DMA data transfer timing mode */
        unsigned short ii_reserved3[75];     /* reserved */
        unsigned short ii_vendor2[32];       /* vendor unique data */
        unsigned short ii_reserved4[96]; /* reserved */
} ide_info_t;
```

Field **ii_config** is a set of flags that describes how the drive is configured, as follows:

bit 0 Not used.
bit 1 Disk is hard sectored.
bit 2 Disk is soft sectored.
bit 3 Disk is not MFM encoded.
bit 4 Disk's head switch time is less than 15 microseconds.
bit 5 Spindled motor control option is implemented.
bit 6 Fixed drive.
bit 7 Not used.
bit 8 Disk's transfer rate is less than five megabytes per second.
bit 9 Disk's transfer rate exceeds five megabytes per second but less than or equal to 10 megabytes per second.
bit 10 The disk's transfer rate exceeds ten megabytes per second.
bit 11 The rotational's speed tolerance is greater than 0.5%.
bit 12 The data strobe offset option is available.

bit 13 The track offset option is available.
bit 14 The format speed-tolerance gap required.
bit 15 Not used.

Structure **hdparm_s** holds the drive's attributes. It is configured for binary compatibility with ROM data.

```
typedef struct hdparm_s {
        unsigned char ncyl[2];              /* number of cylinders */
        unsigned char nhead;                /* number heads */
        unsigned char rwccp[2];             /* reduced write curr cyl */
        unsigned char wpcc[2];              /* write pre-compensation cyl */
        unsigned char eccl;                 /* max ecc data length */
        unsigned char ctrl;                 /* control byte */
        unsigned char fill2[3];
        unsigned char landc[2];             /* landing zone cylinder */
        unsigned char nspt;                 /* number of sectors per track */
        unsigned char hdfill3;
} hdparm_t;
```

See Also

hard disk, header files, ideinfo

head — Command

Print the beginning of a file
head [+*n***[bcl]] [***file***]**
head [-*n***[bcl]] [***file***]**

head copies the first part of *file*, or of the standard input if none is named, to the standard output.

The given *number* tells **head** where to begin to copy the data. Numbers of the form *+number* measure the starting point from the beginning of the file; those of the form *-number* measure from the end of the file.

A specifier of blocks, characters, or lines (*b*, **c**, or **l**, respectively) may follow the number; the default is lines. If no *number* is specified, a default of +4 is assumed.

See Also

commands, dd, egrep, sed, tail

Notes

Because **head** buffers data measured from the end of the file, large counts may not work.

header files — Overview

A *header file* is a file of C code that contains definitions, declarations, and structures commonly used in a given situation. By tradition, a header file always has the suffix ".h". Header files are invoked within a C program by the command **#include**, which is read by **cpp**, the C preprocessor; for this reason, they are also called "include files".

Header files are one of the most useful tools available to a C programmer. They allow you to put into one place all of the information that the different modules of your program share. Proper use of header files will make your programs easier to maintain and to port to other environments.

COHERENT includes the following header files:

a.out.h . . . . . Include all COFF header files
acct.h. . . . . Format for process-accounting file
ar.h . . . . . Format for archive files
assert.h . . . . . Define **assert()**
sys/buf.h . . . . . Buffer header
sys/cdrom.h . . . . . Definitions for CD-ROM drives
coff.h . . . . . Format for COHERENT objects
sys/con.h . . . . . Configure device drivers
sys/core.h. . . . . Declare structure of a **core** file
ctype.h. . . . . Header file for data tests
curses.h . . . . . Declare/define **curses** routines
dbm.h . . . . . Header file for DBM routines
sys/deftty.h. . . . . Default tty settings
dirent.h . . . . . Define constant **dirent**

errno.h . . . . . Error numbers used by **errno()**
fcntl.h . . . . . Manifest constants for file-handling functions
sys/fd.h . . . . . Declare file-descriptor structure
sys/fdioctl.h . . . . . Control floppy-disk I/O
sys/fdisk.h . . . . . Fixed-disk constants and structures
sys/filsys.h . . . . . Structures and constants for super block
float.h . . . . . Define constants for floating-point numbers
fnmatch.h . . . . . Constants used with function **fnmatch()**
fperr.h . . . . . Constants used with floating-point exception codes
gdbm.h . . . . . Header file for GDBM routines
gdbmerrno.h . . . . . Define error messages used by GDBM routines
grp.h . . . . . Declare group structure
sys/hdioctl.h . . . . . Control hard-disk I/O
sys/ino.h . . . . . Constants and structures for i-nodes
sys/inode.h . . . . . Constants and structures for memory-resident i-nodes
sys/io.h . . . . . Constants and structures used by I/O
sys/ipc.h . . . . . Declarations for interprocess communication
sys/kb.h . . . . . Define keys for loadable keyboard driver
l.out.h . . . . . Format for COHERENT-286 objects
limits.h . . . . . Define numerical limits
sys/lpioctl.h . . . . . Definitions for line-printer I/O control
math.h . . . . . Declare mathematics functions
mnttab.h . . . . . Structure for mount table
mon.h . . . . . Read profile output files
sys/mount.h . . . . . Define the mount table
mprec.h . . . . . Multiple-precision arithmetic
sys/msg.h . . . . . Definitions for message facility
mtab.h . . . . . Currently mounted file systems
sys/mtioctl.h . . . . . Magnetic-tape I/O control
mtype.h . . . . . List processor code numbers
n.out.h . . . . . Define **n.out** file structure
ndbm.h . . . . . Header file for NDBM routines
netdb.h . . . . . Define structures used to describe networks
path.h . . . . . Define/declare constants and functions used with **path**
poll.h . . . . . Define structures/constants used with polling devices
sys/proc.h . . . . . Define structures/constants used with processes
sys/ptrace.h . . . . . Perform process tracing
pwd.h . . . . . Define password structure
regexp.h . . . . . Header file for regular-expression functions
sys/sched.h . . . . . Define constants used with scheduling
sys/seg.h . . . . . Definitions used with segmentation
sys/sem.h . . . . . Definitions used by semaphore facility
setjmp.h . . . . . Define **setjmp()** and **longjmp()**
sgtty.h . . . . . Definitions used to control terminal I/O
shadow.h . . . . . Definitions used with shadow passwords
sys/shm.h . . . . . Definitions used with shared memory
signal.h . . . . . Define signals
socket.h . . . . . Define constants and structures with **sockets**
sys/stat.h . . . . . Definitions and declarations used to obtain file status
stdarg.h . . . . . Declare/define routines for variable arguments
stddef.h . . . . . Declare/define standard definitions
stdio.h . . . . . Declarations and definitions for I/O
stdlib.h . . . . . Declare/define general functions
sys/stream.h . . . . . Definitions for message facility
string.h . . . . . Declare string functions
stropts.h . . . . . User-level STREAMS routines
termio.h . . . . . Definitions used with terminal input and output
termios.h . . . . . Definitions used with POSIX extended terminal interface
time.h . . . . . Give time-description structure
sys/timeb.h . . . . . Define **timeb** structure
sys/times.h . . . . . Definitions used with **times()** system call
sys/tty.h . . . . . Define flags used with tty processing

sys/types.h . . . . . Define system-specific data types
ulimit.h . . . . . Define manifest constants used by system call **ulimit()**
unctrl.h . . . . . Define macro **unctrl()**
unistd.h . . . . . Define constants for file-handling routines
sys/uproc.h . . . . . Definitions used with user processes
utime.h . . . . . Declare system call **utime()**
utmp.h . . . . . Login accounting information
sys/utsname.h . . . . . Define **utsname** structure
varargs.h. . . . . Declare/define routines for variable arguments
sys/wait.h . . . . . Define wait routines

Compilation Environments and Feature Tests

The COHERENT header files are designed to let you invoke any of several "compilation environments". Each environment offers its own features; in this way, you can easily import code that conforms to the POSIX or ANSI standards, compile device drivers, or otherwise fine tune how your programs are compiled. To invoke a given compilation environment, you must set a *feature test*.

As discussed in the Lexicon article **name space**, the ISO Standard reserves for the implementation every identifier that begins with a single underscore followed by an upper-case letter. The POSIX Standards define several symbols in this name space that the implementation can use as "feature tests" — that is, as symbols that you can use in your source code to determine the presence or absence of a particular feature or combination of features. Note that a feature test applies to an *implementation* of C, rather than to an operating system. A feature test combines aspects of the host system and the language translator: some tests apply to the operating system, some purely to the C translator.

The operating system's header files can define them (for example, **_POSIX_SAVED_IDS**) to control compilation of user code or to deal with optional features, or you can define them (e.g., **_POSIX_C_SOURCE**) to control how the system's header files declare or define constants, types, structures, and macros.

In general, a feature test must either be undefined or have an integer value. It must not be defined as having no expansion text, or expand into a string. For example,

```
# CORRECT
cc -D_POSIX_C_SOURCE=1 foo.c
```

is correct, as is:

```
# CORRECT
cc -U_POSIX_C_SOURCE foo.c
```

However,

```
# WRONG
cc -D_POSIX_C_SOURCE foo.c
```

is incorrect, as is:

```
# WRONG
cc -D_POSIX_C_SOURCE="yes" foo.c
```

This is to permit the constants to be tested with expressions like

```
#if _POSIX_C_SOURCE > 1
```

where an integer value is required. (If the symbol is used in a **#if** test and is undefined, **cpp** replaces it with zero, which is still an integer value). This permits the implementation to use different values of the feature test to invoke different feature sets; and it simplifies testing for complex combinations of feature tests.

Although nearly all feature tests behave as shown above, there are a few exceptions, namely **_POSIX_SOURCE** and **_KERNEL**. These symbols are not defined as having a specific value, so many users do not supply a value. To deal with this, the COHERENT header files check whether these constants have expansion text. If they do not, the header files redefine these constants with value 1, so that they can be used like the other feature tests that the COHERENT header files define.

The following describes the feature tests used in the COHERENT header files, and briefly describes the compilation environment each invokes. Because we are continually adding new features to the kernel, this list is not guaranteed to be complete.

_DDI_DKI
Invoke the environment for compiling device drivers. This environment makes visible all DDI/DKI function prototypes and data definitions, and defines all fundamental data types and structures as mandated by UNIX System V, Release 4.

Please note that this feature test is an COHERENT extension, and is not portable to other operating systems.

_KERNEL
Invoke the environment for compiling the kernel or a device driver. This environment gives code full access to system's private header files. Under COHERENT, this option is equivalent to defining **_DDI_DKI** to value 1, because COHERENT only supports compiling DDI/DKI driver source code from System V, Release 4. This means that the definitions of many fundamental data types such as **pid_t** are changed to the System V, Release 4 definitions rather than the System V, Release 3 definitions used by user code. (This is a System V convention.)

_POSIX_SOURCE
_POSIX_C_SOURCE
Select a "clean" compilation environment, in which the headers defined in the **POSIX.1** or **POSIX.2** standards define no symbols other than the ones that those environments require. Defining **_POSIX_C_SOURCE** with value 1 selects the **POSIX.1** environment, as defined in the POSIX.1 standard. Defining **_POSIX_C_SOURCE** with value 2 selects the **POSIX.2** environment, as defined in the POSIX.2 standard. Defining **_POSIX_SOURCE** has the same effect as defining **_POSIX_C_SOURCE** with value 1.

_STDC_SOURCE
Select a "clean" compilation environment. In this environment, the headers that the ANSI C standard defines define no symbols other than those that the standard requires. This feature test is designed to let you compile conforming Standard C programs that themselves define functions or macros that the COHERENT header files defined in addition to those described in the ANSI standard.

Please note that this feature test is an COHERENT extension, and is not portable to other operating systems.

_SUPPRESS_BSD_DEFINITIONS
This feature test invokes a compilation environment that excludes all definitions that are included for compatibility with Berkeley UNIX. As of this writing, this feature test affects only the header file **<string.h>**, and prevents it from defining the macros **bcopy()**, **bzero()**, **index()**, and **rindex()**. Note that selecting a POSIX or Standard C environment also suppresses these definitions.

Please note that this feature test is an COHERENT extension, and is not portable to other operating systems.

_SYSV3
This feature test invokes a compilation environment in which all fundamental types and data structures have the definitions mandated by UNIX System V, Release 3.

_SYSV4
This feature test invokes a compilation environment in which all fundamental types and data structures have the definitions mandated by UNIX System V, Release 4. As of this writing, this facility is incomplete and used mainly to develop device drivers and extensions to the kernel.

Please note that this feature test is an COHERENT extension, and is not portable to other operating systems.

See Also

#include, C language, cpp, portability

help — Command

Print concise description of command
help [-d*c***] [-f***file***] [-i***file***] [-r] [***command***]...**

help prints a concise description of the options available for each specified *command*. If *command* is omitted, **help** prints a simple description of itself, followed by information about the command given by **$LASTERROR**, which is the last command returning a nonzero exit status.

help provides more information than the usage message printed by a command, but less than the detailed

description given by the **man** command. The primary purpose of **help** is to refresh your memory if you have forgotten an option to *command*.

help looks in **/usr/lib/helpfile** for system information and the file named in environmental variable **$HELP** for user-specific information. Information about a *command* begins with a line

```
#command
```

and ends with the next line beginning with '#' in **/usr/lib/helpfile** or **$HELP.**

help recognizes the following options:

-d*c* Use *c* as the delimiting character within the helpfile, instead of the default **#**.

-f*file* Read the help entries from *file* instead from the default, **/usr/lib/helpfile**.

-i*file* Read the helpfile's index from *file* instead of from the default, **/usr/lib/helpindex**. **help** uses the index to speed its retrieval of an entry, and does not work without it.

-r Rebuild the index. If you modify a helpfile, you must rebuild its index, or **help** will no longer retrieve items correctly.

Example

The following shows how to rebuild the index for helpfile **myhelp**, using **@** as the delimiting character:

```
help -d@ -fmyhelp -imyindex -r
```

Files

/usr/lib/helpfile — Additional system information
/usr/lib/helpindex — Index for helpfile
$HELP — User information
$LASTERROR — Default command help

See Also

apropos, commands, man, Using COHERENT

hmon — Command

Monitor the COHERENT System
hmon

The command **hmon** continually displays a summary of your system's activity. It uses an interactive display with which you can easily send a signal to a selected process.

When you invoke **hmon**, it displays a display that resembles the following:

```
Last PID=91     Total Mem=15684K        Free Mem=7844K (50.01%)
Total=20  Running=1   Zombies=0   Locked=0   Waiting=5   Sleeping=14
PID=91  Idle=75.68%     User= 8.11%     Sys=16.22%
Load= 1.60      Load Averages:  1:3.38  5:1.01  20:0.27

PID   PPID  Username Ksize User  Sys   %User %Sys  Flag tty     S Command
   91    89 fred       148 00:04 00:01  5.41  1.80 4001 ttyp1   R hmon
   89    88 fred       129 00:00 00:00  0.00  0.00 6001 ttyp1   W ksh
   88     1 root       735 00:04 00:19  0.00  1.80 4001 null    S xterm
   86    80 fred       208 00:00 00:00  0.00  0.00 6001 ttyp0   S me
   80    78 fred       129 00:00 00:00  0.00  0.00 6001 ttyp0   W ksh
   79    76 fred       284 00:00 00:07  0.90  9.01 4001 null    S fvwm
   78    76 root       727 00:00 00:01  0.00  0.00 4001 null    S xterm
   76    64 fred        79 00:00 00:00  0.00  0.00 6001 null    S sh
   70    64 root      2423 00:15 00:11  1.80  3.60 6001 console S X
   64    54 fred       105 00:00 00:00  0.00  0.00 6001 color0  W xinit
   56     1 root        28 00:00 00:00  0.00  0.00 4001 com2l   S getty
   55     1 root        28 00:00 00:00  0.00  0.00 4001 com3l   S getty
   54     1 fred       129 00:00 00:00  0.00  0.00 6001 color0  W ksh
   53     1 root        28 00:00 00:00  0.00  0.00 4001 color1  S getty
   52     1 root        28 00:00 00:00  0.00  0.00 4001 color2  S getty
   51     1 root        28 00:00 00:00  0.00  0.00 4001 color3  S getty
   47     1 daemon      55 00:00 00:00  0.00  0.00    1 null    S lpsched
   45     1 root        36 00:00 00:00  0.00  0.00    1 null    S cron
```

The first four lines

```
Last PID=91     Total Mem=15684K        Free Mem=7844K (50.01%)
Total=20  Running=1   Zombies=0   Locked=0   Waiting=5   Sleeping=14
PID=91  Idle=75.68%     User= 8.11%     Sys=16.22%
Load= 1.60      Load Averages:  1:3.38  5:1.01  20:0.27
```

summarize your system's status. The lines that follow summarize each process. Each line contains the following information:

PID The identifier of the process.

PPID The process identifier its parent process. Note that process 1, **init**, has no parent process. For more details on **init**, see its entry in the Lexicon

Username
The login identifier of the user who owns this process.

Ksize The process's size, in kilobytes. Note that this does *not* include memory that the process allocates for itself.

User The amount of user time that this process has consumed.

Sys The amount of system time that this process has consumed.

%User The percent of user time this process has consumed.

%Sys The percent of system time this process has consumed.

Flag The process's flag bits OR'd together, as follows:

PFCORE	00001	Process is in core
PFLOCK	00002	Process is locked in core
PFSWIO	00004	Swap I/O in progress
PFSWAP	00010	Process is swapped out
PFWAIT	00020	Process is stopped (not waited)
PFSTOP	00040	Process is stopped (waited on)
PFTRAC	00100	Process is being traced
PFKERN	00200	Kernel process
PFAUXM	00400	Auxiliary segments in memory
PFDISP	01000	Dispatch at earliest convenience
PFNDMP	02000	Command mode forbids dump
PFWAKE	04000	Wakeup requested

For example, process 8460 has flag "4001". This means that the process is swapped out and and that a wakeup has been requested. This is consistent with the 'S' status, which means that it is sleeping. Note that the flags for swapping do not contain useful information as COHERENT does not yet support demand paging.

tty The port from which the process was launched. This can be the console, a pseudo-tty, or a serial port.

S The process's status, as follows:

- **R** Ready to run (waiting for CPU time)
- **S** Stopped for other reasons (I/O completion, pause, etc.)
- **T** Process is being traced by another process
- **W** Waiting for an existent child
- **Z** Zombie (dead, but parent not waiting)

Command
The name of the program that this process represents.

One of the process lines will be highlighted. You can shift the line of highlighting by pressing the keys (ª) and (°). When a process line is highlighted, you can send that process a signal simply by pressing a key, as follows:

1 Send signal **HUP**. Equivalent to typing **kill -1**.

2 Send signal **INTR**. Equivalent to typing **kill -2**.

3 Send signal **QUIT**. Equivalent to typing **kill -3**.

9 Send signal **KILL**. Equivalent to typing **kill -9**.

Whether the signal has any effect will, of course, depend upon the degree of control you have over that process.

To refresh the **hmon** screen, type **L**. To quit, type **Q**.

See Also

commands, ps

Notes

hmon reads the free memory from **/dev/freemem**. If this device does not exist on your system, create it as follows:

```
mknod /dev/freemem c 0 12
chmog 444 sys sys /dev/freemem
```

hmon uses **curses** to manage its display. Your screen will not appear properly if the environmental variable **TERM** is not set correctly for the display device you are using, or if its **terminfo** entry is not correct.

hmon was written by Harry C. Pulley, IV (hpulley@uoguelph.ca).

HOME — Environmental Variable

User's home directory
HOME=*home directory*

The environmental variable **HOME** name's the user's home directory. Some commands use this name by default if they require the name of a directory and none is supplied. For example, if you type the change directory command **cd** without an argument, it will change the current directory to the one named by the **HOME**.

See Also

environmental variables

hosts — System Administration

Names and addresses of hosts on the local network
/etc/hosts

The file **/etc/hosts** gives the name and Internet-protocol (IP) address of remote hosts with which your system can communicate via a network.

Each line within **hosts** describes one host on the network. A description of a host begins with that host's IP address, in normal "dot" notation. This is followed by its name and any aliases it has — that is, other names that also refer to that host. For example, consider the following:

```
666.16.16.27      accounting  acct  beancounters
666.16.16.2 president   boss
666.16.3.5  engineering
```

As you can see, a given host can have more than one alias. Aliases need not be terse; however, you should not use an alias name that you would not want the users of that host to see.

An IP address can appear on more than one line. For example, entry

```
137.229.10.39     raven raven.alaska raven.alaska.edu
```

can also be rendered as:

```
137.229.10.39     raven
137.229.10.39     raven.alaska
137.229.10.39     raven.alaska.edu
```

You may find this to be more legible. However, if you need to change this host's IP address, you must be careful to change every entry, or trouble will result.

/etc/hosts must include the following standard entries:

```
127.1          localhost
127.0.0.1      loopback
```

When you specify only two parts of an Internet address, the second part represents the final three bytes of that address. Thus, the addresses **127.1** and **127.0.0.1** are, in fact, the same address.

The address **127.1** by convention names the local host. Packets sent to this address return to the local host: they do not go onto the Ethernet. This feature is useful in debugging software. The host names **localhost** and **loopback** are also conventional names for your local host.

/etc/hosts should also contain a separate entry for your local host's Internet address and name. You set the name for your system when you installed COHERENT. To change your system's name, edit the file **/etc/uucpname**.

See Also

Adminstering COHERENT, hosts.equiv, inetd.conf, networks, protocols, services, uucpname

hosts.equiv — System Maintenance

Name equivalent hosts
/etc/hosts.equiv

File **/etc/hosts.equiv** names every host on your network whose users are equivalent those on your system.

For example, if system **mwc** names system **lepanto** in its copy of **/etc/hosts.equiv**, then **mwc** assumes that user **fred** on **lepanto** is the same person as user **fred** on **mwc**.

See Also

Administering COHERENT, hosts inetd.conf, networks, protocols, services

hosts.lpd — System Administration

Local system name and domain
/etc/hosts.lpd

File **hosts.lpd** gives your system's name and domain, using dot notation. For example:

```
lepanto.mwc.com
```

Your system's name should be the same as that set in file **/etc/uucpname**, and its domain should be the same as that set in file **/etc/domain**.

See Also

Adminstering COHERENT, domain, hosts, hosts.equiv, inetd.conf, networks, protocols, services, uucpname

hp — Command

Prepare files for Hewlett-Packard LaserJet printer
hp [-acflr] [-i*marg* **] [-t***top* **] [-p***lines* **] [** *file ...* **]**

The command **hp** translates **nroff** font specifications into the correct escape sequences for an HP LaserJet compatible printer. It also allows the user to set indentation, page length, landscape mode, and so on. Because some LaserJet printers stack pages in reverse order as they are printed, **hp** can put pages out in reverse order.

hp recognizes the following options:

-f Print pages in the normal order. This is the default.

-i*marg* Set the page indentation to *marg*.

-l Print pages in landscape mode.

-p*lines* Set the page length to *lines*.

-r Print pages in reverse order (for LaserJet I).

-t*top* Set the top margin to *top*.

Example

To generate listings of all C programs in the current directory, enter the command

```
pr *.c | hp | hpr -B
```

See Also

commands, hpd, printer

hpd — System Administration

Spooler daemon for laser printer
/usr/lib/hpd

hpd is the daemon that prints jobs spooled by the command **hpr**. All jobs are printed on the printer that is accessed through device **/dev/hp**. For information on this device, and on printer management in general, see the Lexicon entry **printer**.

The command **hpr** invokes **hpd** automatically. If there is no printing to do, or if another daemon is already running (as indicated by the file **dpid**), **hpd** exits immediately. Otherwise, it searches the spool directory for control files of listings to print. A control file contains the names of files to print, the user name, banner pages, and files to be removed upon completion.

hpd does not print listings in any particular order. There is no prioritization of printing, either by size or by requester.

The command **hpskip** aborts or restarts printing of the job currently being printed by **hpd**.

Files

/dev/rhp — Raw device for LaserJet printer
/usr/spool/hpd — Spool directory
/usr/spool/hpd/cf* — Control files
/usr/spool/hpd/df* — Data files
/usr/spool/hpd/dpid — Lock and process id

See Also

Administering COHERENT, despooler, hpr, hpskip, init, lpd, printer

Notes

Beginning with release 4.2, COHERENT also includes the printer daemon **despooler**, which prints files spooled with the command **lp**. For details on how COHERENT manages printing, see the Lexicon entry for **printer**.

hpr — Command

Spool a job for printing on the laser printer
hpr [-Bcemnr] [-b *banner***] [-f** *fontnum***] [***file ...***]**

The command **hpr** spools each *file* for printing on the Hewlett-Packard LaserJet printer. If no *file* is named on the command line, **hpr** spools what it reads from the standard input.

hpr recognizes the following options:

-B Suppress printing of a banner page. Note that **hpr** outputs its banner in plain text; therefore, if you have a PostScript printer, you *must* use this option. If you do not, your printer will hang.

-b *banner* Print *banner* on the banner page. The default banner is the user's login identifier.

-c Copy each *file* into the spooling directory, instead of reading the file from its home directory. This option lets you edit a *file* before it has finished printing.

-e Erase all "soft fonts" from the printer's memory.

-f *fontnum file1 ... fileN*
Load the Hewlett-Packard "soft fonts" stored in files *file1* through *fileN* into the printer's memory; set the font identifiers to begin at *fontnum*.

-m Write a message on the user's terminal when printing completes.

-n Do not send a message (default).

-r Remove the files when they have been spooled.

The command **hpskip** aborts or restarts printing of the file that is currently being printed. The command **hp** converts **nroff** output into a form usable by the LaserJet.

Examples

To print the file **foo** on the LaserJet, type:

```
hpr -B foo
```

The following example loads the soft fonts in files **foo**, **bar**, and **baz** into the printer's memory, and sets their font identifiers to begin at 15:

```
hpr -f 15 foo bar baz
```

Files

/dev/rhp — Raw device for LaserJet printer
/usr/lib/hpd — Line-printer daemon for LaserJet printer
/usr/spool/hpd — Spool directory for LaserJet printer
/usr/spool/hpd/dpid — Daemon lockfile

See Also

commands, hp, hpd, hpskip, printer

Notes

Beginning with release 4.2, COHERENT also includes the **lp** print spooler. **lp** offers a more sophisticated way to manage printers, especially on machines that support multiple printers of the same type. For details, see the Lexicon entries for **printer** and **lp**.

hpskip — Command

Abort/restart current job on Hewlett-Packard LaserJet
hpskip [-r]

The command **hpskip** aborts or restarts the job being printed on the printer plugged into device **/dev/hp**. The job must have been spooled with the command **hpr**.

By default, **hpskip** aborts the job and prints a message on the user's terminal. When invoked with the **-r** option, it restarts the printing of the current job. This is useful when a printing is spoiled due to, say, a paper jam.

Files

/usr/lib/hpd — LaserJet printer daemon
/usr/spool/hpd — Spool directory
/usr/spool/hpd/dpid — Daemon lockfile

See Also

commands, hpd, hpr, lpskip, printer

Notes

To cancel jobs spooled with the command **lpr**, use the command **lpskip**. To cancel or reprint jobs spooled with the command **lp**, use the commands **cancel** and **reprint**. See the Lexicon entry **printer** for details.

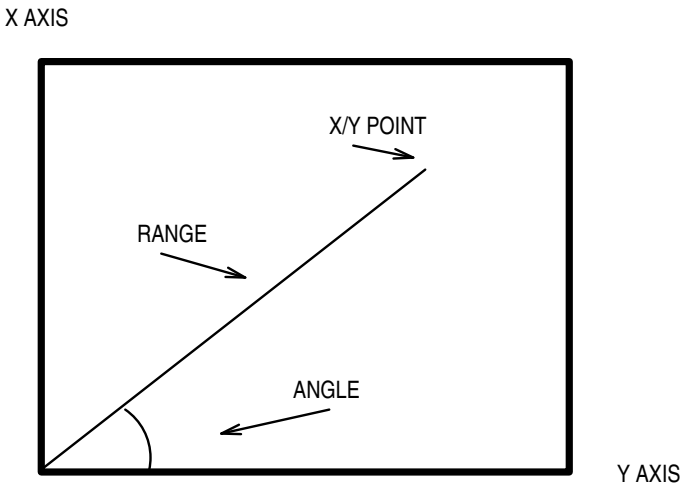
hypot() — Mathematics Function (libm)

Compute hypotenuse of right triangle
#include <math.h>
double hypot(*x***,** *y***) double** *x***,** *y***;**

hypot() computes the hypotenuse, or distance from the origin, of its arguments *x* and *y*. The result is the square root of the sum of the squares of *x* and *y*.

Example

The following example demonstrates the functions **hypot()** and **atan2()**. It converts an X/Y pair of rectangular coordinates into polar coordinates. Thus, an X/Y pair of 1,1 produces a range of 1.41 and 45°; and an X/Y pair of 3,4 would produce a range of five and 36.87°. The following sketch illustrates this:



This example was written by Brent Seidel (brent_seidel@chthone.stat.com):

```
#include <stdio.h>
#include <math.h>

main()
{
      double x, y, angle, range;
      char  buffer[100];

      printf("Enter the X/Y pair: ");
      fflush(stdout);
      gets(buffer);
      sscanf(buffer, "%lf,%lf", &x, &y);

      range = hypot(x, y);
      angle = atan2(x, y);
      printf("The range is %f\n", range);
      printf("The angle is %f radians or %f degrees.\n",
            angle, angle * 180.0/PI);
}
```

See Also

cabs(), libm

i-node — Definition

COHERENT system file identifier

Each file on a COHERENT file system is identified by a unique number, called an *i-node number* or *i-number*. Each i-node contains information about a file: its mode, link count, user identifier, group identifier, size, location on the file system, access time, modify time, and creation time.

The user refers to a file by a file name, stored in a directory; the directory entry identifies the file by its i-node number. A device and i-node number together uniquely specify a file. The headers **ino.h** and **i-node.h** define, respectively, disk i-nodes and memory i-nodes.

See Also

Using COHERENT

icheck — Command

i-node consistency check
icheck [-s] [-b *N ...***] [-v]** *filesystem ...*

Each block in a file system must be either free (i.e., in the free list) or allocated (i.e., associated with exactly one i-node). **icheck** examines each specified *filesystem*, printing block numbers that are claimed by more than one i-node, or claimed by both an i-node and the free list. It also checks for blocks that appear more than once in the block list of an i-node or in the free list.

The option **-v** (verbose) causes **icheck** to print a summary of block usage in the *filesystem*. The option **-s** causes **icheck** to ignore the free list, to note which blocks are claimed by i-nodes, and to rebuild the free list with the remainder. A list of block numbers may be submitted with the **-b** flag; **icheck** prints the data structure associated with each block as the file system is scanned.

The raw device should be used, and the *filesystem* should be unmounted if possible. If this is not possible (e.g., on the root file system) and the **-s** option is used, the system must be rebooted immediately to expunge the obsolete superblock.

The exit status bits for a bad return are as follows:

0x01 Miscellaneous error (e.g. out of space)
0x02 Too hard to fix without human intervention
0x04 Bad free block
0x08 Missing blocks
0x10 Duplicates in free list
0x20 Bad block in free list

See Also

clri, commands, dcheck, fsck, ncheck, sync, umount

Diagnostics

The message “dups in free” indicates a block is in the free list more than once. “bad freelist” indicates the presence of bad blocks on the free list. A “bad” block is one that lies outside the bounds of the file system. A “dup” (duplicated) block is one associated with the free list and an i-node, or with more than one i-node. All the errors above *must* be corrected before the file system is mounted. “bad ifree” means allocated i-nodes are on the free i-node list; this is inconsequential.

This command has largely been replaced by **fsck**.

id — Command

Print user and group IDs and names
id

The command **id** prints the user's real user ID and group ID. It also prints the effective IDs if they differ from the real IDs.

See Also

commands, getuid(), geteuid(), getgid(), getegid()

idbld — Command

Reconfigure the COHERENT kernel
/etc/conf/bin/idbld [-o *kernelname* **]**

The command **/etc/conf/bin/idbld** lets you reconfigure the entire COHERENT kernel. It systematically invokes all **mkdev** scripts in the subdirectories of **/etc/conf**. Each **mkdev** script, in turn, walks you through the task of formatting of one COHERENT's device drivers. This duplicates much of the work you performed when you first installed COHERENT onto your system.

After all of the **mkdev** scripts have been run, **idbld** invokes command **/etc/conf/bin/idmkcoh** to create a new kernel. Option **-o** tells **idbld** to name the new kernel *kernelname*. If you do not name this option, **idbld** by default names the new kernel **/coherent**.

See Also

commands, idenable, idmkcoh, idtune

ideinfo — Command

Display information of an IDE hard-disk drive
ideinfo [-c] /dev/at??

The command **ideinfo** displays information about device **/dev/at**??, which names a partition on an IDE hard disk. For example, command

```
ideinfo /dev/at0a
```

displays information about the first IDE drive on your system (drive 0). Among other things, this command displays the disk's manufacturer, the number of cylinders, header, sectors, and the number of bytes per sector on the disk.

Option **-c** tells **ideinfo** also to display how the device is partitioned.

See Also

at [device driver], commands

Notes

This command fails if the device is not an IDE hard drive.

idenable — Command

Enable or disable a device driver
/etc/conf/bin/idenable [-f *file***] [-de]** *driver*

The command **idenable** lets you enable or disable a device driver within the COHERENT kernel. *driver* is the device driver to enable or disable

The flag **-e** tells **idenable** to enable *driver*. This is the default.

The flag **-d** tells **idenable** to disable it.

For example, to enable STREAMS and disable the pseudo-tty driver **pty**, use the following commands:

```
/etc/conf/bin/idenable streams
/etc/conf/bin/idenable -d pty
```

idenable's command line can name more than one driver. For example, the command

```
/etc/conf/bin/idenable streams -d pty
```

is the equivalent of the two commands given above. The command line is parsed from left to right, so whatever you say last about a driver is what ultimately happens.

The option **-f** forces **idenable** to enable a driver. If **idenable** is directed to enable a device that will conflict with another enabled device in some way, it normally reports the conflict and not make the change. **-f** directs **idtune** to "force" the driver to be enabled by simply shutting off all other drivers with which a conflict occurs. For example, this is used with keyboard drivers, only one of which can occupy a major number at a time.

To implement your changes, you must then invoke the command **/etc/conf/bin/idmkcoh** to build a new kernel, which will reflect your changes, and then boot the new kernel.

idenable works by modifying the file **/etc/conf/sdevice**. It consists of a series of lines with the following format:

```
streams N   0   0   0   0   0x0   0x0   0x0   0x0
console Y   0   0   0   0   0x0   0x0   0x0   0x0
cohmain Y   0   0   0   0   0x0   0x0   0x0   0x0
```

The first column names the driver in question. The second column indicates whether it is incorporated into the kernel. The other columns give "magic cookies" that describe how the driver works.

You can read **/etc/conf/sdevice** to see how your kernel is currently configured. Note, however, that you must *never* modify **sdevice** by hand. **idenable** performs consistency checking to ensure, for example, that you do not load two competing keyboard drivers or hard-disk drivers. If you modify **sdevice** by hand, you run the risk of building a kernel that that will not boot or will trash your file system.

See Also

cohtune, commands, device drivers, idmkcoh, idtune, vtkb, vtnkb

idle — Device

Device that returns system's idle time
/dev/idle

/dev/idle is the device from which you can read the system's idle time. It has major device 0, the same as **/dev/null** and **/dev/cmos**; and has minor number 11. This non-portable device node is used exclusively for tracking system load. Its driver recognizes the system calls **open()**, **ioctl()**, and **close()**, but not **read()** or **write()**.

The only available **ioctl()** for **/dev/idle** writes a pair of **long**s to an address that you supply. The **long** at the lower address contains the number of system idle clock ticks (or, more precisely, the number of ticks at the end of which the system was idle) that have occurred since system startup. The **long** at the higher address contains the total number of clock ticks that have occurred since system startup. To estimate system load during a specific interval of time, perform the **ioctl()** for **/dev/idle** at the start and end of an interval.

Example

The following program prints system load over a five-second interval. To see a nonzero load percentage, run it concurrently with a CPU-intensive process.

```
#include <sys/null.h>

main()
{
      long x[2];  /* tick values at start of interval */
      long y[2];  /* tick values at end of interval */

      long delta_idle, delta_lbolt;
      int fd;

      /* We need to open a device before we can ioctl it. */
      fd = open("/dev/idle", 0);

      /* Get tick values at start of interval. */
      ioctl(fd, NLIDLE, x);

      /* Sleep during the interval. */
      sleep(5);
```

```
	/* Get tick values at end of interval. */
	ioctl(fd, NLIDLE, y);

	/* Compute number of system idle ticks during the interval. */
	delta_idle = y[0] - x[0];

	/* Compute total number of clock ticks during the interval. */
	delta_lbolt = y[1] - x[1];

	/* System is loaded when it isn't idle, so system load factor
	 * is 100% minus the percentage of system idle time.
	 */
	printf("system load = %ld%%\n",
		100L - (100L * delta_idle)/delta_lbolt);
	close(fd);
}
```

See Also

device drivers, ioctl(), null

idmkcoh — Command

Build a new kernel
idmkcoh [-o *kernelfile* **]**

The command **idmkcoh** creates a new bootable kernel. The kernel incorporates any changes that you may have made with the commands **idenable**, **idtune**, or **cohtune**. For details on how to use these commands, see their entries in the Lexicon. The changes you have made will take effect as soon as you boot the kernel that **idmkcoh** creates.

By default, **idmkcoh** writes the new kernel into file **/coherent**. The option **-o** tells **idmkcoh** to write the kernel into file **kernelfile** instead.

See Also

cohtune, commands, idbld, idenable, idtune, mdevice, mtune, sdevice, stune

idtune — Command

Set a tunable system value
/etc/conf/bin/idtune [-fm] *switch value*

The command **idtune** lets you "tune" a variable in the COHERENT kernel. It also performs some sanity checking, to help ensure that you do not set a value to an impossible value. It and the related command **cohtune** largely replace the need for the command **patch**.

To use **idtune**, simply invoke it along with the variable you wish to modify and the value to which you wish to set it. For example, to change the maximum size of a shared-memory segment to 128,000 bytes, type the command:

```
/etc/conf/bin/idtune SHMMAX 128000
```

For the new setting to come into effect, you must use the command **/etc/conf/bin/idmkcoh** to build a new kernel, and then boot the newly built kernel.

idenable recognizes the following two command-line options:

-f **idtune** by default will ask you if you are sure that you want to make a given change. This option suppresses that behavior.

-m Check that the value of *switch* is no less than *value*. If the value *switch* is less than *value*, then **idtune** raises it to *value*; otherwise, it leaves the value of *switch* alone.

idtune works by modifying the file **/etc/conf/stune**, which holds the values of system variables that users can set. **stune** consists of a series of entries like the following:

```
LOOP_COUNT            16
DUMP_USERS            2
MONO_COUNT            0
VGA_COUNT             4
```

The allowed range of values for a given variable is set in file **/etc/conf/mtune**, which consists of a series like the following:

```
STREAMS_HEAP    8192    32768   131072
MONO_COUNT      0       4       8
VGA_COUNT       0       4       8
NBUF_SPEC       0       0       5000
NHASH_SPEC      0       1021    5000
NINODE_SPEC     0       128     1024
NCLIST_SPEC     0       64      1024
```

The first column gives the variable, the second gives its minimum allowable value, the third gives its default value, and the last its maximum value.

You can read **mtune** and **stune** to see what kernel variables you can set, and to find the range of values allowed for each. Note, however, that you must *never* modify **stune** or **mtune** by hand. If you do so, you may build a kernel that is unbootable or that trashes your file system.

See Also

cohtune, commands, idenable, idmkcoh

ieee_d() — General Function (libc)

Convert a double from DECVAX to IEEE format
int
ieee_d(*idp***,** *ddp***)**
double **idp*, **ddp*;*

ieee_d() converts a **double** from DECVAX format to IEEE format. *ddp* points to a DECVAX-format **double** to convert. *idp* points to a destination for the converted IEEE value. *idp* may be identical to *ddp* for in-place conversion. The DECVAX significand is truncated, not rounded.

ieee_d() always returns zero, because the conversion always succeeds.

For a description of the IEEE and DECVAX formats for floating-point numbers, see the Lexicon article for **float**.

See Also

decvax_d(), decvax_f(), float, ieee_f(), libc

ieee_f() — General Function (libc)

Convert a float from DECVAX to IEEE format
int
ieee_f(*ifp***,** *dfp***)**
float **ifp*, **dfp*;*

ieee_f() converts a **float** from DECVAX format to IEEE format. *dfp* points to a DECVAX-format **float** to convert. *ifp* points to a destination for the converted IEEE value. *ifp* may be identical to *dfp* for in-place conversion. The DECVAX significand is truncated, not rounded.

ieee_f() always returns zero, because the conversion always succeeds.

For a description of the IEEE and DECVAX formats for floating-point numbers, see the Lexicon article for **float**.

See Also

decvax_d(), decvax_f(), float, ieee_d(), libc

if — Command

Execute a command conditionally
if *sequence1* **then** *sequence2* **[elif** *sequence3* **then** *sequence4***] ... [else** *sequence5***] fi**

The shell construct **if** executes commands conditionally, depending on the exit status of the execution of other commands.

First, **if** executes the commands in *sequence1*. If the exit status is zero, it executes the commands in *sequence2* and terminates. Otherwise, it executes the optional *sequence3* if given, and executes *sequence4* if the exit status is zero. It executes additional **elif** clauses similarly. If the exit status of each tested command sequence is nonzero, it executes the optional **else** part *sequence5*.

Because the shell recognizes a reserved word only as the unquoted first word of a command, each **then**, **elif**, **else**, and **fi** must either occur unquoted at the start of a line or be preceded by ';'.

The shell executes **if** directly.

Example

For an example of this command, see the entry for **trap**.

See Also

commands, ksh, sh, test

if — C Keyword

Introduce a conditional statement

if is a C keyword that introduces a conditional statement. For example,

```
if (i==10)
      dosomething();
```

will **dosomething** only if **i** equals ten.

if statements can be used with the statements **else if** and **else** to create a chain of conditional statements. Such a chain can include any number of **else if** statements, but only one **else** statement.

See Also

C keywords, else
ANSI Standard, §6.6.4.1

IFS — Environmental Variable

Characters recognized as white space

The environmental variable **IFS** lists the characters that the shell recognizes as white space.

See Also

environmental variables, ksh, sh

index() — String Function (libc)

Find a character in a string
#include <string.h>
char *index(*string***,** *c***) char ****string***; char** *c***;**

index() scans the given *string* for the first occurrence of the character *c*. If *c* is found, **index()** returns a pointer to it. If it is not found, **index()** returns NULL.

Note that having **index()** search for a NUL character will always produce a pointer to the end of a string. For example,

```
char *string;
assert(index(string, 0)==string+strlen(string));
```

will never fail.

Example

For an example of this function, see the entry for **strncpy()**.

See Also

libc, pnmatch(), strchr(), string.h, strrchr(), string.h

Notes

You *must* include header file **string.h** in any program that uses **index()**, or that program will not link correctly.

index() is now obsolete. You should use **strchr()** instead.

inet_addr() — Sockets Function (libsocket)

Transform an IP address from text to binary
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
ulong inet_addr(*ip_address***)**
char ******ip_address***;***

The function **inet_addr()** translates an Internet-protocol (IP) address from text into binary format. *ip_address* gives the address where the string that holds the IP address resides in memory.

If all goes well, **inet_addr()** returns the binary address that it built from *ip_address*. If, however, *ip_address* points to a malformed Internet address, **inet_addr()** returns -1.

An IP address consists of four bytes. The four bytes normally are written as four numbers that are separated by periods; for example, "199.3.32.100". This way of rendering an IP address is called *dot notation*. Each byte can as a written as a decimal, octal, or hexadecimal number. By default, a numbers is written in decimal; a leading "0x" or "0X" indicates hexadecimal, and a leading '0' indicates octal.

When **inet_addr()** translates an IP address from text into binary, it simply transforms the four numbers as written into four bytes, which it writes into the four bytes of an unsigned long (32-bit) integer, from left to right, without regard to the machine's byte ordering. This means, among other things, that you cannot perform arithmetic on the address that **inet_addr()** returns — not even to increment or decrement it.

The IP address to which *ip_address* points can have any of the following four forms:

first.second.third.fourth
first.second.third
first.second
first

When the string to which *ip_address* points specifies all four parts of the Internet address, **inet_addr()** writes all four, from left to right, into the long integer that it returns.

When *ip_address* points to a three-part address, **inet_addr()** interprets the last (third) part as a 16-bit value, which it writes into into the rightmost two bytes of the network address. When *ip_address* points to a two-part address, **inet_addr()** interprets the second part as a 24-bit, which it writes into the rightmost three bytes of the network address.

When *ip_address* points to a one-part address, **inet_addr()** simply transforms it into an integer without shuffling any bytes.

See Also

inet_network(), libsocket

Notes

Because COHERENT does not yet support networking, **inet_addr()** is a dummy function that always returns zero.

inet_network() — Sockets Function (libsocket)

Transform an IP address from text to an integer
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
ulong inet_network(*ip_address***)**
char ******ip_address***;***

Function **inet_network()** translates an Internet-protocol (IP) address from text into a long integer. *ip_address* gives the address where the string that holds the IP address resides in memory.

If all goes well, **inet_network()** returns the integer that it built from *ip_address*. If, however, *ip_address* points to a malformed Internet address, **inet_network()** returns -1.

An IP address consists of four bytes. The four bytes normally are written as four numbers that are separated by periods; for example, "199.3.32.100". This way of rendering an IP address is called *dot notation*. Each byte can as a written as a decimal, octal, or hexadecimal number. By default, a numbers is written in decimal; a leading "0x" or

"0X" indicates hexadecimal, and a leading '0' indicates octal.

Unlike the function **inet_addr()**, **inet_network()** translates *ip_addr* into an unsigned long (32-bit) integer. This is the form suitable for a network address.

See Also

inet_addr(), libsocket

inetd.conf — System Administration

Configure the Internet daemons

File **/etc/inetd.conf** holds information that configures the Internet daemons on your system.

See Also

Administering COHERENT, hosts, hosts.equiv, inetd networks protocols services

infocmp — Command

De-compile a terminfo file
infocmp [*file ...* **]**

infocmp reads a set of compiled terminal information, decodes its contents, and writes the decoded information to the standard output. It does its best to recreate the **terminfo** source from which the set of information had been compiled.

file must hold compiled **terminfo** information. If no *file* is named on the command line, **infocmp** reads the standard input.

infocmp first seeks *file* in the directory named by the environmental variable **TERMINFO**. If this variable has not been set, it seeks *file* in the default directory **/usr/lib/terminfo**. Thus, you can type the command

```
infocmp ansipc
```

in any directory and **infocmp** builds the appropriate path on its own.

In case of emergency, the output of **infocmp** can be piped to the **terminfo** compiler **tic**.

See Also

commands, term, tic, terminfo

Notes

infocmp was written by Pavel Curtis of Cornell University. It was ported to COHERENT by Udo Munk, with additional changes by Mark Williams Company.

init — System Administration

System initialization
/etc/init

COHERENT invokes processes in special order. The kernel invokes the command **init** as the initial process in the system. **init** runs as long as the system remains up. **init** is the first process that the kernel starts. The kernel always gives this process identifier 1.

init has two primary tasks: First, it guides the system through the latter stages of booting and entering multi-user mode. Second, it launches the appropriate processes so that users can log in and log out of COHERENT correctly. The rest of this article describes how **init** performs these tasks.

Booting and Entering Multi-user Mode

The following that **init** performs as it guides the system through entering multi-user mode.

First, if file **/usr/adm/wtmp** exists, **init** records there the date and time at which the system is being booted.

init then executes the shell script **/etc/brc**. This script usually loads the keyboard table and invokes the command **fsck** to check the file systems for errors If this script returns zero, then **init** enters multi-user mode; if not, it spawns the single-user shell.

When the user at the console terminates the single-user shell (usually by typing **<ctrl-D>**), **init** executes script **/etc/rc** and brings the system up to the multiuser state. **/etc/rc** performs such chores as setting the time zone, removing stale temporary files and lock files, and initializing the modem. If you wish, it can invoke the command

accton to enable process accounting.

init then reads file **/etc/ttys**. For every local, enabled line, **init** spawns the command **getty** with two arguments: the name of the port, and its speed (as given in **/etc/ttys**). Before it spawns a **getty**, **init** sets the group number for a new process group.

For a remote line, **init** spawns another copy of itself, which waits for carrier detect. Each **init** process spawned for a remote line also spawns **getty** when it detects a carrier signal on its port. (Note that this use of a second **init** process is unique to COHERENT.)

init then waits for the termination of its child processes. If one of the **getty** processes terminates, **init** respawns it. If another process terminates, **init** waits to receive its return value, so the process does not become a "zombie".

Logging In Users

The following describes how **init** logs users in.

As mentioned in the previous section, **init** invokes process **getty** for each enabled device on the system. **getty** and passes it as arguments the speed and the device upon which it should run. **getty** waits until someone tries to log in. Under COHERENT, **getty** sets the tty's line speed and local-edit characters and prompts the user to log in. It then locks the port, and invokes **login** with what the user has typed.

At this point, the command **login** takes over the task of logging in the user. **login** first asks the user for his password. It then reads the encrypted password from file **/etc/passwd**. If the password consists of one asterisk '*', **login** then reads the encrypted password from file **/etc/shadow**. It then compares the retrieved password with what the user has typed.

If the user has entered his password correctly, **login** executes various "housekeeping" tasks needed to get the user up and running under COHERENT. These include It records in file **/usr/adm/utmp** the fact of the user's logging in, which lets the system keep a running talley of who is logged into the system. For details on how **login** manages the task of logging in, see its entry in the Lexicon.

As its last action, **login** invokes the program named in **/etc/passwd**. This usually is an interactive shell (i.e., **sh** or **ksh**), but can also be another program (e.g., **uucico**). If **login** invokes an interactive shell, it does so with the first character of its **argv[0]** set to '-', so that the shell knows that it is a login shell. (For example, if **login** invokes **ksh**, its **argv[0]** is **-ksh**.)

The shell first executes file **/etc/profile**, then **$HOME/.profile**. Once these are executed, the shell displays its command-line prompt, and the user is ready to begin issuing commands to COHERENT

When the login shell terminates, **init** removes its record from file **/usr/adm/utmp**. Then it reopens the appropriate terminal and invokes **getty**, as described above. The device is now ready to receive another login.

Signals

init accepts two signals. When it receives **SIGQUIT**, it re-reads **/etc/ttys**, spawns **getty**s on newly enabled devices, and stops **getty**s on disabled devices. The command

```
kill quit 1
```

sends **SIGQUIT** to the **init** process. When **init** receives **SIGHUP**, it sends **SIGKILL** to every process and brings the system down to single-user mode. The command

```
kill -1 1
```

sends **SIGHUP** to the **init** process.

Files

/dev/console — Console terminal
/dev/tty?? — Terminal devices
/etc/rc — initialization command file
/etc/brc — Boot command file
/etc/ttys — Active terminals
/etc/utmp — Logged in users
/usr/adm/wtmp — Login accounting data
/usr/spool/uucp/LCK..* — Terminal locks

LEXICON

See Also

Administering COHERENT, getty, kill, ksh, login, sh, ttys

initgroups() — General Function (libc)

Initialize the supplementary group-access list
#include <sys/types.h>
#include <grp.h>
int initgroups(*user*, *basegid***)**
const char **user*; **gid_t** *basegid*;

The "supplemental group-access list" is the list of group identifiers that are used in addition to the effective group identifier when determining the level of access that a process has to a file. The function **initgroups()** initializes the supplemental group-access list to the groups to which *user* belongs.

user is the login identifier of the user in question. *basegid* identifies that user's base group, as set in the file **/etc/passwd**. **initgroups()** calls the library function **getgrent()** to read from **/etc/group** all of the groups to which *user* belongs (in addition to her base group). It then calls **setgroups()** to initialize the supplementary group-access list to *user*'s base group and the additional groups returned by **getgrent()**.

If all goes well, **initgroups()** modifies the supplementary group-access list returns zero. Otherwise, it does not modify the list, returns -1, and sets **errno** to an appropriate value.

See Also

getgrent(), libc, setgroups()

Notes

If *user* belongs to more than **NGROUPS_MAX** groups, **initgroups()** reads only the first **NGROUPS_MAX** groups from **/etc/group** and ignores all of the others. Note that **NGROUPS_MAX** is a limit set by the POSIX Standard. For a fuller discussion of these limits, see the Lexicon entries for **sysconf()** and **limits.h**.

Only the superuser **root** can use **initgroups()**.

initialization — Definition

The term *initialization* refers to setting a variable to its first, or initial, value.

Rules of Initialization

Initializers follow the same rules for type and conversion as do assignment statements.

If a static object with a scalar type is not explicitly initialized, it is initialized to zero by default. Likewise, if a static pointer is not explicitly initialized, it is initialized to NULL by default. If an object with automatic storage duration is not explicitly initialized, its contents are indeterminate.

Initializers on static objects must be constant expressions; greater flexibility is allowed for initializers of automatic variables. These latter initializers can be arbitrary expressions, not just constant expressions. For example,

```
double dsin = sin(30.0);
```

is a valid initializer, where **dsin** is declared inside a function.

To initialize an object, use the assignment operator '='. The following sections describe how to initialize different classes of objects.

Scalars

To initialize a scalar object, assign it the value of a expression. The expression may be enclosed within braces; doing so does not affect the value of the assignment. For example, the expressions

```
int example = 7+12;
```

and

```
int example = { 7+12 };
```

are equivalent.

Unions and Structures

The initialization of a **union** by definition fills only its *first* member.

To initialize a **union**, use an expression that is enclosed within braces:

```
union example_u {
      int member1;
      long member2;
      float member3;
} = { 5 };
```

This initializes **member1** to five. That is to say, the **union** is filled with an **int**-sized object whose value is five.

To initialize a structure, use a list of constants or expressions that are enclosed within braces. For example:

```
struct example_s {
      int member1;
      long member2;
      union example_u member3;
};

struct example_s test1 = { 5, 3, 15 };
```

This initializes **member1** to five, initializes **member2** to three, and initializes the *first* member of **member3** to 15.

Strings

To initialize a string pointer,
use a string literal.

The following initializes a string:

```
char string[] = "This is a string";
```

The length of the character array is 17 characters: one for every character in the given string literal plus one for the null character that marks the end of the string.

If you wish, you can fix the length of a character array. In this case, the null character is appended to the end of the string only if there is room in the array. For example, the following

```
char string[16] = "This is a string";
```

writes the text into the array **string**, but does not include the concluding null character because there is not enough room for it.

A pointer to **char** can also be initialized when the pointer is declared. For example:

```
char *strptr = "This is a string";
```

initializes **strptr** to point to the first character in **This is a string**. This declaration automatically allocates exactly enough storage to hold the given string literal, plus the terminating null character.

Arrays

To initialize an array, use a list of expressions that is enclosed within braces. For example, the expression

```
int array[] = { 1, 2, 3 };
```

initializes **array**. Because **array** does not have a declared number of elements, the initialization fixes its number of elements at three. The elements of the array are initialized in the order in which the elements of the initialization list appear. For example, **array[0]** is initialized to one, **array[1]** to two, and **array[2]** to three.

If an array has a fixed length and the initialization list does not contain enough initializers to initialize every element, then the remaining elements are initialized in the default manner: static variables are initialized to zero, and other variables to whatever happens to be in memory. For example, the following:

```
int array[3] = { 1, 2 };
```

initializes **array[0]** to one, **array[1]** to two, and **array[2]** to zero.

The initialization of a multi-dimensional array is something of a science in itself. The ANSI Standard defines that the ranks in an array are filled from right to left. For example, consider the array:

```
int example[2][3][4];
```

This array contains two groups of three elements, each of which consists of four elements. Initialization of this array will proceed from **example[0][0][0]** through **example[0][0][3]**; then from **example[0][1][0]** through **example[0][1][3]**; and so on, until the array is filled.

It is easy to check initialization when there is one initializer for each "slot" in the array; e.g.,

```
int example[2][3] = {
      1,  2,  3,  4,  5,  6
};
```

or:

```
int example[2][3] = {
     {  1,  2,  3 }, {  4,  5,  6 }
};
```

The situation becomes more difficult when an array is only partially initialized; e.g.,

```
int example[2][3] = {
       { 1 }, { 2, 3 }
};
```

which is equivalent to:

```
int example[2][3] = {
     { 1, 0, 0 }, { 2, 3, 0 }
};
```

As can be seen, braces mark the end of initialization for a "cluster" of elements within an array. For example, the following:

```
int example[2][3][4] = {
     5, { 1, 2 }, { 5, 2, 4, 3 }, { 9, 9, 5 },
     { 2, 3, 7 } };
```

is equivalent to entering:

```
int example[2][3][4] = {
     { 5, 0, 0, 0 },
     { 1, 2, 0, 0 },
     { 5, 2, 4, 3 },

     { 9, 9, 5, 0 },
     { 2, 3, 7, 0 },
     { 0, 0, 0, 0 }
};
```

The braces end the initialization of one cluster of elements; the next cluster is then initialized. Any elements within a cluster that have not yet been initialized when the brace is read are initialized in the default manner.

See Also

array, C language, Programming COHERENT, struct, union
ANSI Standard, §3.5.7

ino.h — Header File

Constants and structures for disk i-nodes
#include <sys/inode.h>

inode.h declares structures and constants that are used to describe i-nodes.

See Also

i-node, header files

inode.h — Header File

Constants and structures for memory-resident i-nodes
#include <sys/inode.h>

inode.h declares structures and constants for memory-resident i-nodes.

See Also

header files, i-node

install — Command

Install a software update onto COHERENT
/etc/install [-c] *id device ndisks*

The command **install** installs an update of the COHERENT system onto your hard disk. *id* identifies the update to be installed. *device* is the device from which the update disks will be read. *ndisks* is the number of disks that comprise the update.

Option **-c** tells **install** to uncompress compressed files directly from the installation disks, rather than copy the compressed files onto disk and uncompress them there. **install** reads information about compression formats and options from file **/etc/install.u**. This switch permits software packages other than COHERENT to use compressed files.

Third-Party Software

install also provides a standard mechanism by which software developers can install their software onto systems that run COHERENT. The rest of this article discusses how to prepare a software release so that it can be installed using **install**.

For **install** to be able to install a software distribution, the distribution must consist of a set of mountable floppy disks, each holding a COHERENT file system created by **mkfs**. This keeps the disks independent of each other and also lets the user to insert the disks in any order. **install** records the fact that it has read a given disk from the distribution, thus preventing the user from attempting to read a given disk more than once during an installation session.

Floppy disks should be built using **mkfs**. Each disk in the distribution must hold in its root directory a file whose name is of the form:

/id.sequence

Here, *id* identifies the release, as described above. Note that *id* must be formed from the set of upper- and lower-case letters, digits, the period '.', and the underscore character '_', and not exceed nine characters in length. *sequence* indicates which disk in the distribution this disk is, from one through the total number of disks.

install uses the command **cpdir** to copy each of the distribution disks to directory **/** on the current system. Therefore, every disk should be "root based" (i.e., full path names should be used). Because **install** is run by the superuser, **cpdir** preserves the date and time for each file, and preserves ownership and modes. To keep file ownership consistent with COHERENT conventions, make files that are neither *setuid* nor *setgid* owned by user **bin** and group **bin**. **install** recreates on your hard disk all directories that it finds on the distribution disks, as needed. Be careful when choosing the ownership and mode of directories because you could inadvertently compromise the security of your users' systems.

Preprocessing

When you load a disk, **install** seeks a file named *id.disk*.**pre** upon that disk. If it finds such a file, **install** assumes that that file is a script, copies it into directory **/conf**, and executes it from there *before* it copies any files from the disk onto your system. If you are installing COHERENT, it uses the command:

id.disk.**pre** *id.disk*.**arch**

If you are updating a package rather than installing it, **install** uses the command:

id.disk.**pre -u** *id.disk*.**arch**

install always gives the same argument to the **.pre** script. As its suffix **.arch** indicates, the argument usually names a file whose contents name files that should be archived **install** copies the contents of the disk onto your system. **install** merely passes the name of the **.arch** file that *might* exist on the installation disk to the **.pre** script: it is up to the **.pre** script to check for the existence of the **.arch** file, read it, and perform the archiving. Of course, the **.pre** script can ignore this argument should it choose.

For example, if you are installing X Windows onto your COHERENT system, the identifier is **CohX**. When you load the first disk into your system, **install** looks for file **CohX.1.pre** on that disk. If it finds that file, **install** copies it into directory **/conf** on your root file system, and invokes it with the command:

```
/conf/CohX.1.pre CohX.arch
```

All of this occurs *before* **install** copies any files from that disk onto your system. In this way, files can be archived or otherwise backed up before they are overwritten by the package you are installing onto your system.

One last behavior should be noted: if **install** finds a **.pre** file on the *first* disk of the installation kit, it also seeks on that disk a file that has the suffix **.supp** on that disk. The suffix **.supp** stands for "suppression": normally, it names files that are *not* to be copied from the release onto your system. It is the flip side of the **.arch** file.

Note that you can mount the disks of a release and edit these **.arch** and **.supp** files *before* you install the package onto your system. In this way, you can protect your system from being damaged by installing new software onto your system.

Postprocessing

After all disks in a distribution have been successfully copied by the user, **install** checks for the existence of a file of the form

```
/conf/id.post
```

where *id* matches the *id* field found on the **install** command line. If found, **install** executes this file to allow special "postprocessing," such as installing manual pages into directory **/usr/man** or executing installation-specific commands.

Before an installation procedure completes its postprocessing, it should remove any *id* files of the following form from the target system:

/conf/*id*.post
/*id.sequence*

Adding Manual-Page Entries

As part of building a distribution, you usually must generate pre-processed or "cooked" manual-page entries for distribution with your upgrade or add-on package. These should be inserted into the subdirectories of **/usr/man**, with the name of the subdirectory being specific to your product. This naming convention avoids name-space collisions, should multiple applications use the same name for a manual-page entry.

If you install new or additional manual pages, you must update the index file used by the **man** command to locate manual entries. File **/usr/man/man.index** on the target file system contains index entries for all manual pages on the system. As part of postprocessing, you must append index information for your manual pages to the end of the existing index file. In addition, file **/usr/man/man.help** contains the **man** command's help message. This includes a list of valid topics and some explanatory text. You should also append to this file a brief list of the manual page entries that you have added. For further information on manual pages, see the Lexicon entry for the command **man**.

Logging

install logs all partial as well as completed installations in file **/etc/install.log**. This information includes date/time stamps and the command-line arguments to **install**.

Example

The following installs COHERENT update **coh.301**, which consists of one disk, from a high-density 5.25-inch floppy drive:

```
/etc/install coh.301 /dev/fha0 1
```

Files

/etc/install.log

See Also

commands, man, mkfs

int — C Keyword

Data type

An **int** is the most commonly used numeric data type, and is normally used to encode integers. Under COHERENT 386, an **int** is the same size as a **long**; **sizeof int** equals 4 (31 bits plus a sign bit), and can hold any value from -2,147,483,647 to 2,147,483,647. Under COHERENT 286, an **int** is the same size as a **short**; that is, **sizeof int**

equals 2 (15 bits plus a sign bit), and can hold any value from from -32768 to +32767.

An **int** normally is sign extended when cast to a larger data type; an **unsigned int**, however, will be zero extended.

See Also

C keywords, data formats, data types, long, short
ANSI Standard, §6.1.2.5

interrupt — Definition

An **interrupt** is an interruption of the sequential flow of a program. It can be generated by the hardware, from within the program itself, or from the operating system.

See Also

Programming COHERENT, signal()

io.h — Header File

Constants and structures used by I/O
#include <sys/io.h>

io.h declares constants and structures used by various I/O routines.

See Also

header files

ioctl() — System Call (libc)

Device-dependent control
#include <unistd.h>
#include <*header***.h>**
ioctl(*fd***,** *command***,** *arg***)**
int *fd***,** *command***; char** **arg***;*

ioctl() lets you interact directly with a device driver. You can use it to set or retrieve parameters for devices (line printers, communications lines, terminals), and non-standard spacing operations for tape drives.

ioctl() acts upon the block-special file or character-special file associated with the file descriptor *fd*. *command* points to the specific request.

header names the header file that defines symbolic commands for the device you wish to manipulate. Using the symbolic command definitions from the header files promotes device independence within each device type. A complete list of symbolic commands appears below.

arg passes a buffer of information (defined by structures in the appropriate header files) to the driver. For any *command* not needing additional information, this argument should be NULL.

Some **ioctl()** requests work on all files, and are not passed to any driver.

ioctl() returns -1 on errors, such as a bad file descriptor. Because the call is device dependent, almost any other error could be returned.

Commands

The following gives the commands that can be used with **ioctl()**, as extracted from COHERENT's header files. Please note the following caveats:

- New drivers are being added continually to COHERENT, both by Mark Williams Company and by users and third-party vendors. You should regard the following list as being tentative at best.
- Because the commands and arguments with with **ioctl()** are unique to COHERENT's suite of device drivers, **ioctl()** is one of the least portable of all system calls. If you want your code to run on multiple operating systems, you should use **ioctl()** judiciously.

<sys/cdrom.h>
Header file used to manipulate a CD-ROM device. Unless otherwise noted, *arg* is ignored:

CDROMPAUSE Pause playing an audio CD.

CDROMRESUME	Resume playing an audio CD.
CDROMPLAYMSF	Play an audio CD at a given minute-second frame (MSF) address. *arg* points to an array of six bytes that give the MSF address.
CDROMPLAYTRKIND	Play a track on an audio CD. *arg* points to an array of four bytes that give, respectively, the start track, the start index, the end track, and the end index of the track to be played.
CDROMREADTOCHDR	Read the CD's table-of-contents header. *arg* points to a structure of type **cdrom_tochdr** into which the header is written.
CDROMREADTOCENTRY	Read an entry from the table-of-contents header. *arg* points to a structure of type **cdrom_tocentry** into which the entry is written.
CDROMSTOP	Spin down the CD-ROM drive's motor.
CDROMSTART	Turn on the CD-ROM drive's motor.
CDROMEJECT	Eject the CD-ROM. Note that this does not work on every variety of CD-ROM drive.
CDROMVOLCTRL	Control the volume on an audio CD. *arg* points to an array of four bytes that, respectively, set the the volume on channels zero through three.
CDROMSUBCHNL	Read data about a sub-channel. *arg* points to a structure of type **cdrom_subchnl** into which the information about the sub-channel is written.
CDROMREADMODE1	Read type-1 data. *arg* points to a structure into which the data are written.
CDROMREADMODE2	Read type-2 data. *arg* points to a structure into which the data are written.

<sys/fdioctl.h>

This header file is used with the floppy-disk drive:

FDFORMAT	Format a track on a floppy disk. *arg* points to a two-byte array that identifies, respectively, the cylinder and head to format.

<sys/hdioctl.h>

This header file is used with AT-style hard-disk drives (i.e., IDE, ESDI, MFM, or RLL disks). *arg* gives the address in user memory where drive attributes reside, or to which they should be written:

HDGETA	Get drive attributes.
HDSETA	Set drive attributes.
HDGETIDEINFO	Get the attributes of an IDE drive. *arg* should point to a copy of the structure **ide_info**; this call to **ioctl()** initializes the structure with the requested information.

<sys/null.h>

This header file defines **ioctl**s that examine system memory:

NLFREE	Read the amounts of memory on your system that are available and free. *arg* gives the address of an object of type **FREEMEM**, which is defined in header file **<null.h>**. This type is an array of two **long**s: the first receives the amount of available memory, and the second the amount of free memory. For an example of a program that uses this **ioctl()**, see the Lexicon entry for **freemem**.
NLIDLE	Read the system's idle time. *arg* points to an array of two **long**s. The first **long** receives system's idle ticks; the second, the number of ticks since system startup. From reading these values repeatedly, you can compute the changes in system idle time and time since startup, and so see what the system's load is. For an example of how to this call to **ioctl()**, see the Lexicon entry for **idle**.

<sys/sdioctl.h>

The commands defined in this header file are passed to the driver **aha**, which manipulates Adaptec SCSI disks. None does anything.

<sgtty.h>

The following commands are used with the **sgtty** method of controlling terminal devices. They are documented in more detail in the Lexicon entry for **sgtty**. *arg* points to a structure of type **sgttyb**, which is defined in that header file:

TIOCHPCL	Hang up on last close.
TIOCGETP	Get modes (old **gtty**).
TIOCSETP	Set modes (old **stty**).
TIOCSETN	Set modes without delay or flush.

TIOCEXCL	Set exclusive use.
TIOCNXCL	Set non-exclusive use.
TIOCFLUSH	Flush I/O queues.
TIOCSETC	Set characters.
TIOCGETC	Get characters.

<stropts.h>

STREAMS commands. *arg* points to a STREAMS control block that will be used to generate an **M_IOCTL** message.

I_NREAD	Get message length, count.
I_PUSH	Push named module.
I_POP	Pop topmost module.
I_LOOK	Get name of the topmost module.
I_FLUSH	Flush read/write side.
I_SRDOPT	Set stream head read mode.
I_GRDOPT	Get stream head read mode.
I_STR	Send **ioctl()** message downstream.
I_SETSIG	Register for signal **SIGPOLL**.
I_GETSIG	Return registered event mask.
I_FIND	Locate named module on stream.
I_LINK	Link two streams.
I_UNLINK	Unlink two streams.
I_RECVFD	Receive file descriptor from pipe.
I_PEEK	Examine stream head data.
I_SENDFD	Send file descriptor to pipe.

The following commands are not covered by iBCS2:

I_SWROPT	Set stream write mode.
I_GWROPT	Get stream write mode.
I_LIST	Get name of all modules/drivers.
I_PLINK	Create persistent link.
I_PUNLINK	Undo persistent link.
I_FLUSHBAND	Flush priority band.
I_CKBAND	Check for existence of priority band.
I_GETBAND	Get band of first message.
I_ATMARK	Check whether current message is marked.
I_SETCLTIME	Set drain timeout for stream.
I_GETCLTIME	Get the current close timeout.
I_CANPUT	Check if band is writeable.

<sys/tape.h>

Header file for interfacing with magnetic-tape devices. *arg* points to an area in user space that holds additional information for the tape device. A tape driver may recognize any of the following **ioctl()** commands:

T_ERASE	Erase tape.
T_LOAD	Load. Not used.
T_RDSTAT	Read status.
T_RST	Reset.
T_RETENSION	Retension tape.
T_RWD	Rewind tape.
T_SBB	Space block backward — move backward by *arg* blocks. Not used.
T_SBF	Space Block Forward — move forward by *arg* blocks. Not used.
T_SBREC	Not used.
T_SFB	Space Filemark Backward — move backwards by *arg* files.
T_SFF	Space Filemark Forward — move forward by *arg* files.
T_SFREC	Not used.
T_TINIT	Not used.
T_UNLOAD	Unload. Not used.
T_WRFILEM	Write file marks. Not used.

<termio.h>

The following commands are used with the **termio** method of controlling a terminal. They are documented in more detail in the Lexicon entry for **termio**. *arg* points to a structure of type **sgttyb**, which is described above.

TCGETA	Get terminal parameters.
TCSETA	Set terminal parameters.
TCSETAW	Wait for drain, set parameters.
TCSETAF	Wait for drain, flush input, set parms.
TCSBRK	Send 0.25-second break.

The following commands also take arguments when called via **ioctl()**:

TCXONC	Start/stop control: An argument of zero suspends output; an argument of one restarts suspended output.
TCFLSH	Flush queues: An argument of zero flushes the input queue; an argument of one flushes the output queue; and an argument of two flushes both queues.

<sys/vtkd.h>

This header file defines commands used with the keyboard driver. *arg* points to a structure of type **sgttyb**, which is defined in header file **sgtty.h**.

KDMAPDISP	Map the display into user space.
KDSKBMODE	Toggle the scan code **xlate**.
KDMEMDISP	Dump a byte of virtual or physical memory.
KDGKBSTATE	Get the keyboard's shift state.
KIOCINFO	Determine the workstation of the virtual terminal.
KIOCSOUND	Start sound generation.
KDGETLED	Get the state of the keyboard's LEDs.
KDSETLED	Set the state of the LEDs.

The following four **ioctl()** commands allow user programs to perform I/O instructions directly, rather than going through the system-call interface and having the kernel perform the I/O. The most common need for these functions is in window managers and similar applications, where the usual kernel interface would be unacceptably slow.

Normally, any user program that attempts to execute I/O instructions directly to hardware will get an immediate **SIGSEGV** and be terminated. Use of the commands below allow user-level programs to perform I/O without being terminated. The I/O operations are available through functions **inb()**, **outb()**, etc., which are present in the kernel-support library **/etc/conf/lib/k386.a** and are documented in the manual to the COHERENT Device Driver Kit.

Access to any of these functions may be restricted to the superuser on some systems:

KDENABIO	Allow the user process permission to perform input/output operations to all available I/O addresses. The third argument to **ioctl()** is ignored.
KDDISABIO	Prohibit user processes from performing input/output operations to all available I/O addresses. The third argument to **ioctl()** is ignored. It is normal for direct I/O to ports to be disallowed at user level. The main reason for this call is to undo the effect of preceding **KDENABIO** or **KDADDIO** calls.
KDADDIO	Allow user-level I/O to a port. The third argument to **ioctl()** is an **unsigned short** that gives the single address value of the port.
KDDELIO	Disallow user-level I/O to a port. The third argument to **ioctl()** is an **unsigned short** that gives the single address value of the port. It is normal for direct I/O to ports to be disallowed at user level. The main reason for this call is to undo the effect of preceding **KDADDIO** calls.

Example

The following program, by Udo Munk, demonstrates how to use **ioctl()** to read a mouse plugged into a serial port. It takes one argument, the name of the port you wish to check.

```
#include <fcntl.h>
#include <poll.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
#include <termio.h>

char *mouse;
int mouse_fd;
struct termio old_tty, new_tty;

/* do the right thing by signals */
sig_handler()
{
      ioctl(mouse_fd, TCSETAF, &old_tty);
      exit(EXIT_SUCCESS);
}

/* cry and die */
void fatal(message)
char *message;
{
      fprintf (stderr, "%s\n", message);
      exit(EXIT_FAILURE);
}

/* run the whole shebang */
main(argc, argv)
int argc; char **argv;
{
      struct pollfd fds[1];

      if (argc != 2)
            fatal ("Usage: findmouse /dev/com[1-4]pl");

      if (strncmp(argv[1], "/dev/com1pl", 11) &&
                  strncmp(argv[1], "/dev/com2pl", 11) &&
                  strncmp(argv[1], "/dev/com3pl", 11) &&
                  strncmp(argv[1], "/dev/com4pl", 11))
            fatal ("Usage: findmouse /dev/com[1-4]pl");

      mouse = argv[1];

      signal(SIGINT, sig_handler);
      signal(SIGQUIT, sig_handler);
      signal(SIGHUP, sig_handler);

      fprintf(stdout, "Trying to open %s ...\n", mouse);
      if ((mouse_fd = open(mouse, O_RDONLY)) < 0)
            fatal ("Cannot open this device.");
      fprintf(stdout, "Success.\n");

      fprintf(stdout, "Trying to read line mode of %s ...\n", mouse);
      if (ioctl(mouse_fd, TCGETA, &old_tty) < 0)
            fatal ("Cannot read this device’ss line mode.");
      fprintf(stdout, "Success.\n");

      new_tty = old_tty;
      new_tty.c_cflag &= ~(CBAUD | HUPCL);
      new_tty.c_cflag |= CLOCAL | B1200;
      new_tty.c_iflag = IGNBRK;
      new_tty.c_oflag = new_tty.c_lflag = 0;

      /*
       * VMIN = 0, VTIME = 0 has the same effect as setting O_NDELAY on the
       * input line.
       */
      new_tty.c_cc[VMIN] = 0;
      new_tty.c_cc[VTIME] = 0;
```

```
      /* Set up to poll the input line. */
      fds->fd = mouse_fd;
      fds->events = POLLIN;

      fprintf(stdout, "Trying to set new line mode for %s ...\n", mouse);
      if (ioctl(mouse_fd, TCSETAF, &new_tty) < 0)
            fatal ("Cannot set new tty line mode");
      fprintf(stdout, "Success.\n");

      fprintf(stdout, "\nI'm reading from %s. To exit, type <ctrl-C>.\n",
            mouse);
      fprintf(stdout,
            "If you see stuff on the screen when you move the mouse,\n");
      fprintf(stdout,
            "then you have found the mouse port.\n");
      fprintf(stdout, "\nNow wiggle your mouse:\n");

      for (;;) {
            size_t read_count;
            unsigned char mousebuf [128];

            /* Block waiting for mouse input. */
            if (poll (fds, 1, -1) < 0)
                  break;

            /* Drain input in large chunks until it becomes time to block. */
            while ((read_count = read (mouse_fd, mousebuf,
                        sizeof (mousebuf))) != 0) {
                  unsigned char * scan = mousebuf;

                  do
                        printf ("%02x ", * scan ++);
                  while (-- read_count != 0);

                  fflush (stdout);
            }
      }
}
```

See Also

device drivers, exec, getty, header files, libc, open(), read(), sgtty, stty(), termio

Notes

The type of the *arg* to **ioctl()** is declared as **char** * mainly to improve portability. In most cases, the actual argument type will be something like **struct sgttyb** *, depending on the device and command. The actual argument should be cast to type **char** * to ensure cross-machine portability.

Under COHERENT 286, the main header file for **ioctl()** is **<sgtty.h>**. This header file is also included with COHERENT 386 for compatibility with older applications.

ipc.h — Header File

Definitions for interprocess communications
#include <sys/ipc.h>

ipc.h defines constants and structures used by functions that perform interprocess communications.

See Also

header files, msgget(), semget(), shmget()

ipcrm — Command

Remove an interprocess-communication memory item
ipcrm [-mqs *id***] [-MQS** *key***]**

The command **ipcrm** removes a memory item used for interprocess communication: either shared-memory segment, message queue, or semaphore set. You can use this command either with an *id*, which is the identifying number assigned by the function that created the memory item in question; or with a *key*, which is the identifier used by the application that requested the memory item.

ipcrm recognizes the following options:

m *id* Remove the shared-memory segment with an identifier of *id*.

q *id* Remove the message queue with an identifier of *id*.

s *id* Remove the semaphore set with an identifier of *id*.

M *key* Remove the shared-memory segment with a key of *key*.

Q *key* Remove the message queue with a key of *key*.

S *key* Remove the semaphore set with a key of *key*.

To find the identifiers and keys for for the IPC resources that are currently allocated, use the command **ipcs**.

See Also

commands, ipcs, msgget(), semget(), shmget()

Notes

ipcrm does not remove a shared-memory segment until all processes attached to it are removed by calls to the function **shmat()**.

Any user can run **ipcrm**; however, a user can remove only those memory items that he "owns", as noted in the control structure for the item. The superuser **root** can, of course, remove all memory items, no questions asked.

ipcs — Command

Display a snapshot of interprocess communications
ipcs [-abcmopst] [-N *kernel***]**

The command **ipcs** prints information about interprocess communication (IPC) objects.

Options

ipcs recognizes the following command-line options:

-a "All" print option; i.e., combine the options **-b**, **-c**, **-o**, **-p**, and **-t**.

-b "Biggest" option: Display the maximum size that the kernel allows for each kind of IPC object.

-c Display the login name and group name of the user who created each IPC object.

-m Display information about shared-memory segments. By default, **ipcs** displays information about all IPC objects.

-N *kernel*
Read kernel-specific information from *kernel* instead of from the default kernel **/autoboot**.

-o "Outstanding usage" option: Display the number of messages currently queued and their total size in bytes, and the number of processes attached to each shared-memory segment.

-p Display the process identifiers of the following:

- The last process to send a message.
- The last process to receive a message on each message queue.
- Each creating process.
- The last process to attach to or detach from each shared-memory segment.

-q Display information about message queues. By default, **ipcs** displays information about all IPC objects.

-s Display information about sets of semaphores. By default, **ipcs** displays information about all IPC objects.

-t Display the following information about times:

- When functions **msgsnd()** and **msgrcv()** were last executed for each message queue.
- When the functions **shmat()** and **shmdt()** were last executed for each shared-memory segment.
- When the function **semop()** was last executed for each set of semaphores.

Format of Displayed Information

The following names and describes each column of information that **ipcs** can display for each IPC object. The letters in parentheses name the command-line options tell **ipcs** to display this column; **all** means that **ipcs** always displays this column:

ATIME (**-at**)
The last time a process attached itself to this shared-memory segment.

CBYTES (**-ao**)
The total number of bytes in this message queue.

CGROUP (**-ac**)
The name of the group to which the creator of this IPC object belongs.

CPID (**-ap**)
The identifier of the process that created this shared-memory segment.

CREATOR (**-ac**)
The login identifier of the user who created this IPC object.

CTIME (**-at**)
The time when this IPC object was created or last changed.

DTIME (**-at**)
The most recent time a process detached itself from this shared-memory segment.

GROUP (**all**)
The name of the group to which the owner of this IPC object belongs.

ID (**all**) The numeric identifier of this IPC object.

KEY (**all**)
The key that names this IPC object. Applications use this key to identify and manipulate the IPC object.

LPID (**-ap**)
The identifier of the last process to have attached itself to or detached itself from this shared-memory segment.

LRPID (**-ap**)
The identifier of the last process to have received a message from this message queue.

LSPID (**-ap**)
The identifier of the last process to have sent a message to this message queue.

MODE (**all**)
The IPC object's mode. The mode is a string of 11 characters that interprets the value of field **mode** in the structure **ipc_perm**, which is part of each IPC object. (For more information on this structure, see the Lexicon entries **msgget()**, **semget()**, and **shmget()**.) The first two mode characters are as follows:

R A process is waiting for **msgrcv()**.
S A process is waiting for **msgsnd()**.
D The associated shared-memory segment has been removed.
C The associated shared-memory segment will be cleared when the first process attaches itself to it.
- The corresponding flag is not set.

The last nine characters of the mode give the permissions on the IPC object — three sets of three characters each. In each set, the first character marks whether read permission is granted, the second whether permission to write or alter is granted, and the third is unused. The first set gives the permissions of the user who created the object (its "owner"); the second, the permissions of other users in the owner's group; and the third, the permissions of all other users.

NATTCH (**-ao**)
The number of processes attached to this shared-memory segment.

NSEMS (**-ab**)
The number of semaphores in this set.

OTIME (..... t)
The last time a semaphore operation was completed on this set.

OWNER (**all**)
The login identifier of the user who "owns" this IPC object.

QBYTES (**-ab**)
The number of bytes left available to the messages in this queue.

QNUM (**-ao**)
The number of messages in this queue.

RTIME (**-at**)
The last time a message was received from this queue.

SEGSZ (**-ab**)
The size of this shared-memory segment.

STIME (**-at**)
The last time a message was sent to this queue.

T (**all**) The type of IPC object this is, as follows:

m	Shared-memory segment
q	Message queue
s	Set of semaphores

See Also

commands, ipcrm, msgget(), semget(), shmget()

Notes

ipcs gives information about the way interprocess communications are at the moment you run it. The data it returns can change even as you read them.

IRQ — Technical Information

Interrupts on the IBM PC

The term *IRQ* stands for "interrupt request". The IBM PC has 16 interrupts channels built into it. Some are reserved for system hardware; most are available for cards and peripheral devices. The following gives the default assignments for IRQs:

IRQ	*Device*
0	System timer
1	Keyboard controller
2	Second IRQ controller
3	Serial port (COM) 2
4	COM1
5	Line printer (LPT) 2 or LPT3
6	Floppy-disk controller
7	LPT1
8	Real-time clock
9	Re-directed IRQ2
10	Available
11	Available
12	Motherboard mouse port (available if no mouse)
13	Mathematics coprocessor
14	Hard-disk (AT) controller
15	Available

As you can see, there are two banks of interrupt controllers, each of which controls eight interrupts, with IRQ2 latched to the first port on the second chip, IRQ9.

Channel 5 handles two parallel ports — LPT2 and LPT3. If you install three serial ports onto your system, be careful on how you jumper the card, or you will confuse your system.

Due to the design of the PC, IRQ 7 can display spurious interrupts when a device signals an IRQ line other than 7, then cancels the signal before the interrupt controller figures out which line the IRQ occurred on. Thus, you

should not assign other devices to IRQ 7, if at all possible.

Three interrupt channels are available for user hardware: channels 10, 11, and 15. Channel 12 is also available if you do not have a bus mouse.

Only two interrupts are available for serial ports, COM1 and COM2. Note that COM3 and COM4 are "linked" to COM1 and COM2, respectively. For this reason, if you have both COM1 and COM3 your system, or both COM2 and COM4, only one of the pair can be interrupt driven; the other port of the pair must be polled.

See Also

Administering COHERENT

isalnum() — ctype Function (libc)

Check if a character is a number or letter
#include <ctype.h>
int isalnum(*c***) int** *c***;**

isalnum() tests whether the argument *c* is alphanumeric (**0-9**, **A-Z**, or **a-z**). It returns a number other than zero if *c* is of the desired type, and zero if it is not. **isalnum()** assumes that *c* is an ASCII character or EOF.

Example

For an example of how to use this macro, see the entry for **ctype.h**.

See Also

ASCII, libc
ANSI Standard, §7.3.1.1
POSIX Standard, §8.1

isalpha() — ctype Function (libc)

Check if a character is a letter
#include <ctype.h>
int isalpha(*c***) int** *c***;**

isalpha() tests whether the argument *c* is a letter (**A-Z** or **a-z**). It returns a number other than zero if *c* is an alphabetic character, and zero if it is not. **isalpha()** assumes that *c* is an ASCII character or EOF.

Example

For an example of this macro, see the entry for **ctype.h**.

See Also

ASCII, libc
ANSI Standard, §7.3.1.2
POSIX Standard, §8.1

isascii() — ctype Function (libc)

Check if a character is an ASCII character
#include <ctype.h>
int isascii(*c***) int** *c***;**

isascii() tests whether *c* is an ASCII character (0 <= *c* <= 0177). It returns a number other than zero if *c* is an ASCII character, and zero if it is not. Many **ctype** macros fail if passed a non-ASCII value other than EOF.

Example

For an example of how to use this function, see the entry for **ctype.h**.

See Also

ASCII, libc

Notes

Please note that **isascii()** is not part of the ANSI standard. Programs that use it may not be portable to all implementations of C.

isatty() — General Function (libc) (libc)

Check if a device is a terminal
#include <unistd.h>
int isatty(*fd***) int** *fd***;**

isatty() checks to see if a device is a terminal. It returns one if the file descriptor *fd* describes a terminal, and zero otherwise.

Files

/dev/* — Terminal special files
/etc/ttys — Login terminals

See Also

ioctl(), libc, tty, ttyname(), ttyslot(), unistd.h
POSIX Standard, §4.7.2

iscntrl() — ctype Function (libc)

Check if a character is a control character
#include <ctype.h>
int iscntrl(*c***) int** *c***;**

iscntrl() tests whether the argument *c* is a control character (including a newline character) or a delete character. It returns a number other than zero if *c* is a control character, and zero if it is not. **iscntrl()** assumes that *c* is an ASCII character or **EOF**.

Example

For an example of how to use this macro, see the entry for **ctype.h**.

See Also

libc
ANSI Standard, §7.3.1.3
POSIX Standard, §8.1

isdigit() — ctype Function (libc)

Check if a character is a numeral
#include <ctype.h>
int isdigit(*c***) int** *c***;**

isdigit() tests whether the argument *c* is a numeral (**0-9**). It returns a number other than zero if *c* is a numeral, and zero if it is not. **isdigit()** assumes that *c* is an ASCII character or **EOF**.

Example

For an example of how to use this macro, see the entry for **ctype.h**.

See Also

ASCII, libc
ANSI Standard, §7.3.1.4
POSIX Standard, §8.1

isgraph() — ctype Function (libc)

Check if a character is printable
#include <ctype.h>
int isgraph(int *c***);**

isgraph() tests whether *c* is a printable letter within the ASCII character set, but excluding the space character. The ANSI Standard defines a printable character as any character that occupies one printing position on an output device. *c* must be a value that is representable as an **unsigned char** or **EOF**.

isgraph() returns nonzero if *c* is a printable character (except for space), and zero if it is not.

See Also

ASCII, libc
ANSI Standard, 7.3.1.5
POSIX Standard, §8.1

islower() — ctype Function (libc)

Check if a character is a lower-case letter
#include <ctype.h>
int islower(*c***) int** *c***;**

islower() tests whether the argument *c* is a lower-case letter (**a-z**). It returns a number other than zero if *c* is is a lower-case letter, and zero if it is not. **islower()** assumes that *c* is an ASCII character or **EOF**.

Example

For an example of how to use this macro, see the entry for **ctype.h**.

See Also

ASCII, libc
ANSI Standard, §7.3.1.6
POSIX Standard, §8.1

ispos() — Multiple-Precision Mathematics (libmp)

Return if variable is positive or negative
#include <mprec.h>
int ispos(*a***)**
mint **a***;*

ispos() returns true (nonzero) if *a* is not negative, false (zero) if *a* is negative.

See Also

libmp

isprint() — ctype Function (libc)

Check if a character is printable
#include <ctype.h>
int isprint(*c***) int** *c***;**

isprint() is a macro that tests if *c* is printable, i.e, if it is neither a delete nor a control character. It returns a number other than zero if *c* is a printable character, and zero if it is not. **isprint()** assumes that *c* is an ASCII character or **EOF**.

Example

For an example of how to use this macro, see the entry for **ctype.h**.

See Also

ASCII, libc
ANSI Standard, §7.3.1.7
POSIX Standard, §8.1

ispunct() — ctype Function (libc)

Check if a character is a punctuation mark
#include <ctype.h>
int ispunct(*c***) int** *c***;**

ispunct() tests whether the argument *c* is a punctuation mark, i.e., neither an alphanumeric character nor a control character. It returns a number other than zero if the character tested is a punctuation mark, and zero if it is not. **ispunct()** assumes that *c* is an ASCII character or **EOF**.

Example

For an example of how to use this macro, see the entry for **ctype.h**.

See Also

ASCII, libc
ANSI Standard, §7.3.1.8
POSIX Standard, §8.1

isspace() — ctype Function (libc)

Check if a character prints white space
#include <ctype.h>
int isspace(*c***) int** *c***;**

isspace() tests whether the argument *c* is a space, tab, newline, carriage return, or form-feed character. It returns a number other than zero if *c* is a white-space character, and zero if it is not. **isspace()** assumes that *c* is an ASCII character or EOF.

Example

For an example of how to use this macro, see the entry for **ctype.h**.

See Also

ASCII, libc
ANSI Standard, §7.3.1.9
POSIX Standard, §8.1

isupper() — ctype Function (libc)

Check if a character is an upper-case letter
#include <ctype.h>
int isupper(*c***) int** *c***;**

isupper() tests whether the argument *c* is an upper-case letter (**A-Z**). It returns a number other than zero if *c* is an upper-case letter, and zero if it is not. **isupper()** assumes that *c* is an ASCII character or **EOF**.

Example

For an example of how to use this macro, see the entry for **ctype.h**.

See Also

ASCII, libc
ANSI Standard, §7.3.1.10
POSIX Standard, §8.1

isxdigit() — ctype Function (libc)

Check if a character is a hexadecimal numeral
#include <ctype.h>
int isxdigit(*c***)**
int *c***;**

isxdigit() tests whether *c* is a hexadecimal numeral — that is, any of the characters '0' through '9', any of the letters 'a' through 'd', or any of the letters 'A' through 'D'. *c* must be a value that is representable as an **unsigned char** or **EOF**.

isxdigit() returns nonzero if *c* is a hexadecimal numeral, and zero if it is not.

See Also

ASCII, libc
ANSI Standard, §7.3.1.11
POSIX Standard, §8.1

itom() — Multiple-Precision Mathematics (libmp)

Create a multiple-precision integer

#include <mprec.h>
mint *itom(*n***)**
int *n***;**

itom() creates a new multiple-precision integer (or **mint**), initializes it to the signed integer value *n*, and returns a pointer to it. You can use the function **mintfr()** to reclaim the storage used by the **mint** created by **itom()**.

See Also

libmp

j0() — Mathematics Function (libm)

Compute Bessel function
#include <math.h>
double j0(*z***)**
double *z***;**

j0() computes the Bessel function of the first kind for order 0 for its argument *z*.

Example

This example, called **bessel.c**, demonstrates the Bessel functions **j0()**, **j1()**, and **jn()**. Compile it with the following command line

```
cc -f bessel.c -lm
```

to include floating-point functions and the mathematics library.

```
#include <errno.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define display(x) dodisplay((double)(x), #x)

dodisplay(value, name)
double value; char *name;
{
      if (errno)
            perror(name);

      else
            printf("%10g %s\n", value, name);
      errno = 0;
}

main()
{
      extern char *gets();
      double x;
      char string[64];

      for(;;) {
            printf("Enter number: ");
            if(gets(string) == NULL)
                  break;
            x = atof(string);

            display(x);
            display(j0(x));
            display(j1(x));
            display(jn(0,x));

            display(jn(1,x));
            display(jn(2,x));
            display(jn(3,x));
      }
}
```

See Also

j1(), jn(), libm

j1() — Mathematics Function (libm)

Compute Bessel function
#include <math.h>
double j1(*z***) double** *z***;**

j1() takes *z* and computes the Bessel function of the first kind for order 1.

Example

For an example of this function, see the entry for **j0()**.

See Also

j0(), jn(), libm

jn() — Mathematics Function (libm)

Compute Bessel function
#include <math.h>
double jn(*n***,** *z***) int** *n***; double** *z***;**

jn() takes *z* and computes the Bessel function of the first kind for order *n*.

Example

For an example of this function, see the entry for **j0()**.

See Also

j0(), j1(), libm

jobs — Command

Print information about jobs
jobs

The command **jobs** is used with the Korn shell's job-control feature. It prints information about all background jobs. The information printed is in the following format:

%*num [+-] pid status command*

num indicates the job number, **+** indicates that the job is the "current job"; **-** indicates that it is the "previous job". *pid* gives the process identifier of the job. *status* indicates the status of the job. *command* gives the job's command line.

For details about job control, see the Lexicon entry for **ksh**.

See Also

commands, ksh

join — Command

Join two data bases
join [-a [*n***]] [-e** *string* **] [-j[***n***]** *keyf***] [-o** *n.m ...***] [-t***c***]** *file1 file2*

join processes the text files *file1* and *file2*, each of which contains a relational data base. If either file name is '-', the standard input is used for that file.

For the purposes of **join**, a data base file contains a set of records, one per input line. Each record contains a number of *fields.* One field is differentiated as *key* field for each file. Each file must be sorted by key field, for example with **sort**.

By default, the key field is the first field in each record. The **-j** option changes the key field number to *keyf* for the desired file. In this and other options below, the optional file number *n* must be **1** to indicate *file1* or **2** to indicate *file2*. If no *n* is given, both *file1* and *file2* are assumed.

Normally, fields are separated by any amount of white space (blanks or tabs). Leading blanks or tabs are not considered part of the fields. With the **-t** option, the separator character is *c*. With this option zero-length fields are possible; every occurrence of the separator ends the previous field and starts a new one.

Output consists only of records for which the key field occurs in both files. As a consequence of the sorted order of the input, the output is also sorted by the key field. Each output record has first the key field, then each field from the *file1* record but the key field, and then each field from the *file2* record but the key field. Fields are separated in the output with the specified field character, or with a space character if no **-t** option was given. Output records are always terminated with a newline. Under the **-e** option, *string* is printed for each empty field.

The **-a** option enables printing of records found in only file *n*. If *n* is missing, unpaired records are printed from both input files. To output only certain fields, the **-o** option precedes a list of desired fields to print. Each element is of the form *n.m* where *n* is the file number and *m* is the field number.

For example,

```
join -t: -j1 3 -o 1.3 2.4 1.4 1.1 2.2 filea fileb
```

joins **filea** and **fileb** which have fields separated by the colon (':') character. The join field number is 3 for **filea** and 1 (by default) for **fileb.** The selected five fields are produced in the output.

See Also

awk, comm, commands, sort, uniq

jrand48() — Random-Number Function (libc)

Return a 48-bit pseudo-random number as a long integer
long jrand48(*xsubi***)**
unsigned short *xsubi***[3];**

Function **jrand48()** generates a 48-bit pseudo-random number, and returns its upper 32 bits in the form of a **long**. The value returned is (or should be) uniformly distributed throughout the range of -2^31 through 2^31. *xsubi* is an array of three unsigned short integers from which the pseudo-random number is built.

See Also

libc, srand48()

kb.h — Header File

Define keys for loadable keyboard driver
#include <sys/kb.h>

The header file **kb.h** defines macros and manifest constants that are used with **nkb**, the user-configurable keyboard driver. It is **include**d with the C programs that the user can modify and compile to remap her keyboard. See the Lexicon entries **nkb** and **keyboard tables** for more information.

nkb is also used with COHERENT system of virtual consoles. **kb.h** sets default definitions for function keys, as follows:

vt0 — **vt15**
Switch to logical session 0 through 15, respectively.

color0 — **color15**
Switch to color session 0 through 15, respectively.

mono0 — **mono15**
Switch to monochrome session 0 through 15, respectively.

vtn Switch to next higher-numbered open session.

vtp Switch to next lower-numbered open session.

vtt Toggle to most recently used open session

See Also

header files, virtual console, vtkb, vtnkb

kernel — Technical Information

Master program of the COHERENT system

The *kernel* is the master program of the COHERENT system. It manages the file systems, processes, devices, and users.

When you boot COHERENT on your system, the COHERENT bootstrap automatically loads and runs the program **/autoboot**. This file usually is linked to the kernel that you build when you installed COHERENT onto your computer.

Your system may have multiple kernels on it. For example, when you update COHERENT, often the old kernel is saved; and you can also build customized versions of the kernel. The COHERENT bootstrap lets you boot other versions of the kernel besides the one that is linked to file **/autoboot**. For details on how to do this, see the Lexicon article **booting**.

For information on the file system that the kernel supports, see the Lexicon entry for **file**.

The Lexicon entry **coff.h** describes the format of programs that the kernel can execute.

The COHERENT system comes with a set of system calls, which you can call from within user application to obtain kernel services. See the Lexicon entry **libc** summarizes the calls that the kernel offers.

The function **ulimit()** returns and sets some limits for the current process. For details, see its entry in the Lexicon.

The Lexicon article **device drivers** describes the suite of drivers that come with the COHERENT system. It gives the major and minor numbers of each, plus information on how to access and manipulate each driver.

Modifying the Kernel

Beginning with release 4.2, COHERENT contains a System V-style mechanism for modifying the kernel and building a new bootable kernel.

File **/etc/conf/mtune** defines the suite of "tunable variables" available within the kernel and its drivers. These variables define many of the kernel's default behaviors. For a complete list of these variables and notes on what each does, see **/etc/conf/mtune**.

File **/etc/conf/stune** sets the values of the variables actually used in your kernel. To modify the values of these variable, you can edit **stune** by hand, or you can use the commands **/etc/conf/bin/cohtune** and **/etc/conf/bin/idtune**. The former command lets you set or modify the values of variables used by device drivers; the latter command lets you set or modify variables used in the kernel itself.

File **/etc/conf/mdevice** names the drivers that are available for inclusion in your kernel. File **/etc/conf/sdevice** names the drivers that actually are included in your kernel. To include or exclude a driver, you can modify **sdevice** by hand; or you can use the command **/etc/conf/bin/idenable**.

Command **/etc/conf/bin/idmkcoh** builds a new bootable kernel that incorporates any changes you have made. For your changes to become effective, you must build a new kernel that incorporates your changes, and then boot it.

Finally, command **/etc/conf/bin/idbld** walks you the configuration of every device drivers in the kernel, then invokes **idmkcoh** to link a new kernel. In effect, this command lets you reconfigure the entire kernel.

Each of the above commands and files is described in its own Lexicon entry.

Two other files are of interest if you wish to modify the kernel.

- Header file **<sys/devices.h>** gives the major-device numbers for every driver in your kernel. It is read when drivers are compiled. If you are adding a new driver, you must add its name and major-device number to this header file.
- Normally, when you build a new kernel, the symbol table is stripped from it and kept in file */kernel_name***.sym**. The symbols in this file are used to decipher kernel tracebacks, and can be read by the debugger **db**. However, if you wish to hot-patch a kernel variable, that variable's symbol (or name) must be kept in the binary itself. File **/etc/conf/install_conf/keeplist** names the variables (or, more properly, the symbols) that are left in the binary after it is linked. You can then use the command **/conf/patch** to hot-patch these variables. We discourage you from doing this unless it is absolutely necessary.

Example

The Lexicon entry **device drivers** gives an example of how to add a new driver to the kernel. The following example walks you through the process of changing the size of the buffer cache on your system.

The buffer cache is a reserved portion of memory in which the kernel stores data recently read from the disk or to be written to the disk. When you invoke a command from your command line, the kernel checks its buffer cache. If you had invoked the command recently, the kernel should find it within the buffer cache; and it can then call up the command from memory rather than reading it from the disk. This speeds up your system noticeably.

Like everything else in life, the buffer cache involves a tradeoff: the larger the buffer cache, the faster your system will run, but the less memory will be available for running your programs. By default, COHERENT sets aside a portion of memory for the buffer cache; the more memory you have, the more is set aside for the cache. However, you can set the size of the cache by hand. Usually, this is done to limit the size of the cache, which is necessary if your system has limited amounts of memory and you want to run large user programs (e.g., the X Window System).

The following walks you through the process of modifying the kernel to reduce the size of the buffer cache.

1. Log in as the superuser **root**. **cd** to directory **/etc/conf**.

2. Edit file **/etc/conf/stune** and add the following lines:

```
NBUF_SPEC   100
NHASH_SPEC  97
```

NBUF_SPEC sets the size of the buffer, in blocks. Here, we're setting it to 100 blocks (50 kilobytes), which is very small. **NASH_SPEC** sets the number of hash lists in the kernel; this must be the first prime number smaller than the number of blocks in the cache (in this case, 97). This, too, is very small.

3. Build a new kernel with the following command:

```
/etc/conf/bin/idmkcoh -o /cohtest
```

This builds a new kernel named **cohtest**, which incorporates your changes.

4. Shut down your system and boot the new kernel. For information on how to shut the system down, see the Lexicon entry for **shutdown**. For details on how to boot a kernel other than the default kernel, see the Lexicon entry for **booting**.

That's all there is to it. If you wish to make these variables patchable, so you can change them without going to the bother of building a new kernel, do the following:

1. In the file **/etc/conf/install_conf/keeplist**, change

```
echo '-I SHMMNI:SEMMNI:NMSQID'
```

to

```
echo '-I SHMMNI:SEMMNI:NMSQID:NBUF:NHASH'
```

2. Build a new kernel as described above.

Then, to change limit the size of the buffer cache to 50 kilobytes, use the command:

```
/conf/patch /testcoh NBUF=100 NHASH=97
```

Then, boot the patched kernel. As noted above, you should *not* use **/conf/patch** unless you absolutely must.

Files

/autoboot — The default kernel
/etc/conf — Directory that holds configuration files
/etc/conf/mdevices — Suite of available device drivers
/etc/conf/mtune — Suite of legal patchable variables
/etc/conf/sdevices — Drivers included in kernel
/etc/conf/stune — Patchable variables included in kernel
/etc/conf/install_conf/keeplist — Symbols kept in kernel

See Also

Administering COHERENT, booting, coff.h, COHERENT, device drivers, file, idmkcoh, libc, mtune, stune, ulimit()

Diagnostics

The kernel can produce the following error messages. Most are meaningful only to Mark Williams Company. If you encounter these errors, contact MWC and describe the circumstances during which you saw the error. MWC Support will try to solve this problem for you.

Arena *number* too small *(hardware)*

Bad block *number* (alloc) *(hardware)*
The kernel attempted to allocate a block of memory, only to find that there was something physically wrong with it.

Bad block *number* (free) *(hardware)*
The kernel attempted to free a block of memory, only to find that there was something physically wrong with it.

Bad free *number (hardware)*

Bad freelist *(halt)*
The *freelist* is a list of free blocks on the disk. The COHERENT system maintains this list so it can see where it can write data on the disk. This message indicates that the freelist has been corrupted somehow. To fix this problem, run **/etc/shutdown** to return to single-use mode; use **sync** to flush the buffers; use **umount** to unmount the affected file system; and then run **fsck** to repair the file system.

Bad segment count *(hardware)*
Bus error at *number (hardware)*
Cannot allocate stack *(hardware)*

Cannot create process *(hardware)*
Corrupt arena *(hardware)*
Illegal instruction at *number (hardware)*
Inode *number* busy *(hardware)*
Inode table overflow *(hardware)*
Not a separate I/D machine *(hardware)*

Out of i-nodes *(halt)*

A COHERENT file system has one i-node for each file it maintains. The number of i-nodes is set when the file system is created. If you have numerous small files on a file system, it is possible to exhaust that file system's resources even though the command **df** shows that space remains on the file system. To get around this problem, you must delete files, one file for each i-node needed; or you must use **ar** to archive a mass of files. To do this, first use **/etc/shutdown** to return the system to single-user mode, as described above. Delete files, or use **ar** as described above. Then use **sync** to flush all buffers, and use the command **umount** to unmount the affected file system. Then run **fsck** on the affected file system before rebooting. **fsck** checks COHERENT file systems and fixes them if necessary. Consult the Lexicon entry on **fsck** before you use this program for the first time.

Out of space (*m*,*n*) *(halt)*

When this error message appears, your file system still has i-nodes but the alloted disk space has been exhausted; perhaps you have a few large files that are eating up disk space. To get around this problem, you must delete or compress files to clear up disk space. First, use **/etc/shutdown** to return to single-user mode, as described above; then delete files or compress them as described above until enough space has been cleared to allow you to continue your work. Use **sync** to flush buffer, use **umount** to unmount the affected file system, and run **fsck** on the affected file system. Then reboot.

Random trap *(hardware)*
Raw I/O from non user *(hardware)*
System too large *(hardware)*

keyboard — Technical Information

How COHERENT handles the console keyboard

COHERENT comes with two device drivers for the keyboard, as follows:

vtkb	Non-configurable driver
vtnkb	Configurable driver

To change the keyboard driver you are using, or to modify the behavior of the driver **vtnkb**, log in as the superuser **root** and type:

```
cd /etc/conf
console/mkdev
```

The script **/etc/conf/console/mkdev** walks you through the process of configuring your console: you can switch keyboard drivers from load to non-loadable (or vice-versa), change the number of virtual consoles you use, or change the keyboard-translation table you use by default.

Once you have configured the console, type the following command:

```
bin/idmkcoh -o /kernel_name
```

where *kernel_name* is the what you wish to name the newly built kernel. This builds a new kernel that incorporates the changes you requested. To invoke these changes, simply reboot and invoke kernel *kernel_name* in the usual manner.

See Also

Administering COHERENT, vtkb, vtnkb

kill — Command

Signal a process
kill [- *signal* **]** *pid ...*

COHERENT assigns each active process a unique process id, or *pid*, and uses the pid to identify the process. **kill** sends *signal* to each *pid*. *signal* must be one of the numbers described in the header file **<signal.h>**. The signal can be given by number or by name, as **define**d in these header files. By default, *signal* is **SIGTERM**, which terminates a given process.

LEXICON

If *pid* is zero, **kill** signals each process in the same process group (that is, every process started by the same user from the same tty).

If *pid* is negative (but not -1), **kill** signals every process in the process group whose ID equals the absolute value of *pid*.

If *pid* is -1, **kill** signals each process that you own. If you are logged in as the superuser **root**, this signals every process except processes 0 (the kernel) and 1 (**init**).

The shell prints the process id of a process if the command is detached. The command **ps** prints a list of all active processes, with process ids and command-line arguments.

A user can kill only the processes he owns; the superuser, however, can kill anything. A process cannot ignore or catch **SIGKILL**.

See the Lexicon article for **signal()** for a table of the signals and what each means.

Files

<signal.h> — Signals

See Also

commands, getpid(), init, kill(), ksh, ps, sh, signal()

kill() — System Call (libc)

Kill a system process
#include <signal.h>
kill(*pid, sig***)**
int *pid*, *sig*;

kill() is the COHERENT system call that sends a signal to a process. *pid* is the process identifier of the process to be signalled, and *sig* identifies the signal to be sent, as set in the header file **signal.h**. This system call is most often used to kill processes, hence its name.

See the Lexicon article for **signal()** for a table of the signals and what each means. If *sig* is zero, **kill()** performs error checking, but sends no signal. You can use this feature to check the validity of *pid*.

Example

For an example of using this system call in a C program, see **signal()**.

See Also

libc, signal(), signal.h
POSIX Standard, §3.3.2

ksh — Command

The Korn shell
ksh *token ...*

The COHERENT system offers two command interpreters: **sh**, the Bourne shell; and **ksh**, the Korn shell. **sh** is the default COHERENT command interpreter. The shell tutorial included in this manual describes the Bourne shell in detail.

This article describes **ksh**, the Korn shell. **ksh** is a superset of the Bourne shell, and contains many features that you may well find useful. These include MicroEMACS-style editing of command lines; command hashing; a full-featured aliasing feature; and a job-control facility.

Invoking ksh

To invoke **ksh** from within the Bourne shell, simply type **ksh** at the command-line prompt. To use **ksh** as your default shell, instead of **sh**, append the command **/usr/bin/ksh** to the end of your entry in the file **/etc/passwd**. (See the Lexicon entry for **passwd** for more information on this file.)

You can invoke **ksh** with one or more built-in options; these are described below.

Commands

A *command* consists of one or more *tokens*. A *token* is a string of text characters (i.e., one or more alphabetic characters, punctuation marks, and numerals) delineated by spaces, tabs, or newlines.

A *simple command* consists of the command's name, followed by zero or more tokens that represent arguments to the command, names of files, or shell operators. A *complex command* will use shell constructs to execute one or more commands conditionally. In effect, a complex command is a mini-program that is written in the shell's programming language and interpreted by **ksh**.

Shell Operators

The shell includes a number of operators that form pipes, redirect input and output to commands, and let you define conditions under which commands are executed.

command **|** *command*

The *pipe* operator: let the output of one command serve as the input to a second. You can combine commands with '|' to form *pipelines*. A pipeline passes the standard output of the first (leftmost) command to the standard input of the second command. For example, in the pipeline

```
sort customers | uniq | more
```

ksh invokes **sort** to sort the contents of file **customers**. It pipes the output of **sort** to the command **uniq**, which outputs one unique copy of the text that is input into it. **ksh** then pipes the output of **uniq** to the command **more**, which displays it on your terminal one screenful at a time. Note that under COHERENT, unlike MS-DOS, pipes are executed concurrently: that is, **sort** does not have to finish its work before **uniq** and **more** can begin to receive input and get to work.

command **;** *command*

Execute commands on a command line sequentially. The command to the left of the ';' executes to completion; then the command to the right of it executes. For example, in the command line

```
a | b ; c | d
```

first execute the pipeline **a | b** then, when **a** and **b** complete, execute the pipeline **c | d**.

command **&**

Execute a command in the background. This operator must follow the command, not precede it. It prints the process identifier of the command on the standard output, so you can use the **kill** command to kill that process should something go wrong. This operator lets you execute more than one command simultaneously. For example, the command

```
/etc/fdformat -v /dev/fha0 &
```

formats a high-density, 5.25-inch floppy disk in drive 0 (that is, drive A); but while the disk is being formatted, **ksh** returns the command line prompt so you can immediately enter another command and begin to work. If you did not use the '&' in this command, you would have to wait until formatting was finished before you could enter another command.

ksh also prints a message on your terminal when a command that you are running in the background finishes processing. It does not check these "child" processes very often, however, so a command may have finished some time before **ksh** informs you of the fact. See the Lexicon article for the command **ps** for information on all processes; also see the description of the built-in command **jobs**, below.

command **&&** *command*

Execute a command upon success. **ksh** executes the command that follows the token '&&' only if the command that precedes it returns a zero exit status, which signifies success. For example, the command

```
cd /etc
fdformat -v /dev/fha0 && badscan -o proto /dev/fha0 2400
```

formats a floppy disk, as described above. If the format was successful, it then invokes the command **badscan** to scan the disk for bad blocks; if it was not successful, however, it does nothing.

command **||** *command*

Execute a command upon failure. This is identical to operator '&&', except that the second command is executed if the first returns a non-zero status, which signifies failure. For example, the command

```
/etc/fdformat -v /dev/fha0 || echo "Format failed!"
```

formats a floppy disk. If formatting failed, it echoes the message **Format failed!** on your terminal; however, if formatting succeeds, it does nothing.

Note that the tokens newline, ';' and '&' bind less tightly than '&&' and '||'. **ksh** parses command lines

from left to right if separators bind equally.

>*file* Redirect standard output. The *standard input*, *standard output*, and *standard error* streams are normally connected to the terminal. A pipeline attaches the output of one command to the input of another command. In addition, **ksh** includes a set of operators that redirect input and output into files rather than other commands.

The operator '>' redirects output into a file. For example, the command

```
sort customers >customers.sort
```

sorts file **customers** and writes the sorted output into file **customers.sort**. It creates **customers.sort** if it does not exist, and destroys its previous contents if it does exist.

>>*file* Redirect output into a file, and append. If the file does not exist, this operator creates it; however, if the file already exists, this operator appends the output to that file's contents rather than destroying those contents. For example, the command

```
sort customers.now | uniq >>customers.all
```

sorts file **customers.now**, pipes its output to command **uniq**, which throws away duplicate lines of input, and appends the results to file **customers.all**.

<*file* Redirect standard input. Here, **ksh** reads the contents of a file and processes them as if you had typed them from your keyboard. For example, the command

```
ed textfile <edit.script
```

invokes the line-editor **ed** to edit **textfile**; however, instead of reading editing commands from your keyboard, the shell passes **ed** the contents of **edit.script**. This command would let you prepare an editing script that you could execute repeatedly upon files rather than having to type the same commands over and over.

<< *token*
Prepare a "here document". This operator tells **ksh** to accept standard input from the shell input until it reads a line that contains only *token.* For example, the command

```
cat >FOO <<\!
      Here is some text.
!
```

redirects all text between '<<\!' and '!' to the **cat** command. The '>' in turn redirects the output of **cat** into file **FOO**. **ksh** performs parameter substitution on the here document unless the leading *token* is quoted; parameter substitution and quoting are described below.

command **2>** *file*
Redirect the standard error stream into a file. For example, the command

```
nroff -ms textfile >textfile.p 2>textfile.err
```

invokes the command **nroff** to format the contents of **textfile**. It redirects the output of **nroff** (i.e., the standard output) into **textfile.p**; it also redirects any error messages that **nroff** may generate into file **textfile.err**.

Note in passing that a command may use up to 20 streams. By default, stream 0 is the standard input; stream 1 is the standard output; and stream 2 is the standard error. **ksh** lets you redirect any of these streams individually into files, or combine streams into each other.

<&*n* **ksh** can redirect the standard input and output to duplicate other file descriptors. (See the Lexicon article **file descriptor** for details on what these are.) This operator duplicates the standard input from file descriptor *n*.

>&*n* Duplicate the standard output from file descriptor *n*. For example,

```
2>&1
```

redirects file descriptor 2 (the standard error) to file descriptor 1 (the standard output).

Note that each command executed as a foreground process inherits the file descriptors and signal traps (described below) of the invoking shell, modified by any specified redirection. Background processes take input from the null device **/dev/null** (unless redirected), and ignore interrupt and quit signals.

File-Name Patterns

The shell interprets an input token that contain any of the special characters '?', '*', or '[' as a file name *pattern*.

? Match any single character except newline. For example, the command

```
ls name?
```

will print the name of any file that consists of the string **name** plus any one character. If **name** is followed by no characters, or is followed by two or more characters, it will not be printed.

* Match a string of non-newline characters of any length (including zero).

```
ls name*
```

prints the name of any file that begins with the string **name**, regardless of whether it is followed by any other characters. Likewise, the command

```
ls name?*
```

prints the name of any file that consists of the string **name** followed by at least one character. Unlike **name***, the token **name?*** must be followed by at least one character before it will be printed.

~*name*
Replace the name of user *name* with his **$HOME** directory. For example, the command

```
ls -l ~norm/src
```

lists the contents of the *src* subdirectory located under the **$HOME** directory for user **norm**. This spares you from having to know where a given user's HOME directory is located.

The character '~' on its own is a synonym for the home directory of whoever is running the command. For example, the command

```
/usr/lib/uucp FOO mwcbbs:~
```

copies file **FOO** into directory **/usr/spool/uucppublic** on system **mwcbbs**. In this instance, '~' expands into **/usr/spool/uucppublic** because the command **uucico** invokes **setuid()** to change the ownership of the process to user **uucp**, whose home directory is **/usr/spool/uucppublic**.

[!*xyz***]** Exclude characters *xyz* from the string search. For example, the command

```
ls [!abc]*
```

prints all files in the current directory except those that begin with **a**, **b**, or **c**.

[*C-c***]** Enclose alternatives to match a single character. A hyphen '-' indicates a range of characters. For example, the command

```
ls name[ABC]
```

will print the names of files **nameA**, **nameB**, and **nameC** (assuming, of course, that those files exist in the current directory). The command

```
ls name[A-K]
```

prints the names of files **nameA** through **nameK** (again, assuming that they exist in the current directory).

When **ksh** reads a token that contains one of the above characters, it replaces the token in the command line with an alphabetized list of file names that match the pattern. If it finds no matches, it passes the token unchanged to the command. For example, when you enter the command

```
ls name[ABC]
```

ksh replaces the token **name[ABC]** with **nameA**, **nameB**, and **nameC** (again, if they exist in the current directory), so the command now reads:

```
ls nameA nameB nameC
```

It then passes this second, transformed version of the command line to the command **ls**.

Note that the slash '/' and leading period '.' must be matched explicitly in a pattern. The slash, of course, separates the elements of a path name; while a period at the begin of a file name usually (but not always) indicates that that file has special significance.

LEXICON

Pattern Matching in Prefixes and Suffices

Special constructs let you match patterns in the prefixes and suffices of a string:

{**#***parameter*}
This operator gives the number of characters in *parameter*. For example:

```
foo=BAZZ
echo ${#foo} -> 4
```

{*parameter***%***word*}
This returns the shortest string in which the suffix of *parameter* matches *word*. For example, given that **xyzzy=usr/bin/cpio**, then the command

```
echo ${xyzzy%/*}
```

echoes the string **usr/bin**.

{*parameter***%%***word*}
This returns the longest string in which the suffix of *parameter* matches *word*. For example, given that **xyzzy=usr/bin/cpio**, then the command

```
echo ${xyzzy%/*}
```

echoes the string **usr**.

{*parameter***#***word*}
This returns the shortest string in which the prefix of *parameter* matches *word*. For example, given that **plugh=usr/bin/cpio**, the command

```
echo ${plugh#*/}
```

echoes **bin/cpio**.

{*parameter***##***word*}
This returns the longest string in which the prefix of *parameter* matches *word*. For example, given that **plugh=usr/bin/cpio**, the command

```
echo ${plugh##*/}
```

echoes **cpio**.

The following shows how to use these expressions to implement the command **basename**:

```
basename () {
      set $(echo ${1##*/}) $2
      echo ${1%$2}
}
```

Quoting Text

From time to time, you will want to "turn off" the special meaning of characters. For example, you may wish to pass a token that contains a literal asterisk to a command; to do so, you need a way to tell **ksh** not to expand the token into a list of file names. Therefore, **ksh** includes the **quotation operators** '\', '"', and '"; these "turn off" (or *quote*) the special meaning of operators.

The backslash '\' quotes the following character. For example, the command

```
ls name\*
```

lists a file named **name***, and no other.

The shell ignores a backslash immediately followed by a newline, called a *concealed newline*. This lets you give more arguments to a command than will fit on one line. For example, the command

```
cc -o output file1.c file2.c file3.c \
      file4.c file5.c file19.c
```

invokes the C compiler **cc** to compile a set of C source files, the names of which extend over more than one line of input. You will find this to be extremely helpful, especially when you write scripts and **makefile**s, to help you write neat, easily read commands.

A pair of apostrophes ' ' prevents interpretation of any enclosed special characters. For example, the command

```
find . -name '*.c' -print
```

finds and prints the name of any C-source file in the current directory and any subdirectory. The command **find** interprets the '*' internally; therefore, you want to suppress the shell's expansion of that operator, which is accomplished by enclosing that token between apostrophes.

A pair of quotation marks " " has the same effect. Unlike apostrophes, however, **ksh** will perform parameter substitution and command-output substitution (described below) within quotation marks. Note that everything between quotation marks will be a single argument, even if there are spaces between the tokens. For example, the command

```
grep "x y" *.c
```

calls the string-search command **grep** to look for the string **x<space>y**.

Scripts

Shell commands can be stored in a file, or *script.* The command

```
ksh script [ parameter ... ]
```

executes the commands in *script* with a new subshell **ksh**. Each *parameter* is a value for a positional parameter, as described below.

If you have used the command **chmod** to make *script* executable, then it is executed under the Bourne shell **sh**, without requiring the **ksh** command. Because all executable scripts are executed by the Bourne shell by default, not the Korn shell, you should avoid constructions that are unique to the Korn shell.

To ensure that a script is executed by **ksh**, begin the script with the line:

```
#!/usr/bin/ksh
```

Parameters of the form '$*n*' represent command-line arguments within a script. *n* can range from zero through nine; **$0** always gives the name of the script. These parameters are also called *positional parameters*.

If no corresponding parameter is given on the command line, the shell substitutes the null string for that parameter. For example, if the script **format** contains the following line:

```
nroff -ms $1 >$1.out
```

then invoking **format** with the command line:

```
format mytext
```

invokes the command **nroff** to format the contents of **mytext**, and writes the output into file **mytext.out**. If, however, you invoke this command with the command line

```
format mytext yourtext
```

the script will format **mytext** but ignore **yourtext** altogether.

Reference **$*** represents all command-line arguments. If, for example, we change the contents of script **format** to read

```
nroff -ms $* >$1.out
```

then the command

```
format mytext yourtext
```

will invoke **nroff** to format the contents of **mytext** and **yourtext**, and write the output into file **mytext.out**.

Commands in a script can also be executed with the . (dot) command. It resembles the **ksh** command, but the current shell executes the script commands without creating a new subshell or a new environment; therefore, you cannot use command-line arguments.

Variables

Shell variables are names that can be assigned string values on a command line, in the form

name=value

The name must begin with a letter, and can contain letters, digits, and underscores '_'. Note that no white space can appear around the '=', or the assignment will not work.

LEXICON

In shell input, '$*name*' or '${*name*}' represents the value of the variable. For example:

```
TEXT=mytext
nroff -ms $TEXT >$TEXT.out
```

Here, **ksh** expands **$TEXT** before it executes the **nroff** command. This technique is very useful in large, complex scripts: by using variables, you can change the behavior of the script by editing one line, rather than having to edit numerous variables throughout the script.

Note that if an assignment precedes a command on the same command line, the effect of the assignment is local to that command; otherwise, the effect is permanent. For example,

```
kp=one testproc
```

assigns variable **kp** the value **one** only for the execution of the script **testproc**.

ksh sets the following variables by default:

The number of actual positional parameters given to the current command.

@ The list of positional parameters "$1 $2 ...".

* The list of positional parameters "$1" "$2" ... (the same as '$@' unless some parameters are quoted).

- Options set in the invocation of the shell or by the **set** command.

? The exit status returned by the last command.

! The process number of the last command invoked with '&'.

$ The process number of the current shell.

Environmental Variables

ksh references the following environmental variables:

ENV If this variable is set at start-up, after all **.profile** files have been executed, the expanded value is used as the shell's start-up file. It typically defines environmental variables and aliases.

FCEDIT
This sets the editor used by the command **fc**.

HOME Initial working directory; usually specified in the password file **/etc/passwd**.

IFS Delimiters for tokens; by default space, tab, and newline.

KSH_VERSION
The current version of the Korn shell that you are using.

MAIL **ksh** check the file this names, at intervals specified by environmental variable **MAILCHECK**. If the file specified by this variable is new since last checked, the shell prints "You have mail." on the your terminal. If the file has increased in size since the last check, **ksh** prints "You have new mail." on your terminal.

Note that by default, **ksh** does not check **MAIL** when you log in. If you want it to do so, add the following lines to file **/etc/.kshrc**:

```
# The following lines emulate the mail notification of the Bourne Shell.
if [ -s $MAIL ]
then
      echo "You have mail."
fi
```

MAILCHECK
Specifies the number of seconds between checking for new mail. If not specified, **MAILCHECK** defaults to 600 seconds (ten minutes).

PATH Colon-separated list of directories searched for commands.

PS1 First prompt string, usually '$'. Note that in this variable and **PS2**, **ksh** expands the symbol **!** into the current number of the command line. For example, the prompt **ksh !>** prints the prompt **ksh** *NN***>** with every command, where *NN* is the number of the current command. This is useful when you have enabled

the history feature, as described below.

To print a prompt that includes your local site name, include the variable **$PWD** (described below) in the definition of **PS1**. For example,

```
PS1='$PWD>'
```

prints the current directory as your prompt, just like MS-DOS does. To include your system's name, read the contents of file **/etc/uucpname**, as follows:

```
SITE=`cat /etc/uucpname`
PS1='$SITE!!$PWD>'
```

This form of the prompt is quite useful when you are working on networked machines and may not always be sure just what system you are working on. Note that two exclamation points are necessary; as noted above, **ksh** expands one '!' into the number of the current command.

Finally, to include the command number with site name and current directory, do the following:

```
SITE=`cat /etc/uucpname`
PS1='$SITE!!$PWD !>'
```

This will give you a very long prompt, but one with much information in it.

PS2 Second prompt string, usually '>'. **ksh** prints it when it expects more input, such as when an open quotation-mark has been typed but a close quotation-mark has not been typed, or within a shell construct.

PWD The present working directory, i.e., the directory within which you are now working.

SECONDS
The number of seconds since the current shell was started.

SHELL The full path name of the shell that you are now executing.

TERM The name of the type of terminal you are now using, as used by various programs for reading the file **/etc/termcap**.

TIMEZONE
The current timezone you are located in, as set in your **.profile**. This is an interesting and powerful variable; see its entry in the Lexicon for details.

USER The login-identifier of the user, i.e., you.

The following special forms substitute parameters conditionally:

${*name-token***}**
Substitite *name* if it is set; if it is not, substitute *token*.

${*name***=***token***}**
Substitute *name* if it is set; if it is not set, substitute *token* and set *name* to equal *token*.

${*name***+***token***}**
Substitute *token* if *name* is set.

${*name***?***token***}**
Substitute *name* if it is set; if it is not, print *token* and exit from the shell.

To unset an environmental variable, use the command **unset**.

Command Output Substitution

ksh can use the output of a command as shell input (as command arguments, for example) by enclosing the command in grave characters ``. For example, to list the contents of the directories named in file **dirs**, use the command

```
ls -l `cat dirs`
```

Constructs

ksh lets you control the execution of programs through the following constructs. It recognizes a construct only if it occurs unquoted as the first token of a command. This implies that a separator must precede each reserved word in the following constructs; for example, newline or ';' must precede **do** in the **for** construct.

LEXICON

break *[n]*
Exit from **for**, **until**, or **while**. If *n* is given, exit from *n* levels.

case *token* **in [** *pattern [| pattern] ...***)** *sequence***;;] ... esac**
Check *token* against each *pattern*, and execute *sequence* associated with the first matching *pattern*.

continue *[n]*
Branch to the end of the *n*th enclosing **for**, **until**, or **while** construct.

for *name [* **in** *token ...]* **do** *sequence* **done**
Execute *sequence* once for each *token*. On each iteration, *name* takes the value of the next *token*. If the **in** clause is omitted, **$@** is assumed. For example, to list all files ending with **.c:**

```
for i in *.c
do
      cat $i
done
```

if *seq1* **then** *seq2 [* **elif** *seq3* **then** *seq4] ... [* **else** *seq5]* **fi**
Execute *seq1*. If the exit status is zero, execute *seq2*; if not, execute the optional *seq3* if given. If the exit status of *seq3* is zero, then execute *seq4*, and so on. If the exit status of all tested sequences is nonzero, execute *seq5*.

time *sequence*
Time how long it takes *sequence* to execute. When *sequence* has finished executing, the time is displayed on the standard output.

while *sequence1 [* **do** *sequence2]* **done**
Execute *sequence2* as long as the execution of *sequence1* results in an exit status of zero.

(*sequence***)**
Execute *sequence* within a subshell. This allows *sequence* to change the current directory, for example, and not affect the enclosing environment.

{*sequence*}
Braces simply enclose a *sequence*.

Built-in Commands

ksh executes most commands via the **fork** system call, which creates a new process. See the Lexicon articles on **fork()** and **exec** for details on these calls. **ksh** also has the following commands built into itself.

. *script* Read and execute commands from *script*. Positional parameters are not allowed. **ksh** searches the directories named in the environmental variable **PATH** to find the given *script*.

: *[token ...]*
A colon ':' indicates a "partial comment". **ksh** normally ignores all commands on a line that begins with a colon, except for redirection and such symbols as **$**, **{**, **?**, etc.

A complete comment: if **#** is the first character on a line, **ksh** ignores all text that follows on that line.

alias *[name***=***value ...]*
When called without arguments, **alias** prints all aliases and their values. When called with a *name* but no associated value, then it prints the value of *name*. When called with a *name* and *value* combination, it associated *value* with *name*.

For example, the command

```
alias logout='exit'
```

binds the token **logout** to the command **exit**: hereafter, whenever you type **logout**, it will be as if you typed the **exit** command.

Note that when you define an alias, you should be careful not to write one that is self-referring, or **ksh** will go into an infinite loop when it tries to expand the alias. For example, the definition:

```
# DO NOT DO THIS!
alias ls='ls -CF'
```

will send **ksh** into an infinite loop, as it tries infinitely to replace **ls** with **ls**. Rather, use the definition:

```
# THIS IS CORRECT
alias ls='/bin/ls -CF'
```

or

```
# THIS TOO IS CORRECT
alias ls='  ls -CF'
```

In the latter example, note the spaces between the first grave character and the **ls**.

ksh has a number of aliases set by default. See the section **Aliases**, below, for details.

bind *[-m] [key_sequence=binding_name ...]*
When called without arguments, list the current set of key bindings for MicroEMACS-style editing of command lines. When called with arguments, bind the *key_sequence* to *binding_name.*

For example, the command

```
bind '^[^H'=delete-word-backward
```

binds the editing command **delete-word-backward** to the key sequence **<esc><backspace>**. Note that the carat characters in this command are literally that, not the shell's representation of a literal **<esc>** or **<backspace>** character.

When called with the **-m** option, bind more than one *binding_name* to a given *key_sequence*. This lets you build keyboard macros, to perform complex editing tasks with one or two keystrokes.

For details, see the sections below on command-line editing.

builtin *command*
Execute *command* as a built-in command.

cd *dir* Change the working directory to *dir.* If no argument is given, change to the home directory as set by the environmental variable **HOME**. When invoked, it also changes the environmental variables **PWD** and **OLDPWD**.

Using a hyphen '-' as the argument causes **ksh** to change to the previous directory, i.e., the one indicated by shell variable **OLDPWD**. In effect, this swaps **OLDPWD** and **PWD**, thus allowing you to flop back and forth easily between two directories.

echo *token ...*
Echo *token* onto the standard output. **ksh** replaces the command **echo** with the alias **echo='print'**.

eval *[token ...]*
Evaluate each *token* and treat the result as shell input.

exec *[command]*
Execute *command* directly rather than as a subprocess. This terminates the current shell.

exit *[status]*
Set the exit status to *status*, if given, then terminate; otherwise, the previous status is used.

export *[name ...]*
ksh executes each command in an *environment*, which is essentially a set of shell variable names and corresponding string values. It inherits an environment when invoked, and normally it passes the same environment to each command it invokes. **export** specifies that the shell should pass the modified value of each given *name* to the environment of subsequent commands. When no *name* is given, **ksh** prints the name of each variable marked for export.

export *VARIABLE=value*
This form of the **export** command sets *VARIABLE* to *value*, and exports it. Thus, the command

```
export FOO=bar
```

is equivalent to the commands:

```
FOO=bar
export FOO
```

fc *[-**l**] [-**n**] [first [last]]*

Draw the previously executed commands *first* through *last* back for manipulation and possible execution. *first* and *last* can be referenced either by their history numbers, or by a string with which the command in question begins. Normally, the commands are pulled into an editor for manipulation before they are executed; the editor is defined by the environmental variable **FCEDIT** (default, **ed**). The commands in question are executed as soon as you exit from the editor. Option **-l** lists the command(s) on **stdout**, and so suppresses the editing feature. Option **-n** inhibits the default history numbers.

function *funcname* { *script* }

Define function **funcname** for the shell to execute. For example the following defines function **get_name** for the shell:

```
function get_name {
      echo -n Please enter your name ...
      read name
      return 0
}
```

When **ksh** encounters **get_name**, it runs the above-defined function, rather than trying to find **get_name** on the disk. Note that the return status can be any valid status and can be checked in the code that called **get_name** by reading the shell variable **$?** (described above), or by using the function as the argument to an **if** statement. This allows you to build constructs like the following:

```
if get_name; then
      do_something
else
      do_something_else
fi
```

To list all defined functions, type the alias **functions**. To receive detailed information on a defined function, use the command **type** *name* where *name* is the name of the function in which you are interested.

getopts *optstring name [arg ...]*

Parse the *arg*s to *command*. See the Lexicon entry for **getopts** for details.

hash *[-**r**] [name ...]*

When called without arguments, **hash** lists the path names of all hashed commands. When called with *name* **hash** check to see if it is an executable command, and if so adds it to the shell's hash list. The **-r** option removes *name* from the hash list.

kill *[-**l**] [signal] process ...*

Send *signal* to *process*. The default signal is **TERM**, which terminates the process. *signal* may either be a number or a mnemonic as **#define**d in header file **<signal.h>**. When called with the **-l** option, it lists all known types of signals. See the Lexicon entry for **kill** for details.

let *[expression]*

Evaluate each *expression*. This command returns zero if *expression* evaluates to non-zero (i.e., fails), and returns non-zero if it evalutes to zero (i.e., succeeds). This is useful for evaluating expressions before actually executing them.

print *[-**nreu***n**] [argument ...]*

Print each *argument* on the standard output, separated by spaces and terminated with a newline. Option **-n** suppresses printing of the newline. Option **-u***n* redirects output from the standard output to file descriptor *n*.

Note that each *argument* can contain the following standard C escape characters: **\b**, **\f**, **\n**, **\r**, **\v**, and **\###**. See the Lexicon article on **C Language** for details each character's meaning. The option **-r** inhibits this feature, and the **-e** option re-enables it.

read *name ...*

Read a line from the standard input and assign each token of the input to the corresponding shell variable *name*. If the input contains fewer tokens than the *name* list, assign the null string to extra variables. If the input contains more tokens, assign the last *name* the remainder of the input.

readonly *[name ...]*

Mark each shell variable *name* as a read-only variable. Subsequent assignments to read-only variables will not be permitted. With no arguments, print the name and value of each read-only variable.

return *[status]*
Return *status* to the parent process.

set *[-***aefhkmnuvx** *[-***o** *keyword] [name ...]]*
Set listed flag. The **-o** option sets *keyword,* where *keyword* is a shell option.

When used with one or more *names,* this command sets shell variables *name* to values of positional parameters beginning with **$1**.

For example, the command

```
set -h -o emacs ignoreeof
```

performs the following: turns on hashing for all commands, turns on MicroEMACS-style command-line editing, and turns off exiting upon EOF (that is, you must type **exit** to exit from the shell). **set** commands are especially useful when embedded in your **.profile**, where they can customize **ksh** to your preferences.

For details of this command, see its Lexicon entry.

shift Rename positional parameter **1** to current value of **$2,** and so on.

test *[option] [expression]*
Check *expression* for condition *option.* This is a useful and complex command, with more options than can be listed here. See its Lexicon entry for details.

times Print on the standard output a summary of processing time used by the current shell and all of its child processes.

trap *[command] [n ...]*
Execute *command* if **ksh** receives signal *n.* If *command* is omitted, reset traps to original values. To ignore a signal, pass null string as *command.* With *n* zero, execute *command* when the shell exits. With no arguments, print the current trap settings.

typeset *[-***firx***] [+***firx***] [name [=***value*] ...]
When called without an argument, this command lists all variables and their attributes.

When called with an option but without a *name*, it lists all variables that have the specified attribute; **-** tells **typeset** to list the value of each variable and **+** tells it not to.

When called with one or more *names,* it gives *name* to the listed attribute. If *name* is associated with a *value,* **typeset** also assigns the *value* to it.

typeset recognizes the following attributes:

- **-i** Store variable's value as an integer
- **-f** List function instead of variable
- **-r** Make the variable read-only
- **-x** Export variable to the environment

umask *[nnn]*
Set user file creation mask to *nnn.* If no argument is given, print the current file creation mask.

unalias *name ...*
Remove the alias for each *name*.

wait *[pid]*
Hold execution of further commands until process *pid* terminates. If *pid* is omitted, wait for all child processes. If no children are active, this command finishes immediately.

whence *[-***v***] name ...*
List the type of command for each *name.* When called with the **-v** option, also list functions and aliases.

Aliases

ksh implements as aliases a number of commands that **sh** calls as separate executable programs. The **echo** alias, for instance, does everything that **/bin/echo** does, but **ksh** does not have to **fork()** and **exec()** simply to echo a token. Other aliases, like **pwd**, work by printing the contents of shell variables. The command **/bin/pwd** still works should you prefer it, but you must request it by its full path name should you not wish to use the much faster alias version.

LEXICON

ksh sets the following aliases by default. If you wish, you can use the built-in command **unalias** to make one or all of them go away.

```
echo=print
false=let
functions=typeset -f
history=fc -l
integer=typeset -i
login=exec login
newgrp=exec newgrp
pwd=print -r $PWD
true=:
type=whence -v
```

The alias **history** is especially useful when you are using the Korn shell's history feature. When invoked with no argument, it prints the last 13 commands you typed. When invoked with one numeric argument, it lists the command that corresponds to that argument; for example

```
history 106
```

prints the 106th command you entered (assuming that you've entered that many). When used with two numeric arguments, it prints the range of commands between the two arguments; for example

```
history 10 99
```

prints the tenth through the 99th commands you entered.

Job Control

ksh lets you manipulate and monitor background jobs via its *job control* commands.

The following commands manipulate background jobs:

jobs Display information about all controlled jobs. Information is in the following format:

%*num* **[+-]** *pid status command*

where *num* indicates the job number, '+' indicates the current job, '-' indicates the previous job, *pid* is the job's process identifier, *status* shows the status of the job (e.g., Running, Done, Killed), and *command* is the command description. Note that **ksh** only checks for changes in job status when waiting for a command to complete.

kill *[-signal] pid ...*
Described above.

wait *[pid]*
Hold execution of further commands until process *pid* terminates. See its Lexicon entry for details.

The following '%' syntax can be used with the above commands:

%+ Select the current job.

%- Select the previous job.

%*num* Select the job with job number *num*.

%*string* Select the most recently invoked job whose command begins with *string*.

%?*string*
Select the most recently invoked job whose command contains *string*.

vi-Style Command-line Editing

ksh, has built into it an editing feature that lets you recall and edit commands using **vi**-style editing commands. When you have finished editing, simply typing `(¢)` dispatches the command for re-execution.

To turn on **vi**-style editing, use the command

```
set -o vi
```

The following table gives each input-mode command:

**** Escape the next erase or kill character.

<ctrl-D> This character (EOF) terminates **ksh** if the current line is empty. Note that the command

```
alias logout='exit'
```

neutralizes this effect of EOF.

<ctrl-H> Delete previous character — that is, the character to the left.

<ctrl-V> Quote the next character. You can use this to embed editing and kill characters within a command.

<ctrl-W> Delete the previous word. A "word" is any clump of text delineated by white space.

<ctrl-J>
<ctrl-M>
(↵) Execute this line.

The following table gives each editing-mode command:

[count] **k**
[count] **-** Get previous command from the history buffer.

[count] **j**
[count] **+** Get next command from the history buffer.

[count] **G** Get command *count* from the history buffer. Default is the least recently entered command.

/*string* Search the history buffer for the most recently entered command that contains *string*. If *string* is NULL, use the previous string. *string*. must be terminated by **<ctrl-M>** or **<ctrl-J>**.

?*string* Same as **/**, except that **ksh** seeks the least recently entered command.

n Repeat the previous search.

N Repeat the last search, but in the opposite direction.

[count] **l** Move right *count* characters (default, one).

[count] **w** Move forward *count* alphanumeric words (default, one).

[count] **W** Move forward *count* blank-separated words (default, one).

[count] **e** Move forward to the end of the *count*'th word.

[count] **E** Move forward to the end of the *count*'th blank-separated word.

[count] **h** Move left *count* characters (default, one).

[count] **b** Move back *count* words.

[count] **B** Move back *count* blank-separated words.

0 Move cursor to start of line.

^ Move cursor to start of line.

$ Move cursor to end of line.

[count] **f** *c* Move rightward to the *count*'th occurrence of character *c*.

[count] **B** *c* Move leftward to the *count*'th occurrence of character *c*.

[count] **t** *c* Move rightward *almost* to the *count*'th occurrence of character *c* (default, one). Same as **f***c* followed by **h**.

[count] **T** *c* Move leftward *almost* to the *count*'th occurrence of character *c* (default, one). Same as **F***c* followed by **l**.

; Repeats the last **f**, **F**, **t**, or **T** command.

LEXICON

, Reverse of **;**.

a Enter input mode and enter text after the current character.

A Append text to the end of the line; same as **$a**.

*[count]***c***c*
c*[count]c* Delete current character through character *c* and then execute input line.

s Same as **cc**.

*[count]***d***motion*
d*[count]c* Delete current character through the character *c*.

D Delete current character through the end of line. Same as **d$**.

i Enter input mode and insert text before the current character.

I Enter input mode and insert text before the first word on the line.

*[count]***P** Place the previous text modification before the cursor.

*[count]***p** Place the previous text modification after the cursor.

R Enter input mode and overwrite characters on the line.

r*c* Replace the current character with character *c*.

*[count]***x** Delete the current character.

*[count]***X** Delete the preceding character.

*[count]***.** Repeat the previous text modification command

~ Invert the case of the current character and advance the cursor.

*[count]_* Append the *count*'th word from the previous command and enter input mode (default, last word).

* Attempt file-name generate on the current word. If a match is found, replace the current word with the match and enter input mode.

u Undo the last text-modification command.

U Restore the current line to its original state.

*[count]***v** Execute command

```
fc -e ${VISUAL:-${EDITOR:-vi}}
```

<ctrl-L> Line feed and print the current line.

<ctrl-J>
<ctrl-M>
(↵) Execute the current line.

Same as **I#<return>**.

Command Completion

ksh supports *command completion*. This feature permits you to invoke a command by typing only a fraction of it; **ksh** fleshes out the command, based on what commands you have already entered.

To invoke command completion, set the following in **.profile** or **.kshrc**:

```
set -h -o emacs
```

or:

```
set -h -o vi
```

This turns on hashing and tracking. It also turn on command-line editing: the former command turns on MicroEMACS-style editing, whereas the latter turn on **vi**-style editing.

As an example, say that you type the following commands:

```
compress foo.tar
ps alx
df -t
```

With MicroEMACS-style editing, if you type **<ctrl-X>?**, you then see the commands you typed in alphabetical order:

```
compress     df     ps
```

If you want to re-invoke the **compress** command without having to type all of it, you can use either type **<ctrl-R>** followed by 'c' to use the shell's reverse-search capabilities; or you can type 'c' followed by **<esc><esc>** to have the shell's command-completion facility complete the command.

If you use the reverse-incremental search, you get the entire command line as you had typed it. Additional uses of **<ctrl-R>** while already in search mode tell **ksh** to search further back in its history list of commands.

If, however, you use the command completion, you get only the command. So, to continue the example, if you type the letter 'c' followed by **<esc><esc> ksh** displays the word **compress**, followed by a **<space>**, and awaits more input.

In general, the reverse-search is better if you wish to re-execute an entire command; but command completion is better if you want just the command name.

Under **vi**-style editing, you can also use command completion. To complete a command, type '*' while in edit mode; or type **<esc>*** while in input mode.

File-Name Completion

ksh also lets you "complete" file names and directory names, just like you complete command names. With MicroEMACS-style editing, the file-completion command is **<esc><esc>**; with **vi**-style editing, the file-completion command is '*' (in edit mode) or **<esc>*** (in input mode).

If you are entering a file name and have specified enough of the name in order to specify a unique file, typing the file-completion command completes the file name or directory name. If you have not typed enough, **ksh** remains silent; type more characters of the file name, then again try the file-completion command. If you enter a bogus file name or directory name, **ksh** beeps to indicate that it cannot complete the given name. When **ksh** completes a file name, it then prints a space character. This indicates that the string names a file (rather than a directory); the space character lets you begin immediately to type the next argument. When **ksh** completes a directory name, it appends a slash ('/') instead of a space character, and waits for you to type the next part of the path name.

For example, if you type

```
ls -l /usr/spool/uucp
```

followed by **<esc><esc>**, nothing happens because of the ambiguity between directory names **/usr/spool/uucp/** and **/usr/spool/uucppublic/**.

If you then type the letter 'p', the command now appears:

```
ls -l /usr/spool/uucpp
```

Typing **<esc><esc>** now expands it out to

```
ls -l /usr/spool/uucppublic/
```

which is the name you desire. Note that **ksh** appends the trailing slash and waits for more.

A file-name completion example is:

```
more /usr/lib/uucp/P
```

followed by **<esc><esc>**; this yields:

```
more /usr/lib/uucp/Permissions
```

which saves you eight keystrokes.

.profile and .kshrc

When a user of the Korn shell logs into COHERENT, **ksh** first executes the script **/etc/profile**. This sets up default environmental variables for every user on the system, such as the default **PATH** and the default **TERM** variables.

Next, **ksh** executes the script **.profile** in the user's home directory. You can customize this file to suit your preferences. For example, you can set up a customized **PATH**, define aliases, and have the shell execute some

LEXICON

programs automatically (such as **calendar** or **fortune**).

Finally, **ksh** executes the script named in the environmental variable **ENV** whenever you invoke a shell. By custom, this script is named **.kshrc** and is kept in your home directory, but you name it anything you wish. This file should define how you want the shell itself to function. In this way, you can ensure that your settings will be available to all subshells, as well as to your login shell. If you wish to hide these settings from subshells, just conclude your **.kshrc** with the command:

```
unset ENV
```

For more information, see the Lexicon articles **profile**, **.profile**, and **.kshrc**.

Example

The following C code creates a program called **splurt.c**. It demonstrates numbered redirection of **ksh**, by writing to five streams without opening them. Compile it with the command:

```
cc -o splurt splurt.c
```

To call it from the command line, you could type a command of the form:

```
splurt      3> splurt3 4> splurt4 5> splurt5 6> splurt6 7> splurt7
```

This will redirect the **splurt**'s output into files **splurt3** through **splurt7**.

```
#include <stdio.h>
main()
{
      int i;
      char buf[50];

      for(i = 3; i < 8; i++) {
            sprintf(buf, "For fd %d\n", i);
            write(i, buf, strlen(buf));
      }
}
```

Files

/etc/profile — System-wide initial commands
$HOME/.kshrc — Set up user-specific environment
$HOME/.profile — User-specific initial commands
/dev/null — For background input

See Also

bind, commands, dup(), environ, exec, fork(), getopts, jobs, kill, .kshrc, login, newgrp, profile, set, sh, signal(), test, Using COHERENT, vsh, wait

For a list of commands associated with **ksh**, see the **Shell Commands** section of the **Commands** Lexicon article.

Introduction to sh, the Bourne Shell, tutorial

Notes

Note that the queue of previously issued commands is stored in memory, not on disk.

This version of **ksh** does not support variable arrays.

The Mark Williams version of **ksh** is based on the public-domain version of the Korn shell, which in turn is based on the public-domain version of the seventh edition Bourne shell written by Charles Forsyth and modified by Eric Gisin, Ron Natalie, Arnold Robbins, Doug Gwyn, and Erik Baalbergen.

KSH_VERSION — Environmental Variable

List current version of Korn shell

The Korn shell stores its current version in environmental variable **KSH_VERSION**.

See Also

environmental variables, ksh

.kshrc — System Administration

Set personal environment for Korn shell

Whenever you invoke the Korn shell **ksh**, it executes the script named in the environmental variable **ENV**. By custom, this is usually the file **${HOME}/.kshrc**.

To ensure that **.kshrc** is executed whenever you log in, insert the line

```
export ENV=${HOME}/.kshrc
```

into your **.profile**.

.kshrc should include all items that you wish to have known to all of the shells that you invoke — both the login shell and all subshells. These should include aliases, environmental variables, and the **set** commands that you use to fine-tune the operation of the shell. If you wish to define items in your login shell but hide them from subshells, simply place them in your **.profile** instead of your **.kshrc**. For example, the command

```
set -o emacs
```

turns on MicroEMACS-style command-line editing for all of your subshells when you insert it into your **.kshrc**, but turns it on only for your login shell if you insert it only into your **.profile**.

The following gives a sample **.kshrc**:

```
# Set the main prompt (PS1) to be the machine (i.e., site) name, the
# tty name (i.e., session name) and the current directory.  The
# second-level prompt (PS2) used for multi-line commands is much simpler.
SITE=‘cat /etc/uucpname‘
TTY=‘tty | sed s/^.....///‘
PS1=’$SITE $TTY $PWD: ’
PS2=’MORE> ’

# Turn on hashing, tracking, and filename completion (-h), EMACS-like
# command-line editing, and ignore end-of-file (<ctrl-D>) as a way to
# log out.
set -h -o emacs ignoreeof

#
# Add the word "logout" as an alias for "exit".
#
alias logout=’exit’

# Add EMACS command line editing command "delete-word-backword" bound
# to the key sequence <Esc><Backspace>. Note that there are four
# characters inside the apostrophes; the shell interprets a ^
# followed by a character as meaning <Ctrl> character.
bind ’^[^H’=delete-word-backward

#  Select MicroEMACS as the default editor to use with "fc" commands
FCEDIT=emacs
```

See Also

Administering COHERENT, ENV, ksh, profile, .profile, Using COHERENT

ktty.h — Header File

Kernel portion of tty structure
#include <sys/ktty.h>

The header file **ktty.h** defines the kernel's portion of the teletypewriter (tty) structure. It also defines a set of test macros that can be used to test for specific conditions.

See Also

header files

l — Command

List directory's contents in long format
l [*file ...***]**

l is a link to the command **ls -l**. It prints the contents of *file* in long format, that is, showing its length, its owner, the date and time it was last modified, and other useful information. If a *file* is a directory, **l** lists its contents. If no *file* is named, **l** lists the contents of the current directory by default.

See Also

commands, lc, lf, lr, ls, lx

l.out.h — Header File

Format for COHERENT 286 objects
#include <l.out.h>

The header file **l.out.h** describes the **l.out** object format, which is produced by the compiler, assembler, and the linker under COHERENT 286.

The assembler outputs an unlinked object module, which must be bound with any required libraries (leaving no unresolved symbols) to produce an executable file, or *load module*. A call to one of the **exec** routines can then execute the load module directly.

The link module begins with a header, which gives global and size information about each segment. Segments of the indicated size follow the header in a fixed order. The header file **l.out.h** defines the header structure as follows:

```
struct     ldheader {
           short l_magic;
           short l_flag;
           short l_machine;
           unsigned short l_entry;
           fsize_t l_ssize[NLSEG];
};
```

l_magic is the magic number that identifies a link module; it always contains **L_MAGIC. l_flag** contains flags indicating the type of the object. **l_machine** is the processor identifier, as defined in the header file **mtype.h**. **l_tbase** is the start of the text segment. **l_entry** contains the machine address where execution of the module commences. **l_ssize** gives the size of each segment.

Files

l.out — Default load module name
<l.out.h> — Define format of COHERENT 286 objects
<mtype.h> — Machine identifiers

See Also

as, cc, core, exec, ld, libc, mtype, nm

Notes

COHERENT 386 uses the common object file format (COFF) for its executables. See the Lexicon entry for **coff.h** for information on this format.

l3tol() — General Function (libc)

Convert file system block number to long integer
l3tol(*lp, l3p, n***)**
long **lp***;*
char **l3p***;*
unsigned *n***;**

To conserve space inside i-nodes in COHERENT file systems, the system stores block addresses in three bytes. Programs that reference or maintain file systems use the functions **l3tol()** and **ltol3()** routines to convert between the three-byte representation and **long** numbers.

l3tol() converts *n* three-byte block addresses at location *l3p* to an array of **long** integers at location *lp.*

See Also

libc

LASTERROR — Environmental Variable

Program that last generated an error
LASTERROR=*program name*

The environmental variable **LASTERROR** names the last program to have returned an error to the shell. For example, if you had used the command **set** with an incorrect number of arguments, it would have failed to run and would have exited with an error condition, and **LASTERROR** would read **LASTERROR=set**.

The command **help** reads **LASTERROR** to determine what the last program was for which you needed help. Thus, if you type **help** without an argument, it will return information about the program named in **LASTERROR**.

See Also

environmental variables

.lastlogin — System Administration

Record of last login
$HOME/.lastlogin

The command **login** records in file **$HOME/.lastlogin** the date and time you last logged in. **login** displays the information the next time you log in.

If this file does not exist **login** assumes that you are a new user, and by default executes the file **/etc/default/welcome**. This provides a "friendly" environment for users who are using COHERENT for the first time.

See Also

Administering COHERENT, login, Using COHERENT

Latin 1 — Definition

Standard 8859 of the International Standards Organization (ISO) defines a set of tables of eight-bit codes for the printable characters used in various languages.

The lower seven bits of each table (i.e., values 0x20 through 0x7E) are the same as those defined in ISO Standard 646; which, in turn, are the same as those in the character set called "U.S. ASCII", which encodes the Latin alphabet, plus numerals, punctuation marks, and additional characters commonly used in the United States. Values 0xA0 through 0xFF of each table in ISO 8859 either adds additional characters used in family of languages, or maps the characters in the Latin alphabet to the characters in another alphabet. For example, the table ISO 8859.2 (also called ISO Latin 2) adds to U.S. ASCII the characters used in the languages Albanian, Czech, Slovak, Hungarian, Polish, Romanian, and Croatian; whereas the table ISO 8859.6 maps the Latin alphabet to the Arabic alphabet.

Table ISO 8859.1 is also called *ISO Latin 1*. It adds the national characters used most Romance and Germanic languages (i.e., English, German, Dutch, Flemish, Norwegian, Icelandic, Swedish, Danish, Spanish, French, Italian, and Portuguese), plus Anglo-Saxon, Irish, and Finnish.

Please note that unlike U.S. ASCII, the characters in an ISO 8859 table are *not* in their proper order for lexical sorting. The character tables for locale-specific sorting are kept elsewhere.

As mentioned above, the printable characters 0x20 (space) through 0x7E (tilde) in ISO Latin 1 are the same as they are in U.S. ASCII. The following table gives the additional printable characters that ISO Latin 1 defines, from 0xA1 through 0xFF. The table gives each character's value in octal, decimal, and hexadecimal, and its description. Note that our printer does cannot print every character, so in some cases the description must suffice.

0241	161	0xA1	¡	Inverted exclamation mark
0242	162	0xA2	¢	Cent sign
0243	163	0xA3	£	Pound sign
0244	164	0xA4	¤	Currency sign
0245	165	0xA5	¥	Yen sign
0246	166	0xA6		Broken bar
0247	167	0xA7	§	Section symbol
0250	168	0xA8	¨	Diaeresis
0251	169	0xA9	©	Copyright sign
0252	170	0xAA	ª	Feminine ordinal indicator
0253	171	0xAB	«	Left angle quotation mark
0254	172	0xAC	–	Not sign
0255	173	0xAD		Soft hyphen
0256	174	0xAE	®	Registered trade mark sign
0257	175	0xAF	¯	Macron
0260	176	0xB0	°	Ring above or degree sign
0261	177	0xB1	±	Plus-minus sign
0262	178	0xB2		Superscript two
0263	179	0xB3		Superscript three
0264	180	0xB4		Acute accent
0265	181	0xB5	µ	Micro sign (mu)
0266	182	0xB6	¶	Pilcrow (paragraph) sign
0267	183	0xB7		Middle dot
0270	184	0xB8	¸	Cedilla
0271	185	0xB9		Superscript one
0272	186	0xBA	º	Masculine ordinal indicator
0273	187	0xBB	»	Right angle quotation mark
0274	188	0xBC		Vulgar fraction one quarter
0275	189	0xBD		Vulgar fraction one half
0276	190	0xBE		Vulgar fraction three quarters
0277	191	0xBF	¿	Inverted question mark
0300	192	0xC0		A with grave accent
0301	193	0xC1		A with acute accent
0302	194	0xC2		A with circumflex accent
0303	195	0xC3		A with tilde
0304	196	0xC4		A with diaeresis
0305	197	0xC5		A with ring above
0306	198	0xC6	Æ	Diphthong A with E
0307	199	0xC7	Ç	C with cedilla
0310	200	0xC8		E with grave accent
0311	201	0xC9		E with acute accent
0312	202	0xCA		E with circumflex accent
0313	203	0xCB		E with diaeresis
0310	204	0xCC		I with grave accent
0311	205	0xCD		I with acute accent
0312	206	0xCE		I with circumflex accent
0313	207	0xCF		I with diaeresis
0320	208	0xD0		Capital eth
0321	209	0xD1		N with tilde
0322	210	0xD2		O with grave accent
0323	211	0xD3		O with acute accent
0324	212	0xD4		O with circumflex accent
0325	213	0xD5		O with tilde
0326	214	0xD6		O with diaeresis
0327	215	0xD7		Multiplication sign
0330	216	0xD8	Ø	O with oblique stroke

0331	217	0xD9		U with grave accent
0332	218	0xDA		U with acute accent
0333	219	0xDB		U with circumflex accent
0334	220	0xDC		U with diaeresis
0335	221	0xDD		Y with acute accent
0336	222	0xDE		Thorn
0337	223	0xDF	ß	Sharp s
0340	224	0xE0	à	a with grave accent
0341	225	0xE1	á	a with acute accent
0342	226	0xE2	â	a with circumflex accent
0343	227	0xE3	ã	a with tilde
0344	228	0xE4	ä	a with diaeresis
0345	229	0xE5	å	a with ring above
0346	230	0xE6	æ	Diphthong a with e
0347	231	0xE7	ç	c with cedilla
0350	232	0xE8	è	e with grave accent
0351	233	0xE9	é	e with acute accent
0352	234	0xEA	ê	e with circumflex accent
0353	235	0xEB	ë	e with diaeresis
0354	236	0xEC	ì	i with grave accent
0355	237	0xED	í	i with acute accent
0356	238	0xEE	î	i with circumflex accent
0357	239	0xEF	ï	i with diaresis
0360	240	0xF0		Small eth
0361	241	0xF1	ñ	n with tilde
0362	242	0xF2	ò	o with grave accent
0363	243	0xF3	ó	o with acute accent
0364	244	0xF4	ô	o with circumflex accent
0365	245	0xF5	õ	o with tilde
0366	246	0xF6	ö	o with diaeresis
0367	247	0xF7		Division sign
0370	248	0xF8	ø	o with oblique stroke
0371	249	0xF9	ù	u with grave accent
0372	250	0xFA	ú	u with acute accent
0373	251	0xFB	û	u with circumflex accent
0374	252	0xFC	ü	u with diaresis
0375	253	0xFD	ý	y with acute accent
0376	254	0xFE		Small thorn
0377	255	0xFF	ÿ	y with diaeresis

See Also

ASCII, Programming COHERENT

lc — Command

List directory's contents in columnar format
lc [-1abcdfp] [*directory ...***]**

lc lists the names of the files in each *directory*, or the current directory if no *directory* is named. The files are categorized by type (files, directories, and so on) and listed in columns within each category.

The following options modify the output.

-1 List only one file name per line (do not print in columns). Please note that this is the numeral one, not a lower-case el.

-a List all file names, including '.' and '..'.

-b List block-special files only.

-c List character-special files only.

-d List directories only.

-f List regular files only.

-p List pipe files only.

See Also

commands, ls

Notes

lc -lf is useful for producing a list of regular files. For example

```
cp `lc -lf` mydir
```

copies all regular files to directory **mydir**.

lcasep — Command

Convert text to lower case
lcasep [-f *inputfile***] [-o** *outputfile***]**

The command **lcasep** converts characters in its input stream to lower case. If you do not name an *inputfile* on the command line, **lcasep** reads the standard input. If you do not name an *outputfile*, it writes it output to the standard output.

See Also

commands, mail

Notes

The command **smail** uses **lcasep** to convert headers on mail messages to convert addresses to lower case. Normally, users do not run it directly.

lcong48() — Random-Number Function (libc)

Initialize values from which 48-bit random numbers are computed
long lcong48(*param***)**
unsigned short *param***[7];**

Computation of 48-bit pseudo-random numbers uses two 48-bit integers and one 16-bit integer. One of the 48-bit values holds the "seed" value from which the 48-bit pseudo-random value is computed. This seed can be set explicitly, or is the previously computed pseudo-random number. The other 48-bit integer holds the multiplier from which the pseudo-random number is computed; and the 16-bit integer gives holds the addend.

Function **lcong48()** initializes the variables used to compute 48-bit pseudo-random numbers. *param* is an array of seven unsigned short integers that hold the initializers: *param***[0]** through *param***[2]** hold the "seed"; *param***[3]** through *param***[5]** hold the multiplier; and *param***[6]** holds the addend.

lcong48() returns nothing.

See Also

libc, srand48()

ld — Command

Link relocatable object modules
ld *[option …] file …*

A compiler translates a file of source code into a *relocatable object*. This relocatable object cannot be executed by itself, for calls to routines stored in libraries have not yet been resolved. **ld** combines, or *links*, relocatable object files with routines stored in libraries produced by the archiver **ar** to construct an executable file. For this reason, **ld** is sometimes called a *linker*, a *link editor*, or a *loader*.

ld scans its arguments in order and interprets each option as described below. Each non-option argument is either a relocatable object file, produced by **cc**, **as**, or **ld**, or a library archive produced by **ar**. It rejects all other arguments and prints a diagnostic message.

Each relocatable file argument is bound into the output file if its machine type matches the machine type of the first file so bound; if it does not, **ld** prints a diagnostic message. The symbol table of the file is merged into the output symbol table and the list of defined and undefined symbols updated appropriately. If the file redefines a symbol defined in an earlier bound module, the redefinition is reported and the link continues. The last such

redefinition determines the value that the symbol will have in the output file, which may be acceptable but is probably an error.

Each library archive argument is searched only to resolve undefined references, i.e., if there are no undefined symbols, the linker goes to the next argument immediately. The library is searched from first module to last and any module that resolves one or more undefined symbols is bound into the output exactly as an explicitly named relocatable file is bound. The library is searched repeatedly until an entire scan adds nothing to the executable file.

The order of modules in a library is important in two respects: it will affect the time required to search the library, and, if more than one module resolves an undefined symbol, it can alter the set of library modules bound into the output.

A library will link faster if the undefined symbols in any given library module are resolved by a library module that comes later in the library. Thus, the low-level library modules, those with no undefined symbols, should come at the end of the library, whereas the higher-level modules, those with many undefined symbols, should come at the beginning. The library module **ranlib.sym**, which is maintained by the **ar s** modifier, provides **ld** with a compressed index to the symbols defined in the library. But even with the index, the library will link much faster if the modules occur in top-down rather than bottom-up order.

A library can be constructed to provide a type of "conditional" linking if alternate resolutions of undefined symbols are archived in a carefully thought-out order. For instance, **libc.a** contains the modules

finit.o
exit.o
_finish.o

in precisely the order given, though some other modules may intervene. **finit.o** contains most of the internals of the STDIO library, **exit.o** contains the **exit()** function, and **_finish.o** contains an empty version of **_finish()**, the function that **exit()** calls to close STDIO streams before process termination. If a program uses any STDIO routines, macros, or data, then **finit.o** will be bound into the output with its version of **finish()**. If a program uses no STDIO, then the "dummy" **_finish.o** will be bound into the output because it is the first module that defines **_finish()** that the linker encounters after **exit.o** adds the undefined reference. This saves approximately 3,000 bytes. To set the order of routines within a library, use the archiver **ar**.

COFF Linking

COHERENT uses the Common Object File Format (COFF). This format renders many advantages, but it also places special demands upon the linker. The following discussing some of the complexities that arise for linking into the COFF format.

Under COFF, common variables are kept aligned according to their most strongly aligned contributor. If *name* is linked with another module that also declares *name* but sets it to another length, the linker creates one such variable and gives it the greater length of the two. **ld** deduces the alignment of a common variable by its length: if the length of a common is divisible by four, it is aligned on a four-byte boundary; if it is divisible by two, it is aligned on a two-byte boundary. Otherwise, it is assumed to be unaligned. The linker supports only three classes of alignment: four-byte, two-byte, and unaligned. It then aligns a common variable according to its most strongly aligned contributor.

For example, if one assembly-language module contributes a **.comm** (common) variable named **xyz** whose length is four bytes, and another contributes another **xyz** whose length is five bytes, **ld** gives the resulting **xyz** a length of eight bytes to satisfy the length requirement (at least five) and the alignment requirement (four-byte boundary).

Or in another example, if you declare a C variable **char x;** x is a common variable, with a length of one byte. If another C module declares **long x;** the two **x**'s will share a length of four bytes. However, in the first module **sizeof(x) == 1** and in the second **sizeof(x) == 4**. These will cause warning messages to appear, which you can turn off by using the **-q** option.

After **ld** has made its first pass, it places all common variables at the end of the **.bss** segment: first the four-byte-aligned variables, then the two-byte-aligned, then the unaligned.

Options

ld recognizes the following options:

-e *entry*
Specify the *entry* point of the output module, either as a symbol or as an absolute octal address.

LEXICON

-f (Force) Force link even if there are errors. Results may be undefined.

-G Suppress the common/global warning — that is, tell **ld** not to complain if a global variable is also used as a common variable.

-i This option is obsolete, but is kept for compatibility purposes. If you include it in a **makefile**, **ld** will silently ignore it.

-K Link a kernel segment.

-L*directory*
Search *directory* for libraries and objects before searching the directories named in **LIBPATH**. Note that you can have more than one **-L** option in a **ld** command line. For example, if **LIBPATH** is set to **/lib:/usr/lib**, then the command line

```
ld -L/search/First -L/search/Next a.o -lxyz
```

tells **ld** to search for libraries **libxyz.a** and **libc.a** along the path:

```
/search/First:/search/Next:/lib:/usr/lib
```

The character that separates entries in the path is set by the macro **LISTSEP**. Header file **path.h** defines this to be the ':'.

-l *name* An abbreviation for the library **/lib/lib***name***.a** or **/usr/lib/lib***name***.a** if the first is not found.

-o *file* Write output to *file*. The default is **a.out**.

-q Suppress all warning messages.

-Q Suppress all error messages, not just warnings.

-r Retain relocation information in the output, and issue no diagnostic message for undefined symbols. This option builds a **.o** file that appears as if its pieces had been compiled together.

-s Strip the symbol table from the output. The same effect may be obtained by using the command **strip**. The **-s** and **-r** options are mutually exclusive.

-u *symbol*
Add *symbol* to the symbol table as a global reference, usually to force the linking of a particular library module.

-X Discard local compiler-generated symbols beginning **.L**.

-x Discard all local symbols.

ld reads the environmental variables **LDHEAD** and **LDTAIL** and appends them to, respectively, the beginning and end of its command line. For example, to ensure that **ld** is always executed with the option **-d**, insert the following into your **.profile**:

```
export LDHEAD=-d
```

Likewise, to ensure that **ld** always includes the mathematics library **libm** when it links, insert the following into your **.profile**:

```
export LDTAIL=-lm
```

LIBPATH

Except when used with its **-l** option, **ld** does not know about paths: it links exactly what you tell it to link via the **cc** command line. **cc** looks for libraries by searching the directories named in the environmental variable **LIBPATH**. If you do not define **LIBPATH** in your environment, it searches the default **LIBPATH** as defined in **/usr/include/path.h**. If you define **LIBPATH**, **cc** searches the directories in the order you specify. For example, a typical definition is:

```
export LIBPATH=:/lib:/usr/lib
```

This searches the current directory '.', then **/lib**, then **/usr/lib**.

Linker-defined Symbols

ld defines the following set of symbols within an executable program:

__end_text End of the **.text** segment
__end_data End of the **.data** segment
__end_bss End of the **.bss** segment
__end End of the highest segment

Note that if you have a segment named **.xyz**, then **ld** will allow you to use **__end_xyz**.

Files

a.out — Default output
/coherent for **-k** option
/lib/lib*.a — Libraries
/usr/lib/lib*.a — More libraries

See Also

ar, ar.h, as, cc, cdmp, coff.h, commands, l.out.h, LIBPATH, strip

Diagnostics

The following gives the error messages returned by **ld**. The messages are in alphabetical order; each is marked as to whether it is a *fatal* or *warning* condition. A fatal message usually indicates a condition that caused the compiler to terminate execution. Warning messages point out code that is compilable, but may produce trouble when the program is executed.

archive '*string*' is corrupt (fatal)
This archive makes no sense. You may wish to examine this with the archiver **ar**.

file *string*: module *string*: bad header (warning)
string does not look like a real object module.

can't find '*string*' (fatal)
ld cannot find the requested library. Make sure that the **cc** command line points to the directory that holds the archive.

cannot create '*string*' (fatal)
ld cannot create its output file.

entry point '*string*' not in .text (warning)
error reading '*string*' (fatal)

'*string*' is not a COFF archive (fatal)
All files ending **.a** should be COFF archives. You may need to rebuild this archive.

Library must be created with ar -s option (fatal)
The option **-s** to **ar** gives libraries a symbol table for the use of **ld**.

No work (fatal)
There were no object files loaded.

pass 1, *n* errors (fatal)
At the end of pass 1 there were *n* errors detected. The link stopped here.

symbol '*string*' redefined in file '*string*': module '*string*' (warning)
A symbol is defined in incompatible ways in different files.

symbol '*string*' redefined in file '*string*' (warning)
A symbol is defined in incompatible ways in different files.

file *string*: module *string*: relocation out of range *0xn* (warning)
A relocation record points outside the range of its segment.

symbol '*string*' severe warning symbol defined as a common and a global (warning)
A symbol was defined as a common, e.g.

```
int x;
```

and as a global, e.g.:

```
int x = 5;
```

There is no good way to fix this without reading the code and thinking about the variable usage. The linker turned the global into an external. That is, it turned

LEXICON

```
int x;
```
into
```
extern int x;
```
This matches the UNIX linker.

file *string*: module *string*: unknown r_type *n* in segment *n* record *n* (warning)
Unknown type on COFF relocation record.

unlikely input file name '*string*' (warning)
Input file names must end **.o** for object or **.a** for archive.

symbol '*string*' warning defined with lengths *n* and *n* (warning)
A common was defined with different lengths, while this is legal it is very unusual in C programs. This warning may be turned off with the flag **-c**.

symbol '*string*' warning, redefines builtin symbol (warning)
Some symbols such as **__end** and **__end_text** are special to the linker. In general, symbols beginning '__' are reserved to implementors and should be avoided by users. Your definition has been used.

write error (fatal)
ld cannot write the executable program. Check that you have permission to write into the target directory.

Notes

If you are linking a program by hand (that is, running **ld** independently from the **cc** command), be sure to include the appropriate run-time start-up routine with the **ld** command line; otherwise, the program will not link correctly.

ldexp() — General Function (libc)

Combine fraction and exponent
#include <math.h>
double ldexp(*f***, *e*)**
double *f*; **int** *e*;

ldexp() combines the fraction *f* with the binary exponent *e* to return a floating-point value *real* that satisfies the equation *real*=*m*×2^*e*.

See Also

atof(), ceil(), fabs(), floor(), frexp(), libc, modf()
ANSI Standard, §7.5.4.3
POSIX Standard, §8.1

LDHEAD — Environmental Variable

Append options to beginning of ld command line
export LDHEAD=*options*

The COHERENT linker **ld** reads the environmental variables **LDHEAD** and **LDTAIL** before it begins its work. You can set these variables to hold the default options that you want the linker always to use.

ld appends the options in **LDHEAD** to the beginning of its command line.

See Also

environmental variables, ld, LDTAIL

Notes

This environmental variable is included only to support existing code. Its use is deprecated, and it may not be supported in future releases of COHERENT.

ldiv() — General Function (libc)

Perform long integer division
#include <stdlib.h>
ldiv_t ldiv(*numerator***, *denominator*)**
long *numerator*, *denominator*;

ldiv() divides *numerator* by *denominator*. It returns a structure of the type **ldiv_t**, which is structured as follows:

```
typedef struct {
      long quot;
      long rem;
} ldiv_t;
```

ldiv() writes the quotient into **quot** and the remainder into **rem**.

The sign of the quotient is positive if the signs of the arguments are the same; it is negative if the signs of the arguments differ. The sign of the remainder is the same as the sign of the numerator.

If the remainder is non-zero, the magnitude of the quotient is the largest integer less than the magnitude of the algebraic quotient. This is not guaranteed by the operators **/** and **%**, which merely do what the machine implements for divide.

See Also

libc
ANSI Standard, §7.10.6.4

Notes

The ANSI Standard includes this function to permit a useful feature found in most versions of FORTRAN, where the sign of the remainder will be the same as the sign of the numerator. Also, on most machines, division produces a remainder. This allows a quotient and remainder to be returned from one machine-divide operation.

If the result of division cannot be represented (e.g., because *denominator* is set to zero), the behavior of **ldiv()** is undefined. *Caveat utilitor.*

LDTAIL — Environmental Variable

Append options to end of ld command line
export LDTAIL=*options*

The COHERENT linker **ld** reads the environmental variables **LDHEAD** and **LDTAIL** before it begins its work. You can set these variables to hold the default options that you want the linker always to use.

ld appends the options in **LDTAIL** to the end of its command line.

See Also

environmental variables, ld, LDHEAD

Notes

This environmental variable is included only to support existing code. Its use is deprecated, and it may not be supported in future releases of COHERENT.

let — Command

Evaluate an expression
let [*expression***]**

The command **let** is built into the Korn shell **ksh**. It evalutes *expression*; it returns zero if *expression* evaluates to non-zero status, and non-zero if it evaluates to zero status.

See Also

commands, ksh

lex — Command

Lexical analyzer generator
lex [-t][-v][*file***]**
cc lex.yy.c -ll

Many programs, e.g., compilers, process highly structured input according to rules. Two of the most complicated parts of such programs are *lexical analysis* and *parsing* (also called *syntax analysis*). The COHERENT system includes two powerful tools called **lex** and **yacc** to help you construct these parts of a program. **lex** converts a set of lexical rules into a lexical analyzer, and **yacc** converts a set of parsing rules into a parser.

The output of **lex** may be used directly, or may be used by a parser generated by **yacc**.

LEXICON

lex reads a specification from the given *file* (or from the standard input if none), and generates a C function called **yylex()**. **lex** writes the generated function in the file **lex.yy.c**, or on standard output if you use the **-t** option. The **-v** option prints some statistics about the generated tables.

The tutorial on **lex** that appear in this manual describes **lex** in detail. In brief, the generated function **yylex()** matches portions of its input to one pattern (sometimes called a regular expression) from a set of rules, or *context*, and executes associated C commands. Unmatched portions of the input are copied to the output stream. **yylex()** returns EOF when input has been exhausted.

lex uses the following macros that you may replace with the preprocessor directive **#undef** if you wish: **input()** (read the standard input stream), and **output(*c*)** (write the character *c* to the standard output stream). You may also replace the following functions if you wish: **main()** (main function), **error(...)** (print error messages; takes same arguments as **printf**), and **yywrap()** (handle events at the end of a file). If an action is desired on end of file, such as arranging for more input, **yywrap()** should perform it, returning zero to keep going.

A full **lex** specification has the following format:

- Macro definitions, of the form:

```
name  pattern
```

- Start condition declarations:

```
%S    NAME ...
```

- Context declarations:

```
%C    NAME ...
```

- Code to be included in the header section:

```
%{
anything
%}
<tab or space> anything
```

- Rules section delimiter (must always be present):

```
%%
```

- Code to appear at the start of **yylex()**:

```
<tab or space> anything
```

- Rules for initial context, in any of the forms:

```
rule        action;
rule        | (means use next action)
rule        {
<tab or space>    action;
<tab or space>    }
```

- For each additional context:

```
%C    NAME
...rules for this context...
```

- End of rules section delimiter:

```
%%
```

- Code to be copied verbatim, such as user provided **input(), output(), yywrap(),** or other.

lex matches the longest string possible; if two rules match the same length string, the rule specified first takes precedence. **lex** puts the matched string, or *token,* in the **char** array **yytext[]**, and sets the variable **yyleng** to its length.

Actions may use the following:

ECHO . . . . . Output the token
REJECT . . . . . Perform action for lower precedence match
BEGIN *NAME* . . . . . Set start condition to *NAME*

BEGIN 0 . . . . . Clear start condition
yyswitch(*NAME***)** . . . . Switch to context *NAME*, return current
yyswitch(0) . . . . . Switch to initial context
yynext() . . . . . Steal next character from input
yyback(*c***)**. . . . . Put character *c* back into input
yyless(*n***)** . . . . . Reduce token length to *n*, put rest back
yymore() . . . . . Append next token to this one
yylook() . . . . . Returns number of chars in input buffer

lex rules are contiguous strings of the form

[<*NAME,...*>][^] *token* [*/lookahead*][$]

where brackets '[]' indicate optional items.

<*NAME,...*> . . . . . Match only under given start conditions
^. . . . . Match the beginning of a line
$. . . . . Match the end of a line
token . . . . . Pattern that a given token is to match
/lookahead. . . . . Pattern that given trailing text is to match

Pattern elements:

a . . . . . The character **a**
\a . . . . . The character **a**, even if special
. . . . . . Any character except newline
[abx-z] . . . . . Any of **a, b,** or **x** through **z**
[^abx-z]. . . . . Any except **a, b,** or **x** through **z**
abc . . . . . The string **abc**, even if any are special
{*name*} . . . . . The macro definition *name*
(*exp***)**. . . . . The pattern *exp* (grouping operator)

Optional operators on elements:

*e***?** . . . . . Zero or one occurrence of *e*
*e** . . . . . Zero or more consecutive *e*s
*e***+** . . . . . One or more consecutive *e*s
e{*n*} . . . . . *n* (a decimal number) consecutive *e*s
e{*m,n*}. . . . . *m* through *n* consecutive *e*s

Patterns may be of the form:

e1e2. . . . . Matches the sequence *e1 e2*
e1|*e2* . . . . . Matches either *e1* or *e2*

lex recognizes the standard C escapes: **\n**, **\t**, **\r**, **\b**, **\f**, and *****ooo* (octal representation). The special characters

```
\ ( ) < > { } % * + ? [ - ] ^ / $ . |
```

must be prefixed with \ or enclosed within quotation marks (excepting " and \) to be normal. Within classes, only the characters . ^ - \ and] are special.

Files

/usr/lib/libl.a
/usr/src/libl/* — library source code

See Also

commands, yacc
Introduction to lex, the Lexical Analyzer

Lexicon — Technical Information

Format of the COHERENT manual pages

The COHERENT manual pages use a unique format, which we call "the Lexicon." Under this format, each function, each command, and each term has its own entry in the manual. All manual entries are printed together, instead of being divided into segments. The purpose of this format is to make it easy for you to find the manual entry for any given command or function. Anyone who as struggled with the multiple volumes of UNIX documentation can

appreciate this feature of the COHERENT Lexicon.

For more details on the Lexicon format, see the introduction to the Lexicon.

See Also

Administering COHERENT, Programming COHERENT, Using COHERENT

lf — Command

List directory's contents in columnar format
lf [*file ...***]**

lf is a link to the command **ls -CF**. It prints *file* in columnar format, like the command **lc**. **lf**, however, combines files and directories into one listing, with directories being indicated by a slash after the file name and executable being indicated by an asterisk. If a *file* is a directory, **l** lists its contents. If no *file* is named, **lf** lists the contents of the current directory by default.

See Also

commands, l, lc, lr, ls, lx

libc — Library

Standard C library
/lib/libc.a

libc is the library that contains most functions linked into C programs. It contains many general-purpose functions, as well as stubs for COHERENT system calls. The following summarizes these functions.

Binary Data

The following functions manipulate binary data types, that is, integers and floating-point numbers.

abs() . . . . . Return the absolute value of an integer
decvax_d() . . . . . Convert a **double** from IEEE to DECVAX format
decvax_f() . . . . . Convert a **float** from IEEE to DECVAX format
div() . . . . . Perform integer division
frexp() . . . . . Separate fraction and exponent
ieee_d() . . . . . Convert a **double** from DECVAX to IEEE format
ieee_f() . . . . . Convert a **float** from DECVAX to IEEE format
ldexp() . . . . . Combine fraction and exponent
ldiv() . . . . . Perform long integer division
modf() . . . . . Separate integral part and fraction

Binary Data and Strings

The following functions convert binary data forms to strings, or strings to binary forms.

atof() . . . . . Convert ASCII strings to floating point
atoi() . . . . . Convert ASCII strings to integers
atol() . . . . . Convert ASCII strings to long integers
ecvt() . . . . . Convert floating-point numbers to strings
fcvt() . . . . . Convert floating-point numbers to strings
gcvt() . . . . . Convert floating-point numbers to strings
strtod() . . . . . Convert string to floating-point number
strtol() . . . . . Convert string to long integer
strtoul() . . . . . Convert string to unsigned long integer

ctype Functions

The **ctype** functions test a character's *type*. Some can transform some characters into others. "ctype" is an abbreviation for "character type"; all are declared or defined in the header file **<ctype.h>**. They are as follows:

_tolower() . . . . . Convert an upper-case character to lower case
_toupper() . . . . . Convert a lower-case character to upper case
isalnum() . . . . . Test if alphanumeric character
isalpha() . . . . . Test if alphabetic character
isascii() . . . . . Test if ASCII character
iscntrl() . . . . . Test if a control character
isdigit() . . . . . Test if a numeric digit

isgraph() . . . . . Test if a graphics character
islower() . . . . . Test if lower-case character
isprint() . . . . . Test if printable character
ispunct() . . . . . Test if punctuation mark
isspace() . . . . . Test if a tab, space, or return
isupper() . . . . . Test if upper-case character
isxdigit() . . . . . Test if hexadecimal numeral
toascii() . . . . . Convert a character to ASCII
tolower() . . . . . Convert an upper-case character to lower case
toupper() . . . . . Convert a lower-case character to upper case

Files and Directories

The following functions are used to manipulate files and directories, and their names.

_getwd() . . . . . Get current working directory name
closedir() . . . . . Close a directory stream
dup2() . . . . . Duplicate a file descriptor
getcwd() . . . . . Get current working directory
mktemp() . . . . . Generate a temporary file name
opendir() . . . . . Open a directory stream
path() . . . . . Build a path name for a file
readdir() . . . . . Read a directory stream
remove() . . . . . Remove a file
rewinddir() . . . . . Rewind a directory stream
seekdir() . . . . . Reset the position within a directory stream
telldir() . . . . . Return position within a directory stream

Interprocess Communication

The following functions perform interprocess communcation.

ftok() . . . . . Generate keys for interprocess communication
msgctl() . . . . . Control message operation
msgget() . . . . . Get a message queue
msgrcv() . . . . . Receive a message
msgsnd() . . . . . Send a message
semctl() . . . . . Control semaphore operations
semget() . . . . . Get a set of semaphores
semop() . . . . . Perform semaphore operations
shmat() . . . . . Attach a shared-memory segment to a process
shmctl() . . . . . Manipulate shared memory
shmdt() . . . . . Detach a shared-memory segment from a process
shmget() . . . . . Get the shared-memory segment

Memory Management

The following functions help to manage memory.

alloca() . . . . . Dynamically allocate space on the stack
calloc() . . . . . Allocate dynamic memory
free() . . . . . Return dynamic memory to free memory pool
malloc() . . . . . Allocate dynamic memory
realloc() . . . . . Reallocate dynamic memory
sbrk() . . . . . Increase a program's data space

Passwords and Groups

The following functions manipulate the system files **/etc/group**, **/etc/password**, and **/etc/shadow**, and uses the information found therein.

endgrent() . . . . . Close group file
endpwent() . . . . . Close password file
endspent() . . . . . Close the shadow-password file
getgrent() . . . . . Get group file information
getgrgid() . . . . . Get group file information, by group id
getgrnam() . . . . . Get group file information, by group name

LEXICON

getlogin() . . . . . Get login name
getpass() . . . . . Get password with prompting
getpw() . . . . . Search password file
getpwent() . . . . . Get password file information
getpwnam() . . . . . Get password file information, by name
getpwuid() . . . . . Get password file information, by identifier
getspent() . . . . . Get a shadow-password record
getspnam() . . . . . Get a shadow-password record, by user name
initgroups() . . . . . Initialize the supplementary group-access list
setgrent() . . . . . Rewind group file
setpwent() . . . . . Rewind password file
setspent() . . . . . Rewind the shadow-password file

Processes

The following functions execute and terminate. For information on how the **exec()** functions differ, see the Lexicon entry **execution**.

_exit() . . . . . Terminate a process
abort() . . . . . End program immediately
atexit() . . . . . Register a function to be called when the program exits
ctermid() . . . . . Name the terminal device that controls the current process
execl() . . . . . Execute a load module
execle() . . . . . Execute a load module
execlp() . . . . . Execute a load module
execlpe() . . . . . Execute a load module
execv() . . . . . Execute a load module
execvp() . . . . . Execute a load module
execvpe() . . . . . Execute a load module
raise() . . . . . Let a process send a signal to itself
sleep() . . . . . Suspend execution

Random Number

libc contains the following functions for generating pseudo-random numbers:

drand48() . . . . . Return 48-bit pseudo-random number as double
erand48() . . . . . Return 48-bit pseudo-random number as double
jrand48() . . . . . Return 48-bit pseudo-random number as long integer
lcong48() . . . . . Initialize values from which 48-bit random numbers are computed
lrand48() . . . . . Return 48-bit pseudo-random number as non-negative long integer
mrand48() . . . . . Return 48-bit pseudo-random number as long integer
nrand48() . . . . . Return 48-bit pseudo-random number as non-negative long integer
rand() . . . . . Generate pseudo-random numbers
seed48() . . . . . Initialize values from which 48-bit random numbers are computed
srand() . . . . . Seed random number generator
srand48() . . . . . Seed 48-bit pseudo-random number routines

Regular Expressions

The following functions read and interpret UNIX-style regular expressions:

regcomp() . . . . . Compile a regular expression into a structure
regerror() . . . . . Return an error message from a regular-expression function
regexec() . . . . . Compare a string with a regular expression
regsub() . . . . . Use regular expression to build a string

STDIO

STDIO is an abbreviation for *standard input and output*. It refers to a set of standard library functions that accompany all C compilers and that govern input and output with peripheral devices. COHERENT includes the following STDIO routines:

clearerr() . . . . . Present status stream
fclose() . . . . . Close a file stream
fdopen() . . . . . Open a file stream for I/O
feof() . . . . . Discover a file stream's status

ferror() . . . . . Discover a file stream's status
fflush() . . . . . Flush an output buffer
fgetc(). . . . . Get a character
fgetpos() . . . . . Read the file-position indicator
fgets(). . . . . Get a string
fgetw() . . . . . Get a word
fileno() . . . . . Get a file descriptor from a **FILE** structure
fopen() . . . . . Open a file stream
fprintf(). . . . . Format and print to a file stream
fputc() . . . . . Output a character
fputs() . . . . . Output a string
fputw() . . . . . Output a word
fread() . . . . . Read a file stream
freopen() . . . . . Open a file stream
fscanf() . . . . . Format and read from a file stream
fseek() . . . . . Seek in a file stream
fsetpos() . . . . . Set the file-position indicator
ftell() . . . . . Return file pointer position
fwrite() . . . . . Write to a file stream
getc() . . . . . Get a character
getchar() . . . . . Get a character
gets() . . . . . Get a string
getw(). . . . . Get a word
pclose(). . . . . Close a pipe
popen() . . . . . Open a pipe
printf() . . . . . Print a formatted string
putc() . . . . . Output a character
putchar(). . . . . Output a character
puts() . . . . . Output a string
putw(). . . . . Output a word
rewind() . . . . . Reset a file pointer
scanf() . . . . . Format and input from standard input
setbuf() . . . . . Set alternative file-stream buffer
setvbuf() . . . . . Set alternative file-stream buffer
sprintf() . . . . . Format and print to a string
sscanf(). . . . . Format and read from a string
tmpfile() . . . . . Create a temporary file
tmpnam() . . . . . Generate a unique name for a temporary file
ungetc() . . . . . Return character to file stream
vfprintf() . . . . . Format and print to a file stream
vprintf() . . . . . Print a formatted string
vsprintf(). . . . . Format and print to a string

String Functions

The character string is a common formation in C programs. The runtime representation of a string is an array of ASCII characters that is terminated by a null character ('\0'). COHERENT uses this representation when a program contains a string constant; for example:

```
"I am a string constant"
```

The address of the first character in the string is used as the starting point of the string. A pointer to a string holds only this address. Note, too, that an array of 20 characters can hold a string of 19 (*not* 20) non-null characters; the 20th character is the null character that terminates the string.

The following routines are available to help manipulate strings. The prototypes for most are declared in the header file **string.h**:

bcmp() . . . . . Berkeley function to compare two chunks of memory
bcopy() . . . . . Berkeley function to copy memory
bzero() . . . . . Berkeley function to initialize memory to NUL
fnmatch() . . . . . Match a string with a normal expression
index() . . . . . Search string for a character; use **strchr()** instead
memccpy(). . . . . Copy a region of memory up to a set character

LEXICON

memchr() . . . . . Search a region of memory for a character
memcmp() . . . . . Compare two regions of memory
memcpy() . . . . . Copy one region of memory into another
memmove() . . . . . Copy one region of memory into another with which it overlaps
memset() . . . . . Fill a region of memory with a character
pnmatch() . . . . . Match string pattern
rindex() . . . . . Find rightmost occurrence of a character in a string
strcat() . . . . . Concatenate two strings
strcmp() . . . . . Compare two strings
strncat() . . . . . Append one string onto another
strncmp() . . . . . Compare two lengths for a set number of bytes
strcpy() . . . . . Copy a string
strncpy() . . . . . Copy a portion of a string
strcoll() . . . . . Compare two strings, using locale information
strcspn() . . . . . Return length one string excludes characters in another
strdup() . . . . . Duplicate a string
strerror() . . . . . Translate an error number into a string
strlen() . . . . . Measure a string
strpbrk() . . . . . Find first occurrence in string of character from another string
strchr() . . . . . Find leftmost occurrence of character in a string
strrchr() . . . . . Find rightmost occurrence of character in a string
strspn() . . . . . Return length one string includes character in another
strstr() . . . . . Find one string within another string
strtok() . . . . . Break a string into tokens
strxfrm() . . . . . Transform a string, using locale information

System Logs

The following functions manipulate the files **/etc/utmp** and **/usr/adm/wtmp**, which record login events on your system. The former file records every login that is still executing (i.e., the user has logged in and has not yet logged), and every past login.

endutent() . . . . . Close the logging file.
getutent() . . . . . Read the next entry from **/etc/utmp**.
getutid() . . . . . Find an entry in **/etc/utmp** by login identifier.
getutline() . . . . . Find an entry in **/etc/utmp** by login device.
pututline() . . . . . Write a record into **/etc/utmp**.
setutent() . . . . . Rewind the input stream that is reading **/etc/utmp**
utmpname() . . . . . Manipulate a file other than **/etc/utmp**.

Terminals

The following functions help you cope with terminals.

isatty() . . . . . Check if a device is a terminal
ttyname() . . . . . Identify a terminal
ttyslot() . . . . . Return a terminal's line number

Standard Time Functions

libc includes the following functions to manipulate time:

asctime() . . . . . Convert time structure to ASCII string
clock() . . . . . Get processor time
ctime() . . . . . Convert system time to an ASCII string
difftime() . . . . . Return difference between two times
gmtime() . . . . . Convert system time to calendar structure
localtime() . . . . . Convert system time to calendar structure
mktime() . . . . . Turn broken-down time into calendar time
strftime() . . . . . Format locale-specific time
tzset() . . . . . Set local time zone

System Calls

The COHERENT kernel makes many services available to the C programmer. A programmer can use a COHERENT service through a system call. **libc** includes intefaces to the following system calls:

access() . . . . . Check if file can be accessed in given mode
acct() . . . . . Enable/disable process accounting
alarm() . . . . . Set an alarm
brk() . . . . . Change size of data area
chdir() . . . . . Change working directory
chmod() . . . . . Change file protection modes
chown() . . . . . Change ownership of a file
chroot() . . . . . Change process's root directory
chsize() . . . . . Change the size of a file
close() . . . . . Close a file
creat() . . . . . Create/truncate a file
dup() . . . . . Duplicate a file descriptor
execve() . . . . . Execute a load module
exit() . . . . . Terminate a program gracefully
fcntl() . . . . . Manipulate an open file
fork() . . . . . Create a new process
fpathconf() . . . . . Get a file variable by file descriptor
fstat() . . . . . Get information about a file system
fstatfs() . . . . . Get information about a file system
ftime() . . . . . Get current system time
getdents() . . . . . Read directory entries
getegid() . . . . . Get effective group id
geteuid() . . . . . Get effective user id
getgid() . . . . . Get real group id
getgroups() . . . . . Read the supplemental group-access list
getmsg() . . . . . Get the next message from a stream
getpgrp() . . . . . Get process-group identifier
getpid() . . . . . Get process id
getppid() . . . . . Get process id of parent process
getuid() . . . . . Get real user id
gtty() . . . . . Get terminal modes
ioctl() . . . . . Device-dependent control
kill() . . . . . Send a signal to a process
link() . . . . . Create a link
lseek() . . . . . Set read/write position
mkdir() . . . . . Create a directory
mkfifo() . . . . . Create a FIFO
mknod() . . . . . Create a special file
mount() . . . . . Mount a file system
nap() . . . . . Sleep briefly
open() . . . . . Open a file
pathconf() . . . . . Get a file variable by path name
pause() . . . . . Wait for signal
pipe() . . . . . Create a pipe
poll() . . . . . Query several I/O devices
ptrace() . . . . . Trace process execution
putmsg() . . . . . Place a message onto a stream
read() . . . . . Read from a file
rename() . . . . . Rename a file
rmdir() . . . . . Remove a directory
setgid() . . . . . Set group id and user id
setgroups() . . . . . Set the supplemental group-access list
setpgid() . . . . . Set the process-group identifier
setpgrp() . . . . . Make a process a process-group leader
setsid() . . . . . Set session identifier
setuid() . . . . . Set user id
sigaction() . . . . . Perform detailed signal management
sigaddset() . . . . . Add a signal to a set of signals
sigdelset() . . . . . Delete a signal from a set
sigemptyset() . . . . . Initialize a set of signals
sigfillset() . . . . . Initialize a set of signals
sighold() . . . . . Place a signal on hold

sigignore() . . . . . Tell the system to ignore a signal
sigismember(). . . . . Check if a signal is a member of a set
signal() . . . . . Specify action to take upon receipt of a given signal
sigpause() . . . . . Pause until a given signal is received
sigpending(). . . . . Examine signals that are blocked and pending
sigprocmask(). . . . . Examine or change the signal mask
sigrelse() . . . . . Release a signal for processing
sigset() . . . . . Specify action to take upon receipt of a given signal
sigsuspend(). . . . . Install a signal mask and suspend process
stat() . . . . . Find file attributes
statfs() . . . . . Get information about a file system
stime() . . . . . Set the time
stty() . . . . . Set terminal modes
sync(). . . . . Flush system buffers
sysconf() . . . . . Get configurable system variables
sysi86(). . . . . Identify parts within Intel-based machines
time(). . . . . Get current system time
times() . . . . . Obtain process execution times
ulimit(). . . . . Get/set limits for a process
umask(). . . . . Set file creation mask
umount(). . . . . Unmount a file system
uname() . . . . . Get name and version of COHERENT
unlink(). . . . . Remove a file
ustat() . . . . . Get statistics on a file system
utime() . . . . . Change file access and modification times
wait() . . . . . Await completion of child process
waitpid() . . . . . Wait for a process to terminate
write() . . . . . Write to a file

Miscellaneous

The following functions do not fit neatly into any of the above categories.

bsearch(). . . . . Search an array
coffnlist() . . . . . Symbol table lookup
crypt() . . . . . Encryption using rotor algorithm
getenv() . . . . . Read environmental variable
getopt(). . . . . Get a command-line option
l3tol(). . . . . Convert file system block number to long integer
lockf(). . . . . Lock a file or a section of a file
longjmp(). . . . . Perform a non-local goto
ltol3(). . . . . Convert long integer to file system block number
mtype(). . . . . Return symbolic machine type
perror(). . . . . System call error messages
putenv() . . . . . Add a string to the environment
qsort() . . . . . Sort arrays in memory
setjmp() . . . . . Save machine state for non-local goto
siglongjmp(). . . . . Perform a non-local goto and restore signal mask
sigsetjmp(). . . . . Save machine state and signal mask for non-local jump
shellsort() . . . . . Sort arrays in memory
swab(). . . . . Swap a pair of bytes
system() . . . . . Pass a command to the shell for execution
tempnam(). . . . . Generate a unique name for a temporary file

See Also

libraries

Notes

You do not need to link **libc** explicitly into your programs. The command **cc** always includes it by default.

The macro **offsetof()** is not described above because it does not "live" in **libc**; however, it is a useful, general-purpose entity. For details, see its Lexicon entry.

libcurses — Library

Library of screen-handling functions

libcurses is the library that holds the **curses** screen-handling functions. With **curses**, you can perform rudimentary graphics, even on dumb terminals; the range of routines includes mapping portions of the screen, drawing pop-up windows, creating forms with fields for data entry, and highlighting portions of text.

Implementations of curses

COHERENT uses the Cornell edition of **curses**. This implementation of **curses** reads the **terminfo** data base. It uses eight-bit characters; thus, the COHERENT edition of **curses** can display characters with accents and diacritical marks. Library **libcurses** contains the functions needed to read **terminfo** capability codes; thus, to compile the program **curses_ex.c**, use the following command line:

```
cc curses_ex.c -lcurses
```

Programs that wish to use **curses** *must not* link in both **libcurses** and **libterm**; doing so will cause collisions among library routines. Rather, these programs must link in *only* **libcurses**. On the other hand, if you wish to use the functions that read **terminfo** descriptions, you must link library **libcurses** into your program, even if you are not using any **curses** routines.

If you have special terminal descriptions under **termcap**, the command **captoinfo** converts a **termcap** description into its **terminfo** analogue.

See the Lexicon entries for **termcap** and **terminfo** for more information on this rather confusing topic.

How curses Works

curses organizes the screen into a two-dimensional array of cells, one cell for every character that the device can display. It maintains in memory an image of the screen, called the *curscr*. A second image, called the *stdcur*, is manipulated by the user; when the user has finished a given manipulation, **curses** copies the changes from the *stdcur* to the *curscr*, which results in their being displayed on the physical screen. This act of copying from the *stdscr* to the *curscr* is called *refreshing* the screen. **curses** keeps track of where all changes have begun and ended between one refresh and the next; this lets it rewrite only the portions of the *curscr* that the user has changed, and so speed up rewriting of the screen.

curses records the position of a "logical cursor", which points to the position in the *stdscr* that is being manipulated by the user, and also records the position of the physical cursor. Note that the two are not necessarily identical: it is possible to manipulate the logical cursor without repositioning the physical cursor, and vice versa, depending on the task you wish to perform.

Most **curses** routines work by manipulating a **WINDOW** object. **WINDOW** is defined in the header **curses.h**.

curses defines **WINDOW** as follows:

```
#define WINDOW _win_st
struct _win_st {
        short           _cury, _curx;
        short           _maxy, _maxx;
        short           _begy, _begx;
        short           _flags;
        chtype          _attrs;
        bool            _clear;
        bool            _leave;
        bool            _scroll;
        bool            _idlok;
        bool            _use_keypad;/* 0=no, 1=yes, 2=yes/timeout */
        bool            _use_meta;/* T=use the meta key */
        bool            _nodelay;/* T=don't wait for tty input */
        chtype          **_line;
        short           _firstchar;/* First changed character in the line */
        short           *_lastchar;/* Last changed character in the line */
        short           *_numchngd;/* Number of changes made in the line */
        short           _regtop;/* Top and bottom of scrolling region */
        short           _regbottom;
};
```

Type **bool** is defined in **curses.h**; an object of this type can hold the value of true (nonzero) or false (zero).

The following describes selected **WINDOW** fields in detail.

_cury, _curx Give the Y and X positions of the logical cursor. The upper left corner of the window is, by definition, position 0,0. Note that **curses** by convention gives positions as Y/X (row/column) rather than X/Y, as is usual elsewhere.

_maxy, _maxx Width and height of the window.

_begy, _begx Position of the upper left corner of the window relative to the upper left corner of the physical screen. For example, if the window's upper left corner is five rows from the top of the screen and ten columns from the left, then **_begy** and **_begx** will be set to ten and five, respectively.

_flags One or more of the following flags, logically OR'd together:

_SUBWIN — Window is a sub-window
_ENDLINE — Right edge of window touches edge of the screen
_FULLWIN — Window fills the physical screen
_SCROLLWIN — Window touches lower right corner of physical screen
_FULLINE — Window extends across entire physical screen
_STANDOUT — Write text in reverse video
_INSL — Line has been inserted into window
_DELL — Line has been deleted from window

_ch_off Character offset.

_clear Clear the physical screen before next refresh of the screen.

_leave Do not move the physical cursor after refreshing the screen.

_scroll Enable scrolling for this window.

_y Pointer to an array of pointers to the character arrays that hold the window's text.

_firstch Pointer to an array of integers, one for each line in the window, whose value is the first character in the line to have been altered by the user. If a line has not been changed, then its corresponding entry in the array is set to **_NOCHANGE**.

_lastch Same as **_firstch**, except that it indicates the last character to have been changed on the line.

_nextp Point to next window.

_orig Point to parent window.

When **curses** is first invoked, it defines the entire screen as being one large window. The programmer has the choice of subdividing an existing window or creating new windows; when a window is subdivided, it shares the same *curscr* as its parent window, whereas a new window has its own *stdscr*.

Multiple Terminals

Some applications need to display text on more than one terminal, controlled by the same process. **curses** can handle this, even if the terminals are of different types. The following describes how **curses** output to multiple terminals.

All information about the current terminal is kept in a global variable **struct screen *SP**. Although the screen structure is hidden from the user, the C compiler will accept declarations of variables which are pointers. The user program should declare one screen pointer variable for each terminal it wishes to handle.

The function **newterm()** sets up a new terminal of the given terminal type that is accessed via file-descriptor *fp*. To use more than one terminal, call **newterm()** for each terminal and save the value returned as a reference to that terminal.

To switch to a different terminal, call **set_term()**. It returns the current contents of **SP**. Do not assign directly to **SP** because certain other global variables must also be changed.

All **curses** routines always affect the current terminal. To handle several terminals, switch to each one in turn with **set_term()**, and then access it. Each terminal must first be set up with **newterm()**, and closed down with **endwin()**.

Video Attributes

curses lets you display any combination of video attributes on any terminal. Each character position on the screen has 16 bits of information associated with it. Seven bits are the character to be displayed, leaving bits for nine video attributes. These bits are used for the following modes respectively: standout, underline, reverse video, blink, dim, bold, blank, protect, and alternate-character set. Standout is whatever highlighting works best on the terminal, and should be used by any program that does not need specific or combined attributes. Underlining, reverse video, blink, dim, and bold are the usual video attributes. Blank means that the character is displayed as a space, for security reasons. Protected and alternate character set depend on the terminal. The use of these last three bits is subject to change and not recommended.

The routines to use these attributes include **attron()**, **attroff()**, **attrset()**, **standend()**, **standout()**, **wattroff()**, **wattron()**, **wattrset()**, **wstandend()**, and **wstandout()**. All are described below.

Attributes, if given, can be any combination of **A_STANDOUT**, **A_UNDERLINE**, **A_REVERSE**, **A_BLINK**, **A_DIM**, **A_BOLD**, **A_INVIS**, **A_PROTECT**, and **A_ALTCHARSET**, OR'd together. These constants are defined in **curses.h**. If the particular terminal does not have the particular attribute or combination requested, **curses** will attempt to use some other attribute in its place. If the terminal has no highlighting, all attributes are ignored.

Function Keys

Many terminals have special keys, such as arrow keys, keys to erase the screen, insert or delete text, and keys intended for user functions. The particular sequences these terminals send differs from terminal to terminal. **curses** lets you handle these keys.

A program using function keys should turn on the keypad by calling **keypad()** at initialization. This causes special characters to be passed through to the program by the function **getch()**. These keys have constants that are defined in **curses.h**. They have values starting at 0401, so they should not be stored in a **char** variable, as significant bits will be lost.

A program that uses function keys should avoid using the **<esc>** key: most sequences start with **<esc>**, so an ambiguity will occur. **curses** sets a one-second alarm to deal with this ambiguity, which will cause delayed response to the **<esc>** key. It is a good idea to avoid **<esc>** in any case, because there is eventually pressure for nearly *any* screen-oriented program to accept arrow-key input.

Scrolling Region

Most terminals have a user-accessible scrolling region. Normally, this is set to the entire window, but the calls **setscrreg()** and **wsetscrreg()** set the scrolling region for *stdscr* or the given window to any combination of top and bottom margins. If scrolling has been enabled with **scrollok()**, scrolling takes place only within that window.

TTY Mode Functions

In addition to the save/restore routines **savetty()** and **resetty()**, **curses** contains routines for going into and out of normal tty mode.

The normal routines are **resetterm()**, which puts the terminal back in the mode it was in when curses was started, and **fixterm()**, which undoes the effects of **resetterm()**, that is, restores the "current curses mode". **endwin()** automatically calls **resetterm()**. These routines are also available at the *terminfo* level.

No-Delay Mode

curses offers the call **nodelay()**, which puts the terminal into "no-delay mode". While in no-delay mode, any call to *getch()* returns -1 if nothing is waiting to be read. This is useful for programs that require real-time behavior, where the user watches action on the screen and presses a key when he wants something to happen. For example, the cursor can be moving across the screen, and the user can press an arrow key to change direction. This mode is especially useful for games.

Portability

curses contains several routines to improve portability. Because these routines do not directly relate to terminal handling, their implementation differs from system to system, and the differences can be isolated from the user program by including them in **curses**.

Functions **erasechar()** and **killchar()** return the characters that, respectively, erase one character and kill the entire input line. The function **baudrate()** returns the current baud rate, as an integer. (For example, at 9600 baud, **baudrate()** returns the integer 9600, not the value B9600 from **<sgtty.h>**.) The routine **flushinp()** throws away all typeahead. call **resetterm()** to restore the tty modes. After the shell escape, **fixterm()** can be called to set the tty modes back to their internal settings. These calls are now required, because they perform system-dependent processing.

LEXICON

curses Routines

The following table summarizes the functions and macros that comprise the **curses** library. These routines are declared and defined in the header file **curses.h**.

addch(*ch***) char** *ch***;**
Insert a character into *stdscr*.

addstr(*str***) char** ***str***;**
Insert a string into *stdscr*.

attroff(*at***) int** *at***;**
Turn off video attributes on *stdscr*.

attron(*at***) int** *at***;**
Turn on video attributes on *stdscr*.

attrset(*at***) int** *at***;**
Set video attributes on *stdscr*.

baudrate()
Return the baud rate of the current terminal.

beep() Sound the audible bell.

box(*win, vert, hor***) WINDOW** ***win***; char** *vert, hor***;**
Draw a box. *vert* is the character used to draw the vertical lines, and *hor* is used to draw the horizontal lines. The call

```
box(win, 0, 0)
```

draws a box with solid lines. The call

```
box(win, '|', '-');
```

draws a box around window **win**, using '|' to draw the vertical lines and '-' to draw the horizontal lines. Do not use non-ASCII characters unless you are very sure of the output terminal's identity.

cbreak()
Turn on cbreak mode.

clear() Clear the *stdscr*.

clearok(*win,bf***) WINDOW** ***win***; bool** *bf***;**
Set the clear flag for window *win*. This will clear the screen at the next refresh, but not reset the window.

clrtobot()
Clear from the position of the logical cursor to the bottom of the window.

clrtoeol()
Clear from the logical cursor to the end of the line.

crmode()
Turn on control-character mode; i.e., force terminal to receive cooked input.

delch() Delete a character from *stdscr*; shift the rest of the characters on the line one position to the left.

deleteln()
Delete all of the current line; shift up the rest of the lines in the window.

delwin(*win***) WINDOW** ***win***;**
Delete window *win*.

doupdate()
Update the physical screen.

echo() Turn on both physical and logical echoing; i.e., characters are automatically inserted into the current window and onto the physical screen.

endwin()
Terminate text processing with **curses**.

erase() Erase a window; do not clear the screen.

char *erasechar()
Return the erase character of the current terminal.

flash() Execute the visual bell.

flushinp()
Flush input from the current terminal.

getch() Read a character from the terminal.

getstr(*str***) char** **str**;
Read a string from the terminal.

getyx(*win,y,x***) WINDOW** ***win*****; short** *y,x*;
Read the position of the logical cursor in *win* and store it in *y,x*. Note that this is a macro, and due to its construction the variables *y* and *x* must be integers, not pointers to integers.

idlok(*win*, *flag***) WINDOW** ***win*****; int** *flag*;
Enable insert/delete line operations for window *win*. *flag* must contain the OR'd operations you desire.

inch() Read the character pointed to by the *stdscr*'s logical cursor.

WINDOW *initscr()
Initialize **curses**. Among other things, this function initializes the global variables **LINES** and **COLS**, which give, respectively, the number of lines and the number of columns on your screen.

This is of most use under X Windows. When you change the size of an **xterm** or **xvt** window, the command

```
eval 'resize'
```

resets these variables. The next time you invoke a **curses**-based program, its size will reflect new the dimensions of window.

insch(*ch***) char** *ch*;
Insert character *ch* into the *stdscr*.

insertln()
Insert a blank line into *stdscr*, above the current line.

keypad(*win,flag***) WINDOW** ***win*****; int** *flag*;
Enable keypad-sequence mapping.

char *killchar()
Return the kill character for the current terminal.

leaveok(*win,bf***) WINDOW** ***win*****; bool** *bf*;
Set *win*->**_leave** to *bf*. If set to **TRUE**, refresh will leave the cursor after the last character changed by refresh. This makes sense if you want to minimize the commands sent to the screen and it does not matter where the cursor is.

char *longname(*termbuf, name***) char** ***termbuf, *name*;
Copy the long name for the terminal from *termbuf* into *name*.

meta(*win*, *flag***) WINDOW** ***win*****; int** *flag*;
Enable use of the **meta** key.

move(*y,x***) short** *y,x*;
Move logical cursor to position *y,x* in *stdscr*.

mvaddbytes(*y,x,da,count***) int** *y,x*; **char** ***da*****; int** *count*;
Move to position *y,x* and print *count* bytes from the string pointed to by *da*.

mvaddch(*y,x,ch***) short** *y,x*; **char** *ch*;
Move the logical cursor to position *y,x* and insert character *ch*.

mvaddstr(*y,x,str***) short** *y,x***; char** **str***;*
Move the logical cursor to position *y,x* and insert string *str*.

mvcur(*y_cur,x_cur,y_new,x_new***) int** *y_cur***,** *x_cur***,** *y_new***,** *x_new***;**
Move cursor from position *y_cur,x_cur* to position *y_new,x_new*.

mvdelch(*y,x***) short** *y,x***;**
Move to position *y,x* and delete the character found there.

mvgetch(*y,x***) short** *y,x***;**
Move to position *y,x* and get a character through *stdscr*.

mvgetstr(*y,x,str***) short** *y,x***; char** **str***;*
Move to position *y,x*, get a string through *stdscr*, and copy it into *string*.

mvinch(*y,x***) short** *y,x***;**
Move to position *y,x* and get the character found there.

mvinsch(*y,x,ch***) short** *y,x***; char** *ch***;**
Move to position *y,x* and insert a character into *stdscr*.

mvwaddbytes(*win,y,x,da,count***) WINDOW** **win***;* **int** *y,x***; char** **da***;* **int** *count***;**
Move to position *y,x* and print *count* bytes from the string pointed to by *da* into window *win*.

mvwaddch(*win,y,x,ch***) WINDOW** **win***;* **int** *y,x***; char** *ch***;**
Move to position *y,x* and insert character *ch* into window *win*.

mvwaddstr(*win,y,x,str***) WINDOW** **win***;* **short** *y,x***; char** **str***;*
Move to position *y,x* and insert string *str*.

mvwdelch(*win,y,x***) WINDOW** **win***;* **int** *y,x***;**
Move to position *y,x* and delete character *ch* from window *win*.

mvwgetch(*win,y,x***) WINDOW** **win***;* **short** *y,x***;**
Move to position *y,x* and get a character.

mvwgetstr(*win,y,x,str***) WINDOW** **win***;* **short** *y,x***; char** **str***;*
Move to position *y,x*, get a string, and write it into *str*.

mvwin(*win,y,x***) WINDOW** **win***;* **int** *y,x***;**
Move window *win* to position *y,x*.

mvwinch(*win,y,x***) WINDOW** **win***;* **short** *y,x***;**
Move to position *y,x* and get character found there.

mvwinsch(*win,y,x,ch***) WINDOW** **win***;* **short** *y,x***; char** *ch***;**
Move to position *y,x* and insert character *ch* there.

struct screen ***newterm(***type***,** *fd***) char** **type***;* **int** *fd***;**
Initialize the new terminal *type*, which is accessed via file-descriptor *fd*.

WINDOW ***newwin(***lines***,** *cols***,** *y1***,** *x1***)**
int *lines***,** *cols***,** *y1***,** *x1***;**
Create a new window. The new window is *lines* lines high, *cols* columns wide, with the upper-left corner at position *y1,x1*. It returns a pointer to the **WINDOW** structure that defines the newly created window.

nl() Turn on newline mode; i.e., force terminal to output **<newline>** after **<linefeed>**.

nocbreak()
Turn off cbreak mode.

nocrmode()
Turn off control-character mode; i.e., force terminal to accept raw input.

nodelay(*win***,** *flag***) WINDOW** **win***;* **int** *flag***;**
Make **getch()** non-blocking.

noecho()
Turn off echo mode.

nonl() Turn off newline mode.

noraw()
Turn off raw mode.

overlay(*win1,win2***) WINDOW** ***win1***, *win2***;**
Copy all characters, except spaces, from their current positions in *win1* to identical positions in *win2*.

overwrite(*win1,win2***) WINDOW** ***win1***, *win2***;**
Copy all characters, including spaces, from *win1* to their identical positions in *win2*.

printw(*format[,arg1,...argN]***) char** ***format***; [***data type***] *arg1,..argN***;**
Print formatted text on the standard screen.

raw() Turn on raw mode; i.e., kernel does not process what is typed at the keyboard, but passes it directly to **curses**. In normal (or *cooked*) mode, the kernel intercepts and processes the control characters <*ctrl-C*>, <*ctrl-S*>, <*ctrl-Q*>, and **<ctrl-Y>**. See the entry for **stty** for more information.

refresh()
Copy the contents of *stdscr* to the physical screen.

resetty()
Reset the terminal flags to values stored by earlier call to **savetty**.

saveterm()
Save the current state of the terminal.

savetty()
Save the current terminal settings.

scanw(*format[,arg1,...argN]***) char** ***format***; [***data type***] *arg1,..argN***;**
Read the standard input; translate what is read into the appropriate data type.

scroll(*win***) WINDOW** ***win***;**
Scroll *win* up by one line.

scrollok(*win,bf***) WINDOW** ***win***; bool** *bf***;**
Permit or forbid scrolling of window *win*, depending upon whether *bf* is set to true or false.

setscrreg(*top***,** *bottom***) int** *top***, bottom;**
Set the scrolling region on *stdscr*.

setterm(*name***) char** ***name***;**
Set term variables for *name*.

struct screen *set_term(*new***) struct screen** ***new***;**
Switch output to terminal *new*. It returns a pointer to the previously used terminal.

standend()
Turn off standout mode.

standout()
Turn on standout mode for text. Usually, this means that text will be displayed in reverse video.

WINDOW *subwin(*win***,** *lines***,** *cols***,** *y1***,** *x1***)**
int *win,lines,cols,y1,x1***;**
Create a sub-window in window *win*. The new sub-window is *lines* lines high, *cols* columns wide, and is fixed at position *y1,x1*. Note that the position is relative to the upper-left corner of the physical screen. This function returns a pointer to the **WINDOW** structure that defines the newly created sub-window.

touchwin(*win***) WINDOW** ***win***;**
Copy all characters in window *win* to the screen.

traceoff()
Turn off debugging output.

traceon()
Turn on debugging output

LEXICON

unctrl(*ch***) char** *ch*;
Output a printable version of the control-character *ch*.

waddch(*win,ch***) WINDOW** **win*; **char** *ch*;
Add character *ch* to window *win*.

waddstr(*win,str***) WINDOW** **win*; **char** **str*;
Add the string pointed to by *str* to window *win*.

wattroff(*win,att***) WINDOW** **win*; **int** *att*;
Turn off video attributes *att* for the window pointed to by *win*.

wattron(*win,att***) WINDOW** **win*; **int** *att*;
Turn on video attributes *att* for the window pointed to by *win*.

wattrset(*win,at***) WINDOW** **win*; **int** *att*;
Set the video attributes *att* for the window pointed to by *win*.

wclear(*win***) WINDOW** **win*;
Clear window *win*. Move cursor to position 0,0 and set the screen's clear flag.

wclrtobot(*win***) WINDOW** **win*;
Clear window *win* from current position to the bottom.

wclrtoeol(*win***) WINDOW** **win*;
Clear window *win* from the current position to the end of the line.

wdelch(*win***) WINDOW** **win*;
Delete the character at the current position in window *win*; shift all remaining characters to the right of the current position one position left.

wdeleteln(*win***) WINDOW** **win*;
Delete the current line and shift all lines below it one line up.

werase(*win***) WINDOW** **win*;
Clear window *win*. Move the cursor to position 0,0 but do not set the screen's clear flag.

wgetch(*win***) WINDOW** **win*;
Read one character from the standard input.

wgetstr(*win,str***) WINDOW** **win*; **char** **str*;
Read a string from the standard input; write it in the area pointed to by *str*.

winch(*win***) WINDOW** **win*;
Force the next call to **refresh()** to rewrite the entire screen.

winsch(*win,ch***) WINDOW** **win*; **char** *ch*;
Insert character *ch* into window *win* at the current position. Shift all existing characters one position to the right.

winsertln(*win***) WINDOW** **win*;
Insert a blank line into window *win* at the current position. Move all lines down by one position.

wmove(*win,y,x***) WINDOW** **win*; **int** *y*, *x*;
Move current position in the window *win* to position *y,x*.

wnoutrefresh(*win***) WINDOW** **win*;
Copy the window pointed to by *win* to the virtual screen; do not refresh the real screen.

wprintw(*win,format[,arg1,...argN]***)**
WINDOW **win*; **char** **format*;
[*data type***] arg1,..argN;**
Format text and print it to the current position in window *win*.

wrefresh(*win***) WINDOW** **win*;
Refresh a window.

wscanw(*win,format[,arg1,...argN]***)**

WINDOW *win**; char** *format**;**
[*data type***]** *arg1,..argN**;***
Read standard input from the current position in window *win*, format it, and store it in the indicated places.

wstandend(*win***) WINDOW** *win**;**
Turn off standout (reverse video) mode for window *win*.

wstandout(*win***) WINDOW** *win**;**
Turn on standout (reverse video) mode for window *win*.

wsetscrreg(*win***,***top***,***bottom***)WINDOW** *win**;* **int** *top**, bottom**;***
Set the scrolling region on the window pointed to by *win*.

Color Support

Beginning with release 4.2, COHERENT's implementation of **curses** supports color. **curses** defines colors as a video attribute, like any other. It actually handles pairs of colors — one for the foreground and one for the background. You must initialize a color pair and given it a unique identifying number; then pass the identifier of the color pair to the function **wattron()** to turn on, like any other video attribute.

The header file **<terminfo.h>** defines the following colors, which **curses** recognizes:

COLOR_BLACK
COLOR_RED
COLOR_GREEN
COLOR_YELLOW
COLOR_BLUE
COLOR_MAGENTA
COLOR_CYAN
COLOR_WHITE

Header file **<curses.h>** defines the variables **COLORS**, which holds the maximum number of colors that your console or terminal recognizes; and **COLOR_PAIRS**, which holds the maximum number of color pairs that your console or terminal recognizes. The function **start_color()** initializes both variables.

The following gives the functions and macros with which you can manipulate colors:

can_change_colors()
This function returns TRUE if you can change the definition of a color on your device, and FALSE if you cannot. You should call this function before you invoke the function **init_color()**, described below.

color_content(*color***,***r***,***g***,***b***); int** *color***,*****r***,*****g***,*****b**;*
Read the RGB settings for *color* and write them at the addresses given by *r*, *g*, and *b*.

COLOR_PAIR(*pairnum***); int** *pairnum**;*
Return the definition of the color pair identified by *pairnum*. The color pair must have been initialized by a call to **init_pair()**.

has_colors()
Return TRUE if your console or terminal supports color and FALSE if it does not.

init_color(*color***,***r***,***g***,***b***); int** *color***,***r***,***g***,***b**;*
Initialize *color* to the RGB values *r*, *g*, and *b*. *color* must be greater than zero and less than **COLORS**. *r*,' *g*, and *b* must each be between zero and 1,000. Not every console or terminal permits you to reset its colors. Call **can_change_colors()** to see if you can alter your device's colors.

init_pair(*pairnum***,***fc***,***bc***) int** *pairnum**, fc**, bc**;*
Initialize the color pair *pairnum* to the foreground color *fc* and the background color *bc*. *pairnum* must be greater than zero and less than **COLOR_PAIRS**. *fc* and *bc* must be greater than -1 and less than **COLORS**.

pair_content(*pairnum***,***fc***,***bc***) int** *pairnum***,*****fc***,*****bc**;*
Read the foreground and background colors represented by color pair *pairnum* and write them into the areas pointed to by *fg* and *bg*.

start_color()
Turn on color processing. This function must precede all other color routines; usually, it immediately follows the function **initscr()**.

LEXICON

A brief example of how to colors appears in the **Examples** section, below.

terminfo Routines

As noted above, **curses** reads terminal descriptions from **terminfo** rather than **termcap**. The library **libcurses** also holds the following functions, with which you can read a **terminfo** description:

fixterm() . . . . . Set the terminal into program mode
putp() . . . . . Write a string into *stdwin*
resetterm() . . . . . Reset the terminal into a saved mode
setupterm() . . . . . Initialize terminal capabilities
tparm() . . . . . Output a parameterized string
tputs() . . . . . Process a capability string
vidattr() . . . . . Set the terminal's video attributes
vidputs() . . . . . Set video attributes into a function

For more information on these routines, see the Lexicon entry **terminfo**, or see each routine's entry in the Lexicon.

If you define the environment variable **TERMINFO**, **curses** checks for the terminal definition in the directory that **TERMINFO** names rather than in the standard directory **/usr/lib/terminfo**. For example, if you set the environmental variable **TERM** is set to **vt100**, then the compiled **terminfo** definition is kept in directory **/usr/lib/terminfo/v/vt100**. However, if you define **TERMINFO** to be **$HOME/testterm**, **curses** first checks **$HOME/testterm/v/vt100**; if that fails, it then checks **/usr/lib/terminfo/v/vt100**. This is useful when you are debugging a **terminfo** entry.

Structure of a curses Program

To use the **curses** routines, a program must include the header file **curses.h**, which declares and defines the functions and macros that comprise the **curses** library.

Before a program can perform any screen operations, it must call the function **initscr()** to initialize the **curses** environment.

As noted above, **curses** manipulates text in a copy of the screen that it maintains in memory. After a program has manipulated text, it must call **refresh()** to copy these alterations from memory to the physical screen. (This is done because writing to the screen is slow; this scheme permits mass alterations to be made to copy in memory, then written to the screen in a batch.)

Finally, when the program has finished working with **curses**, it must call the function **endwin()**. This frees memory allocated by **curses**, and generally closes down the **curses** environment gracefully.

Examples

The first example shows what modes and characters are visible on your system.

```
#include <curses.h>
#define A_ETX 0x03

main()
{
      int c, y = 0, x = 0, attr = A_NORMAL, mask, state = 0, hibit = 0;

      initscr();
      noecho();
      raw();

      erase();
      move(y, 0);
      addstr(
"+ sets - clears Normal Bold Underline blInk Reverse Standout Altmode");
      move(++y, 0);
      refresh();

      for (;;) {
            if (!state) {
                  switch (c = getch()) {
                  case '+':
                        state = 1;
                        break;
```

```
		case '-':
			state = 2;
			break;

		case '\b':
			if (!x)
				break;
			move(y, --x);
			addch(' ');
			move(y, x);
			refresh();
			break;

		case '\r':
		case '\n':
			move(++y, x = 0);
			refresh();
			break;

		case A_ETX:
			noraw();
			echo();
			endwin();
			exit(0);

		default:
			x++;
			addch(c | hibit);
			refresh();
		}
	} else {
		switch (c = getch()) {
		case 'A':   /* turn on high bit of input */
			hibit = (state & 1) << 7;
			state = 0;
			continue;

		case 'B':
			mask = A_BOLD;
			break;

		case 'U':
			mask = A_UNDERLINE;
			break;

		case 'I':
			mask = A_BLINK;
			break;

		case 'R':
			mask = A_REVERSE;
			break;

		case 'S':
			mask = A_STANDOUT;
			break;

		case 'N':   /* normal is an absence of bits */
			if (state == 1) {
				attr = A_NORMAL;
				hibit = state = 0;
				continue;
			}

		default:
			beep();
			continue;
		}
```

```
                if (state == 1)
                      attr |= mask;
                else
                      attr &= ~mask;
                attrset(attr);
                state = 0;
          }
      }
}
```

The next example demonstrates how to use colors in a **curses** program. It selects colors randomly, builds color pairs, and displays a 40-character text string to demonstrate the newly build color pair.

```
#include <curses.h>
#include <stdlib.h>

void goodbye()
{
      move(23, 0);
      noraw();
      echo();
      endwin();
      exit(EXIT_SUCCESS);
}

main()
{
      int x, y, i;

      initscr();
      start_color();
      noecho();
      raw();

      srand(time(NULL));
      erase();
      if (!has_colors())
            goodbye();

      for (x = 0, y = 0, i = 1; i < COLOR_PAIRS; i++, y++) {
            if (y == 23) {
                  x = 40;
                  y = 0;
            }

            move(y, x);
            init_pair(i, (rand() % COLORS), (rand() % COLORS));
            attrset(COLOR_PAIR(i));
            addstr("Pack my box with five dozen liquor jugs.");
      }

      refresh();
      goodbye();
}
```

The next example shows how to read function keys under **curses**:

```
#include <curses.h>
void text();

main()
{
      int input;

      initscr();
      raw();
      noecho();
      keypad(stdscr, TRUE);
      refresh();
```

```
      while (TRUE) {
            input = wgetch(stdscr);
            switch (input) {
            case 'q':
            case 'Q':
                  endwin();
                  exit(0);

            case KEY_UP:
                  text("cursor up");
                  break;

            case KEY_DOWN:
                  text("cursor down");
                  break;

            case KEY_LEFT:
                  text("cursor left");
                  break;

            case KEY_RIGHT:
                  text("cursor right");
                  break;

            case KEY_F(1):
                  text("function key 1");
                  break;

            case KEY_F(2):
                  text("function key 2");
                  break;

            default:
                  text("some other key");
                  break;
            }
      }
}

void text(s)
register char *s;
{
      move(0, 0);
      clrtoeol();
      printw("Your input was: %s", s);
      refresh();
}
```

See Also

curses.h, libraries, termcap, terminfo
Strang J: *Programming with curses.* Sebastopol, Calif, O'Reilly & Associates Inc., 1986.

Notes

The implementation of **curses** used by the COHERENT system was written by Pavel Curtis of Cornell University. It was ported to COHERENT by Udo Munk.

libedit — Library

Routines to gather and edit user input
/usr/lib/libedit.a

libedit.a is a library of routines that implement a simple tool for entering and editing lines of data. The includes two routines that can be called from a user-level program:

readline()
Read a line of input from the standard input, and let the user edit it.

add_history()
Add a line of edited input into a history buffer, from which the user can retrieve it for further editing and use.

LEXICON

Each is described in its own Lexicon entry.

These routines implement a simple, EMACS-like line-editing interface, much like the one available under the Korn shell **ksh**. To include these routines in an application, just call them as you would any other library function, and link the library **libedit.a** into the executable.

See Also

add_history(), libraries, readline()

Notes

libedit was written by Simmule R. Turner <uunet.uu.net!capitol!sysgo!simmy> and Rich Salz <rsalz@osf.org>.

libgdbm — Library

Library for GNU DBM functions
/usr/lib/libgdbm.a

Archive **libgdbm** contains GNU data-base management (DBM) library of functions. These functions implement a version of the standard UNIX DBM functions, which let you create and manipulate a simple hashed data base.

What is a Hashed Data Base?

A hashed data base consists of a set of records. Each record, in turn, has two elements: a *key* that uniquely identifies the record, and a mass of *data*. For example, if you were creating a data base of the persons who have accounts on your system, the key would be the user's login identifier (because that must be unique), and the data could be the user's full name.

When the GDBM routines add a record to a data base, they do the following:

1. They write the record within a file.
2. They note the size of the record, and its offset within the file.
3. They "hash" each key into a unique number, then write that hash index within a separate index file, along with the offset and size of its associated record. (For a further explanation of hashing and an example implementation, see the Lexicon entry for **strtoul()**.)

By indexing a text file in this manner, a program can find a record much more quickly than it could by simply reading the file from beginning to end. For example, when you mail a message via the mail program **smail**, that program reads a set of aliases to ensure that the message is sent to the right person. If the aliases were kept in a text file, **smail** would have to open the file and read its entire contents every time you sent a mail message; and on a busy system that has a large number of aliases, this can cause a noticeable delay in dispatching a message. By keeping its aliases within a hashed index data base, **smail** greatly reduces the time needed to look up an alias, and so speeds the dispatching of your mail.

Sets of Routines

Library **libgdbm** contains three sets of functions.

- The "GNU DBM" (GDBM) routines. Each of these functions has the prefix **gdbm_**, and is declared in the header file **<gdbm.h>**.
- DBM routines. These re-implement the original UNIX DBM routines. They are declared in header file **<dbm.h>**.
- "New DBM" (NDBM) routines. These re-implement the extended version of the UNIX DBM routines. Each of these functions has the prefix **dbm_**, and is declared in the header file **<ndbm.h>**.

Each set implements a hashed data base, but each has a different provenance, and somewhat different properties and syntax. This library includes all three sets to support the widest possible range of third-party software. If you are writing new software, however, we urge you to use the GDBM routines.

Note that you cannot mix routines from the three sets — you must pick one set, and stick with it. Please note, too, that although this library re-creates the DBM and NDBM sets of routines with regard their calling conventions and return values, internally these re-creations work somewhat different than the UNIX originals; thus, you cannot expect programs compiled with these routines to read binary data bases created by the UNIX originals.

GDBM Routines

The following summarizes the GDBM routines:

gdbm_close() . . . . . Close a GDBM data base
gdbm_delete(). . . . . Delete a record from a GDBM data base
gdbm_exists() . . . . . Check whether a GDBM data base contains a given record
gdbm_fetch() . . . . . Retrieve a record from a GDBM data base
gdbm_firstkey(). . . . . Return the first record from a GDBM data base
gdbm_nextkey() . . . . Return the next record from a GDBM data base
gdbm_open(). . . . . Open a GDBM data base
gdbm_reorganize() . . . Reorganize a GDBM data base
gdbm_setopt(). . . . . Set GDBM options
gdbm_store() . . . . . Add records to a GDBM data base
gdbm_strerror(). . . . . Translate a GDBM error code into text
gdbm_sync(). . . . . Flush buffered GDBM data into its data base

As noted above, these routines are declared in header file **<gdbm.h>**. This header file also defines two structures that the GDBM routines use. The first, **datum**, defines the structure of a data element, either a key or its associated data set:

```
typedef struct {
      char *dptr;
      int dsize;
} datum;
```

This structure lets you have a key and a data element of unlimited length. This is a departure from the orthodox UNIX DBM functions, in which the sizes of the key and the datum are both static.

The other structure, **GDBM_FILE**, holds the information that the GDBM routines use to access a GDBM data base:

```
typedef struct {int dummy[10];} *GDBM_FILE;
```

Error codes are written into global variable **gdbm_errno**, and are defined in header file **<gdbmerrno.h>**.

DBM Routines

The following summarizes the DBM routines:

dbmclose(). . . . . Close a DBM data base
dbminit(). . . . . Open a DBM data base
delete() . . . . . Delete a record from a DBM data base
fetch() . . . . . Fetch a record from a DBM data base
firstkey(). . . . . Retrieve the first record from a DBM data base
nextkey(). . . . . Retrieve the next record from a DBM data base
store() . . . . . Write a record into a DBM data base

As noted above, these routines are declared in header file **<dbm.h>**. It also defines the structure **datum**, which holds a data element, either a key or its associated data set:

```
typedef struct {
      char *dptr;
      int dsize;
} datum;
```

The sizes of the key and its datum together cannot exceed **BSIZE** bytes — that is, the size of one file-system block. **BSIZE** is defined in header file **<sys/buf.h>**; at present, it equals 512 bytes.

Please note that the function **dbmclose()** is non-standard. Programs that use it cannot be recompiled on an orthodox UNIX system.

NDBM Routines

The following summarizes the NDBM routines:

dbm_close() . . . . . Close an NDBM data base
dbm_delete() . . . . . Delete records from an NDBM data base
dbm_dirfno() . . . . . Return the file descriptor for an NDBM **.dir** file
dbm_fetch() . . . . . Fetch a record from an NDBM data base
dbm_firstkey() . . . . . Retrieve the first key from an NDBM data base

LEXICON

dbm_nextkey() . . . . . Retrieve the next key from an NDBM data base
dbm_open() . . . . . Open an NDBM data base
dbm_pagfno() . . . . . Return the file descriptor for an NDBM **.pag** file
dbm_rdonly() . . . . . Set an NDBM data base into read-only mode
dbm_store() . . . . . Store a record into an NDBM data base

As noted above, these routines are declared in header file **<ndbm.h>**. This header file also defines two structures that the NDBM routines use. The first, **datum**, defines the structure of a data element, either a key or its associated data set:

```
typedef struct {
      char *dptr;
      int dsize;
} datum;
```

This structure lets you have a key and a data element of unlimited length.

The other structure, **DBM**, holds the information that the NDBM routines use to access a NDBM data base:

```
typedef struct {int dummy[10];} DBM;
```

See Also

libraries, Programming COHERENT

Notes

libgdbm was written by Philip A. Nelson of the Computer Science Department, Western Washington University, Bellingham (phil@cs.wwu.edu). This Lexicon entry is based on an **info** file written by Pierre Gaumond. **libgdbm** and its documentation are copyright © 1989-1993 by the Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

For a full statement of the rights and obligations attached to **libgdbm**, see the file **COPYING** that accompanies the source code to this library.

libm — Library

COHERENT mathematics library
/lib/libm.a

The COHERENT mathematics library **libm** contains the following useful mathematics functions:

acos() . . . . . Calculate inverse cosine
asin() . . . . . Calculate inverse sine
atan() . . . . . Calculate inverse tangent
atan2() . . . . . Calculate inverse tangent of quotient
cabs() . . . . . Calculate complex absolute value
ceil() . . . . . Set numeric ceiling
cos(). . . . . Calculate cosine
cosh() . . . . . Calculate hyperbolic cosine
exp() . . . . . Calculate exponent
fabs() . . . . . Calculate absolute value function
floor() . . . . . Calculate floor function
fmod() . . . . . Calculate modulus for floating-point number
hypot() . . . . . Calculate hypotenuse
j0() . . . . . Calculate Bessel function, order 0
j1() . . . . . Calculate Bessel function, order 1
jn() . . . . . Calculate Bessel function, order *n*
log() . . . . . Calculate natural logarithm
log10() . . . . . Calculate common logarithm
pow() . . . . . Calculate power
sin() . . . . . Calculate sine
sinh() . . . . . Calculate hyperbolic sine

sqrt() . . . . . Calculate square root
tan(). . . . . Calculate tangent
tanh(). . . . . Calculate hyperbolic tangent

See Also

libmp, libraries, math.h

Hart, J.F., et al.: *Computer Approximations.* New York, John Wiley & Sons, 1968.

Press, W.H., Flannery, B.P., Teukolsky, S.A., Vetterling, W.T.: *Numerical Recipes in C.* New York, Cambridge University Press, 1988. *Highly recommended.*

Notes

When programs that contain mathematics routines are compiled, you must explicitly name the mathematics library on the **cc** command line. For example, to compile the example presented under the entry for **acos()**, use the following **cc** command line:

```
cc -f acos.c -lm
```

The **-f** option links in the floating point routines for **printf()**, while the **-lm** option links in the mathematics libraries. Note that the **-lm** option must come *last* on the **cc** command line, or the library will not be searched properly.

The related library **libmp** performs multi-precision arithmetic. With these routines, you can perform arithmetic on extremely large numbers, to an extremely fine precision. For details, see the Lexicon entry for **libmp**.

libmisc — Library

Library of miscellaneous functions

libmisc is a library of miscellaneous C functions. These functions perform such useful tasks as handling such programming tasks as allocation of memory, copying strings, displaying variables from C with COBOL-like "picture" descriptions, and supporting virtual arrays via secondary storage.

Source code for **libmisc** is kept in the compressed **tar** archive **/usr/src/misc.tar.Z**. To extract the files into a new subdirectory called **misc**, type the command:

```
zcat /usr/src/misc.tar.Z | tar xvf -
```

To build the library, type the following:

```
cd misc
make
```

This compiles the **libmisc** routines and builds the library **libmisc.a**.

Archive **misc.tar** also includes the header file **misc.h** which protypes these functions, and declares the global variables and constants they use. You must include this header file in any program that uses any of the **libmisc** functions.

Functions

The following summarizes the functions in **libmisc.a**:

char * **alloc(***n***) unsigned** *n***;**
malloc() *n* bytes and initialize them to zero. Abort on failure.

int approx(*a***,** *b***) double** *a***,** *b***;**
If *a* and *b* are within epsilon, return one; otherwise, return zero. epsilon is a visable **double**.

char ***ask(***reply***,** *msg***, ...) char** ****reply***, ****msg***;**
Print a message and retrieve the user's reply. *msg* is a **printf()**-style format string that formats the text pointed to by any trailing arguments. **ask()** constructs the prompt message from *msg* and prints it on the standard output; then reads a line from stdin, stores it in the place pointed to by *reply*, and returns its address. *reply* must point to enough space to hold the user's reply.

For example,

```
sscanf(ask(buff, "%d numbers", 3), &a, &b, &c);
```

prints the message

```
Enter: 3 numbers
```

writes the user's reply into **buff**, and hands its address to **sscanf()**.

void banner(*word***,** *pad***) char** ***word*****; int** *pad***;**
Print *word* on stdout as a banner, preceded by *pad* spaces. Each letter of the banner is fashioned from many occurrences of itself. This is especially useful if you wish your listings to look like truly professional, mainframe printouts.

bedaemon()
bedaemon() turns the calling program into a daemon. A *daemon* is a process that executes in the background, without the usual connections to standard I/O streams and terminals. Examples are **cron** and **uuxqt**. To ensure proper operation in connection with other system software, any program that you intend to run as a daemon should call **bedaemon()** as its first step. This call closes all inherited, open-file descriptors, detaches the process from its inherited process group and controlling terminal, sets current directory to '/', and sets **umask** to zero. For further information on daemon processes, see *Unix Network Programming* by W. Richard Stevens (Englewood Cliffs, NJ, Prentice-Hall Inc, 1990), §2.6.

unsigned short crc16(*p***) char** ***p*****;**
Compute the 16-bit cyclic redundency check (crc16) of the string pointed to by *p*, and return it. This function is very useful for building hash tables or checking differences between strings.

void fatal(*msg***, ...) char** ***msg*****;**
Print an error message and call **exit(1)**. *msg* is a **printf()**-style format string; trailing arguments must to point to data.

char ***getline(***ifp***,** *lineno***) FILE** ***ifp*****; int** ***lineno*****;**
Get a line from the input file pointed to by *ifp*. This function returns the address of the line, or NULL to indicate the end of file. **getline()** calls **malloc()** to acquire space for the line, and allows lines to be continued with a \-whitespace. It also implements **lineno**.

getline() recognizes the following escape sequences:

#	to end of line is passed
****	whitespace through end of line is passed
\n	newline
\p	literal '#'
\a	alarm
\b	backspace
\r	carrage return
\f	form feed
\t	tab
****	backslash
*****ddd*	octal number

All other \ sequences are errors that **getline()** reports on stderr.

tm_t ***jday_to_tm(***jd***) jday_t** *jd***;**
Turn a Julian date to **tm** (time) structure. The Julian date is the number of days since the beginning of the Julian calendar, January 1, 4713 B.C.; it is a good way to store dates in a system-independent manner, such as in a data base. Structure **jday_t** is defined in **misc.h**. Structure **tm** is defined in **<time.h>**.

time_t jday_to_time(*jd***) jday_t** *jd***;**
Turn Julian date structure to COHERENT time. Type **time_t** is defined in header file **<sys/types.h>**.

void splitter(*ofp***,** *line***,** *limit***) FILE** ***ofp*****; char** ***line*****; int** *limit***;**
Write *line* into file *ofp*, splitting it into chunks less than *limit* bytes long. **splitter()** inserts a \ between chunks, and attempts to do this on white-space boundaries. **splitter()** produces a long line rather than split on non-whitespace. If *line* does not end in a newline, **splitter()** adds one. This is the inverse of **getline()**.

int is_fs(*special***) char** ***special*****;**
Check whether *special* names a well-formed file system. Users should never put file systems on **/dev/ram1**, but for multi-system software, like **compress**, it is smart to test.

is_fs() returns -1 if *special* is not a device, or if **open()**, **read()**, or **seek()** fails. It returns zero if no file system was found, or one if *special* names a legal file system.

char *lcase(*st***) char ***str***;**
Convert every character in *str* to lower case. Note that this works only with the U.S. dialect of English; it does not work with German or other languages that use characters in the upper half of the ASCII table.

char *match(*string***, *** pattern***, *** fin***) char ***string***, ***pattern***, **** fin***;**
match() resembles **pnmatch()**, except that it returns the address of the pattern matched. *fin* is aimed past the end of the pattern found; that is, **match()** finds a pattern and tells you where it is.

char *metaphone(*word***) char ***word***;**
Translate *word* into a short phonetic equivalent for easy lookup. It resembles Knuth's **soundex** method, except that it uses a superior algorithm.

char *newcpy(*str***) char ***str***;**
Create a NUL-terminated copy of *str* and return its address. Call **fatal()** if there is no space.

char *pathn(*name***, *** envpath***, *** deflpath***, *** access***)**
char ***name*, ***envpath***, ***deflpath***, ***access***;**
pathn() looks for file *name*. It searches the directories named in the environmental variable *envpath*. If the user has not set *envpath*, or if it is NULL, **pathn()** searches the default path *deflpath*. *name* must have *access* permission. **pathn()** returns the full path to the file found. For example:

```
pathn("helpfile", "LIBPATH", "/lib", "r")
```

searches the directories named in **LIBPATH** for file **helpfile**, for which the user must have read permission. If **LIBPATH** is not set, **pathn()** searches **/lib** for **helpfile**.

#include <regexp.h>
regexp *regcomp(*exp***) char ***exp***;**
int regexec(*prog***, *** string***) regexp ***prog***; char ***string***;**
regsub(*prog***, *** source***, *** dest***) regexp ***prog***; char ***source***; char ***dest***;**
regerror(*msg***) char ***msg***;**
These functions implement a standard method for parsing regular expressions. **regcomp()** turns a regular expression into a structure of type **regexp** and returns a pointer to it. **regexec()** matches *string* against the regular expression in *prog*. It returns one if *string* matches *exp*, and zero if it does not. **regsub()** copies *source* to *dest*, and makes substitutions according to the most recent **regexec()** performed using *prog*. **regerror()** is called whenever an error is detected in **regcomp()**, **regexec()**, or **regsub()**. See **regexp.doc** for details.

long randl()
Return a long random number uniformly distributed between 1 and 2,147,483,562. This comes from *Communications of the ACM*, volume 31, number 6. See **srandl()**, below.

char *replace(*s1***, *** pat***, *** s3***, *** all***, *** matcher***) char ***s1***, ***pat***, ***s3***, (***matcher***)();**
Replace one or all occurrences of *pat* in string *s1* by *s3*, and return the result. The definition of match is set by *matcher*. This calls the user-defined function

matcher(*sw***, *** pat***, &***fin***).**

The *matcher* must return the address of the pattern match and its end in **&***fin***. match()** is a valid example of *matcher*. It replaces the first occurrence, or all occurrences of the pattern, and returns the new pattern. The new pattern has been **alloc()**'d.

showflag(*data***, *** flags***, *** output***) long ***data***; char ***flags***, ***output***;**
Turn the bits in *data* to the flags in *flags* or '-' in the string *output*, which must be as long as *flags*.

char *skip(*s1***, *** matcher***, *** fin***) char ***s1***, ****fin***; int (****matcher***)();**
Skip all initial characters in string *s1* that fail when examined *matcher*. *matcher* is usually a character function, e.g., **isdigit()**. It returns the first character skipped. **skip()** points *fin* at the character after the skip.

char *span(*s1***, *** matcher***, *** fin***) char ***s1***, ****fin***; int (****matcher***)();**
Span all initial characters in string *s1* that match when examined by *matcher*. *matcher* is usually a character function, e.g., **isdigit()**. It returns the first character spanned. **span()** points *fin* at the character after the span.

LEXICON

srandl(*seed1***,** *seed2***) long** *seed1***,** *seed2***;**
randl() needs two seeds; **srandl()** sets them. Use it only if you need to repeat a random-number sequence.

strchtr(*from***,** *to***,** *c***,** *def***)**
char **from***, **to***; **int** *c***,** *def***;**
Look up the character *c* in the string *from*. Return the corresponding character in the string *to* if it is found; otherwise, return the default character *def*.

For example, consider the call:

```
strchtr("ab", "xy", c, d);
```

If variable *c* equals 'a', then **strchtr()** returns 'x'; if *c* equals 'b', then it returns 'y'; otherwise, it returns the value of *d*

strcmpl(*s1***,** *s2***)**
Case-insensitive string comparison. Resembles **strcmp()**.

jday_t time_to_jday(*time***) time_t** *time***;**
Turn COHERENT time to Julian date structure. The Julian date is the number of days since the beginning of the Julian calendar, January 1, 4713 B.C. The structure **jday_t** is defined in **misc.h**. Type **time_t** is defined in **<sys/types.h>**.

jday_t tm_to_jday(*tm***) tm_t** **tm***;**
Turn the time structure **tm** date into Julian date structure. Structure **tm** is defined in **<time.h>**.

char **trim(***s***) char** **s***;**
Remove trailing whitespace from string *s*.

ucase(*s***) char** **s***;**
Convert a string to upper case.

usage(*s***) char** **s***;**
Print string *s* and call **exit(1)**.

xdump(*p***,** *length***) char** **p***;**
Print on stdout a vertical hexadecimal dump of string *p*.

A vertical hexadecimal dump prints as three lines. The top line is the display character, or '.' if the character cannot be displayed cleanly. The next two lines are the hexadecimal numerals. The data are blocked into groups of four bytes.

xopen(*filename***,** *acs***) char** **filename***, **acs***;**
Like **fopen()**, but call **fatal()** if the open fails.

yn(*question***, ...) char** **question***;**
Ask a question and retrieve a 'Y' or 'N' answer. *question* is a **printf()**-style format string; any trailing parameters should point to data used in *question*. **yn()** returns one if the user answers 'Y' or 'y', and returns zero if she answers 'N' or 'n'.

Virtual Memory

The following functions are part of a virtual memory system for COHERENT 286. Occasionally, users port programs like **compress** to COHERENT 286 that use a small number of very large arrays. Because COHERENT 286 is a SMALL-model operating system, special provision must be made for arrays too large to fit into a 64-kilobyte data segment. The following functions enable programs that are to be run under COHERENT 286 use very large arrays:

void vinit(*filename***,** *ram***) char** **filename***; **unsigned** *ram***;**
Initialize the virtual-memory system, using *filename* for work. *filename* may be a raw device, such as */dev/rram1*. *ram* is the amount of buffer space to give the system — the more, the better.

void vshutdown()
Shut the virtual-memory system, and make it restartable.

unsigned vopen(*amt***) unsigned long** *amt***;**
Set up a virtual-memory object. For example, if you want to emulate having a 100,000-byte array and a 50,000-byte array, use the call

```
vid1 = vopen(100000L); vid2 = vopen(50000L);
```

This does some checking and tells the system that any reference to *vid2* will be between 100,000 and 150,000 in the virtual file.

char ***vfind(***vid***,** *disp***,** *dirty***)**
unsigned *vid***,** *dirty***; unsigned long** *disp***;**
Find a character in the virtual system, mark the block's dirty bit if the access is to write. Given the example in **vopen()**, if you want to find the 1,000th byte in *vdi1*, use the call:

```
c = *(vfind(vdi1, 1000L, 0));
```

To change the 2000th byte in *vid2* **d**, use the call

```
*(vfind(vid2, 2000L, 1)) = d;
```

Note that the dirty indicator tells the system of the change so that the block will be written back before it is read over. Blocks are 512 bytes long, so **int**'s or **long**'s can be read or written without multiple accesses to **vfind()**.

File Locking

libmisc holds a number of routines with which you can lock and unlock files and devices. It is adapted from the mechanism used by the COHERENT implementation of UUCP.

Lock files are created in **$SPOOLDIR**. A lock file is given the name **LCK..***resource*. It contains a decimal representation of the process identifier (pid) of the process that created the lock.

You can provide an alternate pid by using one of the 'n' routines — i.e., **locknrm()**, **lockntty()**, and **unlockntty()**. The unlocking routines regard a pid of zero as an override — they remove the lock regardless of which process created the lock.

For a tty device, *resource* is a string that consists of a decimal representation of its major number, a decimal point, and the lower five bits of its minor number.

Each routine takes a string that names the resource to lock or unlock. The "tty" routines (i.e., **lockntty()**, **locktty()**, **unlockntty()**, and **unlocktty()**) want the base name of the tty to be locked (without the **/dev/** part).

Every lock routine returns zero on failure and one on success.

lockit(*resource***)** char ***resource***;**
Use a resource string to lock a tty.

lockexist(*resource***)** char ***resource***;**
Check whether a lock file exists for the tty with *resource*.

locknrm(*resource***,** *pid***) char** ***resource***; int** *pid***;**
Remove a lock file for a tty owned by process *pid*.

lockntty(*tty***,** *pid***) char** ***tty***; int** *pid***;**
Lock a tty for process *pid*.

lockrm(*resource***)** char ***resource***;**
Remove a lock file for tty with *resource*.

locktty(*tty***) char** ***tty***;**
Lock a tty.

lockttyexist(*tty***) char** ***tty***;**
Check whether a given tty is locked.

unlockntty(*tty***,** *pid***) char** ***tty***; int** *pid***;**
Unlock a tty for process *pid*. Unlocking always succeeds.

unlocktty(*tty***) char** ***tty***;**
Unlock a tty that the current process owns.

unlockit(*resource***,** *pid***) char** ***resource***; int** *pid***;**
Unlock something for process *pid*.

LEXICON

Templates and Pictures

libmisc includes a function, **picture()**, for formatting numeric strings. It has the following syntax:

double picture(*dble***,** *format***,** *output***)**
double *dble***; char** **format***,* **output***;*

picture() performs numeric formatting under C. It resembles masking functions built into COBOL and BASIC, but is superior to either. *dble* gives the number to format; *format* gives the format mask; and *output* points to the area into which the formatted number is written. *output* must be at least as large as *format*. If *dble* overflows the picture, **picture()** returns the overflow.

The following summarizes the values that can appear in the *format* string. Note that throughout, the symbol **<sp>** indicates a space character, not the literal string "<sp>".

9 Provide a slot for a number. Passing 5.000 through a mask of **999 CR** gives "005". Passing -5.000 through a mask of **999 CR** yields "005 CR". Note that **picture()** does not recognize the characters 'C' and 'R' as being special. Trailing non-special characters print only if the number is negative.

Z Provide a slot for a number, but suppress leading zeroes. For example, passing 1034.000 through a mask of **ZZZ,ZZZ** gives "<sp><sp>1,034". Note that **picture()** does not recognize a comma as being a special character. **picture()** prints embedded non-special characters only if they are preceeded by significant digits.

J Provide a slot for a number, but shrink out leading zeros. For example, passing 1034.000 through a mask of **JJJ,JJJ** yields "1,034".

K Provide a slot for a number, but shrink out any zeros. For example, passing 70884.000 through a mask of **K9/K9/K9** yields "7/8/84".

$** Float a dollar sign to the left of the displayed number. For example, passing 105.000 through a mask of **$ZZZ,ZZZ yields "<sp><sp><sp><sp>$105".

. Separate the number between decimal and integer portions. For example, passing 105.670 through a mask of **Z,ZZZ.999** yields "<sp><sp>105.670".

T Provide a slot for a number, but suppress trailing zeroes. For example, passing 105.670 through a mask of **Z,ZZ9.9TT** yields "<sp><sp>105.67<sp>".

S Provide a slot for a number, but shrink out trailing zeroes. For example, passing 105.670 through a mask of **Z,ZZ9.9SS** yields "<sp><sp>105.67".

- Float a negative sign in front of negitive numbers. For example, passing 105.000 through a mask of **-Z,ZZZ** yields "<sp><sp><sp<105", whereas passing -105.000 through a mask of **-Z,ZZZ** yields "<sp><sp>-105".

(Acts like **-**, but prints a parenthesis. For example, passing 105.000 through a mask of **(ZZZ)** yields "<sp>105<sp>", whereas passing -5.000 through a mask of **(ZZZ)** yields "<sp><sp>(5)".

+ Float a **+** or **-** in front of the number, depending on its sign. For example, pasing 5.000 through a mask of **+ZZZ** yields "<sp><sp>+5", whereas passing -5.000 through a mask of **+ZZZ** yields "<sp><sp>-5".

***** Fill all spaces to right with asterisks. For example, passing 104.100 through a mask of ***ZZZ,ZZZ.99** yields "*****104.10"; whereas passing 104.100 through a mask of ***$ZZZ,ZZZ.99** yields "*****$104.10". **picture()** returns any overflow as a **double**. For example, passing -1234.000 through a mask of **(ZZZ)** yields "(234)" with an overflow of -1.0; passing 123.400 through a mask of **99** yields "23" with an overflow of 1.0; and passing 1200.000 through a mask of **ZZ** yields "00" with an overflow of 12.0.

Files

/usr/src/misc.tar.Z — Compressed **tar** archive of sources

See Also

libraries, tar, zcat

Notes

The **misc** library is provided on an *as-is* basis only. *Caveat utilitor!*

libmp — Library

Library for multiple-precision mathematics
/usr/lib/libmp.a

The COHERENT library **libmp** contains routines that allow you to perform multiple-precision arithmetic. These functions manipulate a data structure called a **mint**, or "multiple-precision integer," which the header file **mprec.h** defines as follows:

```
typedef struct {
      unsigned len;
      char *val;
} mint;
```

You should not depend on the details of this structure, because on some machines a different representation may be more efficient. Using the listed functions is always safe.

The following gives the multiple-precision routines:

gcd(). . . . . Set variable to greatest common divisor
ispos() . . . . . Return if variable is positive or negative
itom(). . . . . Create a multiple-precision integer
madd() . . . . . Add multiple-precision integers
mcmp(). . . . . Compare multiple-precision integers
mcopy() . . . . . Copy a multiple-precision integer
mdiv(). . . . . Divide multiple-precision integers
min() . . . . . Read multiple-precision integer from stdin
minit() . . . . . Condition global or auto multiple-precision integer
mintfr(). . . . . Free a multiple-precision integer
mitom() . . . . . Reinitialize a multiple-precision integer
mneg() . . . . . Negate multiple-precision integer
mout() . . . . . Write multiple-precision integer to stdout
msqrt() . . . . . Compute square root of multiple-precision integer
msub() . . . . . Subtract multiple-precision integers
mtoi(). . . . . Convert multiple-precision integer to integer
mtos(). . . . . Convert multiple-precision integer to string
mult(). . . . . Multiply multiple-precision integers
mvfree() . . . . . Free multiple-precision integer
pow() . . . . . Raise multiple-precision integer to power
rpow(). . . . . Raise multiple-precision integer to power
sdiv() . . . . . Divide multiple-precision integers
smult() . . . . . Multiply multiple-precision integers
spow(). . . . . Raise multiple-precision integer to power
xgcd(). . . . . Extended greatest-common-divisor function
zerop() . . . . . Indicate if multi-precision integer is zero

itom() creates a new **mint**, initializes it to the signed integer value *n*, and returns a pointer to it. Storage used by a **mint** created with **itom** may be reclaimed using **mintfr()**.

A **mint** that already exists may be reinitialized by **mitom()**, which sets *a* to the value *n*. If the **mint** was declared as a global or automatic variable, it must be conditioned before first use by **minit()**, which prevents garbage values in the **mint** structure from causing chaos. A **mint** conditioned by **minit()** has no value; however, it may be used to receive the result of an operation. For **mint**s automatic to a function, **mvfree()** should be used before the function is exited to free the storage used by the **val** field of the **mint** structure. Otherwise, this storage will never be reclaimed.

madd(), **msub()**, and **mult()** set *c* to the sum, difference, or product of *a* and *b*. **mdiv** divides *a* by *b* and writes the quotient and remainder in *q* and *r*. *b* must not be zero. The results of the operation are defined by the following conditions:

1. $a=q*b+r$

2. The sign of *r* equals the sign of *q*

3. The absolute value of *r* < the absolute value of *b*.

smult() is like **mult()**, except the second argument is an integer in the range $0 \le n \le 127$. **sdiv()** is like **mdiv()**, except the second argument is an integer in the range $1 \le n \le 128$, and the remainder argument points to an **int** instead of a **mint()**.

pow() sets *c* to *a* raised to the *b* power reduced modulo *m*. **rpow()** sets *c* to *a* raised to the *b* power. **spow()** is like **rpow()**, except the exponent is an integer. In no case may the exponent be negative.

mcopy() sets *b* equal to *a*. **mneg()** sets *b* equal to negative *a*.

msqrt() sets *b* to the integral portion of the positive square root of *a*; *r* is set to the remainder. *a* must not be negative. The result of the operation is defined by the condition

$$a = b * b + r$$

gcd() sets *c* to the greatest common divisor of *a* and *b*. **xgcd()** is an extended gcd routine that sets *g* to the greatest common divisor of *a* and *b*, and sets *r* and *s* so the relation

$$g = a * r + b * s$$

holds. For **xgcd()**, *r*, *s* and *g* must all be distinct.

mints may be compared with **mcmp()**, which returns a signed integer less than, equal to, or greater than zero according to whether *a* is less than, equal to, or greater than *b*. **ispos()** returns true (nonzero) if *a* is not negative, false (zero) if *a* is negative. **zerop** returns true if *a* is zero, false otherwise.

mtoi() returns an integer equal to the value of *a*. *a* should be in the allowable range for a signed integer.

The external integers **ibase** and **obase** govern the I/O and ASCII conversion routines. Allowable bases run from two to 16. Permissible digits are 0 through 9 and A through F (lower-case letters are not allowed). **min** reads a **mint** in base **ibase** from the standard input and sets *a* to that value. Leading blanks and an optional leading minus sign are allowed; the number is terminated by the first non-legal digit. **mout()** outputs *a* on the standard output in base **obase. mtos()** performs the same conversion as **mout()**, but the result is placed in a character string instead of being output; a pointer to the string is returned. The string is actually allocated by **malloc()**, and may be freed by **free()**.

mzero() and **mone()** point to **mint**s with values zero and one. **mminint()** and **mmaxint()** point to **mint**s containing the minimum and maximum values that will fit in a signed integer. These constants should never be used as the result of an operation.

All the necessary declarations for these constants and for the library functions are contained in the header file **mprec.h**. They need not be repeated.

To link **libmp** modules into an executable object, use the argument **-lmp** at the end of the **cc** command.

Example

The following example converts a string into a multi-precision integer.

```
#include <stdio.h>
#include <mprec.h>
#include <ctype.h>

/*
 * "ibase" is an int which contains the input base used by "stom".
 * It should be between 2 and 16.
 */
int ibase = 10;
```

```
/*
 * stom() reads in a number in base ibase from string 'a' and returns
 * pointer to multiple-precision integer.
 */
mint *stom(s)
register char *s;
{
      char cval;
      mint c = {1, &cval};
      register int ch;
      char mifl = 0;    /* leading minus flag */
      static mint number;

      mcopy(mzero, &number);  /* set number to zero */
      if ((ch = *s) == '-') { /* skip leading '-' */
            mifl = 1;
            ch = *++s;
      }

      /* scan thru string 's', building result in "number" */
      while (isascii(ch) && isdigit(ch)) {
            cval = (isdigit(ch) ? ch - '0': ch - 'A');
            smult(&number, ibase, &number);
            madd(&number, &c, &number);
            ch = *++s;
      }

      if (mifl)   /* adjust sign of a "number" */
            mneg(&number, &number);
      return(&number);
}

/* simple test for "stom" */
main()
{
      char  buffer[80];

      printf("Input string ? ");
      gets(buffer);
      mout(stom(buffer)); /* Print in stdout multiple-precision int */
}
```

Files

<mprec.h>
/usr/lib/libmp.a

See Also

bc, dc, libraries, malloc(), mprec.h

Diagnostics

On any error, such as division by zero, running out of space or taking the square root of a negative number, an appropriate message is printed on the standard error stream and the program exits with a nonzero status.

LIBPATH — Environmental Variable

Directories that hold compiler phases and libraries

LIBPATH names the directories that hold the phases of the COHERENT C compiler, the run-time start-up modules, and libraries. **cc** searches these directories as it orchestrates the compiling and linking of a program written in C or assembly-language.

A typical definition is:

```
export LIBPATH=:/lib:/usr/lib
```

This searches the current directory '.', then **/lib**, then **/usr/lib**.

If you have not set **LIBPATH** in your **.profile**, **cc** uses the default **LIBPATH** that is set in header file **path.h**. This definition is adequate for all standard installations of COHERENT.

See Also

cc, environmental variables, ld

libraries — Overview

A **library** is an archive file of commonly used functions that have been compiled, tested, and stored for inclusion in a program at link time.

The COHERENT system stores its libraries in two directories: **/usr/lib** and **/lib**. or their subdirectories: The following libraries are kept in **/usr/lib**:

libcurses.a . . . . . **curses** library and **terminfo** functions
libedit.a . . . . . Routines to gather and edit user input
libgdbm.a . . . . . Library for GNU DBM functions
libl.a . . . . . **lex** library
libmp.a . . . . . Multi-precision arithmetic library
libsocket.a . . . . . **sockets** emulation library
libterm.a . . . . . Functions to read **termcap** data
liby.a . . . . . **yacc** library
lib.b . . . . . **bc**'s function library (in **bc** source)

The following libraries are kept in **/lib**:

libc.a . . . . . General functions and system calls
libm.a . . . . . Mathematics routines

In addition, COHERENT comes with a library of miscellaneous routines, called **libmisc**. See the Lexicon article **libmisc** for information on how to prepare this library for use.

Lexicon Articles

The following Lexicon articles introduce the library functions included with the COHERENT system:

libc
libcurses
libedit
libgdbm
libm
libmisc
libmp
libsocket
libterm

See Also

ar, C language, Programming COHERENT

libsocket — Library

Library of communications routines

libsocket is a library of routines that emulate the Berkeley **sockets** library. It includes the following functions:

accept() . . . . . Accept a connection on a socket
bind() . . . . . Bind a name to a socket
bitcount() . . . . . Count bits in a bit-mask
connect() . . . . . Connect to a socket
endhostent() . . . . . Close file **/etc/hosts**
endnetent() . . . . . Close network file
endprotoent() . . . . . Close protocols file
endservent() . . . . . Close protocols file
ffs() . . . . . Translate a bit mask into an integer value
getdtablesize() . . . . . Get the number of files a process can open
gethostbyaddr() . . . . . Retrieve host information by address
gethostbyname() . . . . Retrieve host information by name
gethostname() . . . . . Get the name of the local host
getnetbyaddr() . . . . . Get a network entry by address
getnetbyname() . . . . . Get a network entry by address

getnetent() . . . . . Fetch a network entry
getpeername() . . . . . Get name of connected peer
getprotobyname() . . . Get protocol entry by protocol name
getprotobynumb() . . . Get protocol entry by protocol number
getprotoent() . . . . . Get protocol entry
getservbyname() . . . . Get a service entry by name
getservbyport() . . . . . Get a service entry by port number
getservent() . . . . . Get a service entry
getsockname() . . . . . Get the name of a socket
getsockopt(). . . . . Read a socket option
gettimeofday() . . . . . Berkeley time function
inet_addr(). . . . . Transform an IP address from text to binary
inet_network() . . . . . Transform an IP address from text to an integer
listen() . . . . . Listen for a connection on a socket
random(). . . . . Return a random number
recv() . . . . . Receive a message from a connected socket
recvfrom() . . . . . Receive a message from a socket
select() . . . . . Check whether sockets are ready for activity
send(). . . . . Send a message to a connected socket
sendto() . . . . . Send a message to a socket
sethostent(). . . . . Open and rewind file **/etc/hosts**
setnetent(). . . . . Open and rewind file **/etc/networks**
sethostent(). . . . . Open and rewind file **/etc/hosts**
setprotoent() . . . . . Open the protocols file
setservent() . . . . . Open the services file
setsockopt(). . . . . Set a socket option
shutdown() . . . . . Replace function to shut down system
SOCKADDRLEN(). . . . Return length of an address
socket(). . . . . Create a socket
socketpair() . . . . . Create a pair of sockets
srandom() . . . . . Seed the random-number generator
strcasecmp() . . . . . Case-insensitive string comparison
strcasencmp(). . . . . Case-insensitive string comparison
usleep(). . . . . Sleep briefly

Function **socket()** creates a socket; the caller dictates the type of socket to be created, and the communications protocol that it comprehends. **socket()** returns a descriptor, which resembles a file descriptor and which can be passed to the system calls **read()** and **write()** to exchange information with whatever plugs itself into that socket. (For details, see the **Notes** section at the end of this article.)

Function **bind()** binds the newly created socket to a file that you name. To await a connection with another process, invoke the function **listen()**; this alerts the system to the fact that you (via your socket) await messages of a given type. Function **select()** checks whether one or more sockets are ready to be written to, or hold data that need to be read. When a message becomes available, invoke function **accept()** to accept communication with the process that wishes to connect to your socket. These functions generally are used by "server" sockets.

Function **connect()** directly establishes connection with a server socket via its name (that is, via the file to which it is bound). Once connection is established, information can be exchanged via the COHERENT system calls **read()** and **write()**.

System Files

The socket library manipulates the following files. Each is described in its own Lexicon entry:

hosts . . . . . Names and addresses of hosts on the local network
hosts.equiv . . . . . Name equivalent hosts
hosts.lpd. . . . . Local system name and domain
inetd.conf . . . . . Configure the Internet daemons
networks. . . . . Name remote networks
protocols. . . . . Name supported protocols
services . . . . . List supported TCP/IP services

Example

For following gives a pair of programs that demonstrate sockets. They were written by John Dhuse

(jdhuse@sedona.intel.com).

The example consists of two programs, **server.c** and **client.c**. Compile each with the switch **-lsocket**. To see how they work, run each in its own virtual console or **xterm** window. Do not run them in the background; otherwise, you will not be able to work with them interactively. Be sure to start up **server** first, as it creates the socket into which **client** plugs itself.

Each process gives you a prompt; you can type commands into each. **server** recognizes the following commands:

? Print the command menu

c Call **select()** to check the socket. **client** displays the status of the socket.

s Send a string to **server**. **client** prompts for the string, reads up to 20 characters, and writes it to the socket.

r Read from the socket. **client** prompts for the number of bytes to read, and clips any response to a maximum of 20.

q Close the socket and terminate the server process.

server recognizes the following commands:

? Print the command menu.

c Call **select()** to check the socket.

r Read from the socket. **server** does not prompt for the number of bytes to read, but tries to read the entire contents of the socket, up to a maximum of 20 bytes.

e Echo the read message back to the client. The server cannot send its own message the client, just echo what it received.

q Close the socket, terminate the server, and **unlink()** the socket file.

The following gives the source for **server.c**:

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/un.h>

main()
{
      int sd, nsd, err,i,j,rdfds[2],wrtfds[2];
      int efds[2],done,r;
      int arg=1;
      struct sockaddr_un server;
      char *sock_name = "u0";
      char buf[20];
      char command,line[80];
      struct timeval timeout;

      /* clear our address */
      bzero((char *)&server, sizeof(server));

      /* create socket */
      if ((sd = socket(AF_UNIX, SOCK_STREAM, 0)) <= 0) {
            err = errno;
            fprintf(stderr, "server: can't create socket\n");
            fprintf(stderr, "server: errno = %d\n", err);
            exit(EXIT_FAILURE);
      }

      server.sun_family = AF_UNIX;
      bcopy(sock_name,server.sun_path,strlen(sock_name));
```

```
  /* bind the socket */
if ((bind(sd, (struct sockaddr *)&server, sizeof(server))) != 0) {
      err = errno;
      fprintf(stderr, "server: can't bind socket\n");
      fprintf(stderr, "server: errno = %d\n", err);
      close(sd);
      exit(EXIT_FAILURE);
}

  /* listen on the socket */
if ((listen(sd, 1)) != 0) {
      err = errno;
      fprintf(stderr, "server: can't listen on socket\n");
      fprintf(stderr, "server: errno = %d\n", err);
      close(sd);
      exit(EXIT_FAILURE);
}

/* accept connections on the socket */
if ((nsd = accept(sd, (struct sockaddr*)0, (int *)0)) == -1) {
      err = errno;
      fprintf(stderr, "server: can't accept connection\n");
      fprintf(stderr, "server: errno = %d\n", err);
      close(sd);
      exit(EXIT_FAILURE);
}

printf("accepted client connection fd %d\n",nsd);
  /* set to non-blocking io */
ioctl(nsd,FIOSNBIO,&arg);

/* echo every message back to client, exit on terminate string */
printf("entering command loop\n");
command = 'a';
while (command != 'q') {
      printf("server> ");
      scanf("%s",line);
      sscanf(line,"%c", &command);
      switch (command) {

      case 'c' :
            /* set up for select */
            rdfds[0] = 1 << nsd; rdfds[1] = 0;
            wrtfds[0] = 1 << nsd; wrtfds[1] = 0;
            efds[0] = 1 << nsd; efds[1] = 0;
            timeout.tv_sec = 0; timeout.tv_usec = 0;
            r = select(nsd+1,rdfds,wrtfds,efds,(struct timeval *)NULL);
            err = errno;

            if (r < 0)
                      printf("select() returned errno %d\n",err);
            else {
                    if (rdfds[0] & (1 << nsd))
                        printf("socket has data to be read\n");
                  if (wrtfds[0] & (1 << nsd))
                            printf("data can be written to socket\n");
                  if (efds[0] & (1 << nsd))
                        printf("select reports exception on socket\n");
            }
            break;
```

```
            case 'r' :
                      bzero(&buf[0], sizeof(buf));
                  j = read(nsd,buf,sizeof(buf));
                  err = errno;
                  if (j < 0)
                        printf("read() returned errno %d\n",err);
                  else
                        printf("got %d bytes, msg is >%s<\n",j,buf);
                  break;

            case 'e' :
                  printf("echoing >%s< (%d bytes) to client\n",buf,j);
                  write(nsd,&buf[0],j);
                  break;

            case 'q' :
                  close(nsd);
                  close(sd);
                  unlink(sock_name);
                  break;

            case '?' :
                  printf("commands:\n");
                  printf("   c - check the socket\n");
                  printf("   ? - this help message\n");
                  printf("   r - read from socket\n");
                  printf("   e - echo received message to client\n");
                  printf("   q - close socket and quit\n");
                  break;

              default :
                  printf("\n");
                  break;
            }
      }
}
```

The following gives the source for **client.c**:

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/un.h>

main()
{
      int sd, err, i, j, flags;
      int arg=1;
      struct sockaddr_un client;
      char *address="u0";
      char buf[20];
      char command,line[80];
      int rdfds[2],wrtfds[2];
      struct timeval timeout;

      /* clear our address */
      memset((char *)&client,0, sizeof(client));

      /* create socket */
      if ((sd = socket(AF_UNIX, SOCK_STREAM, 0)) <= 0) {
            err = errno;
            fprintf(stderr, "client: can't create socket\n");
            fprintf(stderr, "client: errno = %d\n", err);
            exit(EXIT_FAILURE);
      }
```

```
/* set to blocking so connect hangs, waiting for connect */
arg = 0;
       i = ioctl(sd,FIOSNBIO,&arg);

client.sun_family = AF_UNIX;
memcpy(client.sun_path,address,2);

/* connect to the socket */
if (connect(sd, (struct sockaddr *)&client, sizeof(client))) {
      err = errno;
      fprintf(stderr, "client: can't connect socket\n");
      fprintf(stderr, "client: errno = %d\n", err);
      close(sd);
      exit(EXIT_FAILURE);
}
printf("connected socket fd = %d\n",sd);

arg = 1;
       i = ioctl(sd,FIOSNBIO,&arg);

printf("entering command loop\n");
command = 'a';

while (command != 'q') {
      printf("client> ");
      scanf("%s",line);
      sscanf(line,"%c",&command);

      switch (command) {
      case 's' :
            printf("message to send: ");
            scanf("%s",buf);
            i = write(sd,buf,strlen(buf));
            err = errno;
            if (i < 0)
                  printf("write() returned errno %d\n", err);
            else printf("sent >%s< (%d bytes) to server\n",
                        buf,strlen(buf));
            break;

      case '?' :
            printf("commands:\n");
            printf("   s - send a message\n");
            printf("   ? - this help messge\n");
            printf("   c - check the socket\n");
            printf("   r - read from socket\n");
            printf("   b - set to blocking I/O\n");
            printf("   n - set to non-blocking I/O\n");
            printf("   q - close socket and quit\n");
            break;

      case 'b' :
              arg = 0;
            ioctl(sd,FIOSNBIO,&arg);
            printf("I/O is blocking\n");
            break;

      case 'n' :
              arg = 1;
            ioctl(sd,FIOSNBIO,&arg);
            printf("I/O is non-blocking\n");
            break;
```

```
            case 'c' :
                    /* setup query fields */
                    rdfds[0] = 1 << sd; rdfds[1] = 0;
                  wrtfds[0] = 1 << sd; wrtfds[1] = 0;
                  timeout.tv_sec = 0; timeout.tv_usec = 0;
                  i = select(sd+1,rdfds,wrtfds,(int *)NULL,
                           &timeout);
                  err = errno;
                  if (i < 0)
                          printf("select() returned error %d\n",err);
                  else {
                        if (rdfds[0] & (1 << sd))
                              printf("socket has data to be read\n");
                        if (wrtfds[0] & (1 << sd))
                              printf("data can be written to socket\n");
                    }
                  break;

            case 'r' :
                  printf("number of bytes to read > ");
                  scanf("%d",&i);
                  if (i > sizeof(buf)) i = sizeof(buf);
                    memset(&buf[0],0, sizeof(buf));
                  j = read(sd,buf,i);
                  err = errno;
                  if (j < 0)
                        printf("read() returned errno %d\n",err);
                  else
                        printf("got %d bytes, msg is >%s<\n",j,buf);
                  break;

            case 'q' :
                  close(sd);
                  break;

            default:
                  printf("\n");
                  break;
            }
      }
      exit(EXIT_SUCCESS);
}
```

See Also

device driver, hosts, hosts.equiv, hosts.lpd, inetd.conf, libraries, msgget(), named pipes, networks, pipe(), protocols, semget(), services, shmget(), STREAMS

Notes

The version of sockets included with COHERENT is not built into the kernel. Rather, it uses a library of routines that use named pipes to emulate sockets. You should not invoke the system calls **read()** or **write()** to manipulate directly any descriptor returned by a call to **socket()**, because this descriptor defines only one of a set of named pipes required to mimic a true kernel-level socket. Header file **<sys/socket.h>** replaces these with the macros that perform the task correctly. This means that in every C file where you perform a **read()**, **write()**, **ioctl()**, or **close()** on a socket, you must include **<sys/socket.h>**.

This library was adapted from Berkeley sources by P.Garbha (pgd@compuram.bbt.se), and was extensively revised by Mark Williams Company.

This product includes software developed by the University of California, Berkeley, and its contributors.

libterm — Library

Functions to read termcap descriptions

The library **libterm** holds the following routines, with which a program can read a **termcap** description:

tgetent() Read a **termcap** entry.

tgetflag() Check if a given Boolean capability is present in the terminal's **termcap** entry.

tgetnum() Return the value of a numeric **termcap** feature (e.g., the number of columns on the terminal).

tgetstr() Read and decode a **termcap** string feature.

tgoto() Read and decode a cursor-addressing string.

tputs() Read and decode the leading padding information of a **termcap** string feature.

See the Lexicon entry for each function for details.

See Also

libcurses, libraries, termcap

limits.h — Header File

Define numerical limits
#include <limits.h>

The header file **<limits.h>** defines macros that set the numerical limits for the translation environment.

The following table gives the macros defined in **limits.h**. Each value given is the macro's minimum maximum: a conforming implementation of C must meet these limits, and may exceed them.

CHAR_BIT
Number of bits in a **char**. This must be at least eight.

CHAR_MAX
Largest value representable in an object of type **char**. If the implementation defines a **char** to be signed, then it is equal to the value of the macro **SCHAR_MAX**; otherwise, it is equal to the value of the macro **UCHAR_MAX**.

CHAR_MIN
Smallest value representable in an object of type **char**. If the implementation defines a **char** to be signed, then it is equal to the value of the macro **SCHAR_MIN**; otherwise, it is zero.

INT_MAX
Largest value representable in an object of type **int**; it must be at least 32,767 (0x7FFF).

INT_MIN
Smallest value representable in an object of type **int**; no less more -32,767.

LONG_MAX
Largest value representable in an object of type **long int**; it must be at least 2,147,483,647 (0x7FFFFFFFL).

LONG_MIN
Smallest value representable in an object of type **long int**; it must be at most -2,147,483,647.

MB_LEN_MAX
Largest number of bytes in any multibyte character, for any locale; it must be at least one.

OPEN_MAX
The maximum number of file descriptors that a process can hold at any given time.

Please note that this constant gives a "snapshot" of the state of COHERENT at this time. Using this constant in a program, in particular to size an array, greatly decreases the portability of a program, and may cause it to behave incorrectly. To determine the number of file descriptors that the operating system permits right now, use the system call **sysconf()**. *Caveat utilitor!*

SCHAR_MAX
Largest value representable in an object of type **signed char**; it must be at least 127.

SCHAR_MIN
Smallest value representable in an object of type **signed char**; it must be at most -127.

SHRT_MAX
Largest value representable in an object of type **short int**; it must be at least 32,767 ((**short**)0x7FFF).

SHRT_MIN
Smallest value representable in an object of type **short int**; it must be at most -32,767.

UCHAR_MAX
Largest value representable in an object of type **unsigned char**; it must be at least 255.

UINT_MAX
Largest value representable in an object of type **unsigned int**; it must be at least 65,535 ((**unsigned int**)0xFFFF).

ULONG_MAX
Largest value representable in an object of type **unsigned long int**; it must be at least 4,294,967,295 ((**unsigned long**)0xFFFFFFFFL).

USHRT_MAX
Largest value representable in an object of type **unsigned short int**; it must be at least 65,535 ((**unsigned short**)0xFFFF).

See Also

header files
ANSI Standard, §5.2.4.2.1
POSIX Standard, §2.8

Notes

limits.h sets fixed limits. If a limit is not completely fixed, then the symbol is not defined, and a process must use **sysconf()** or **pathconf()**, as appropriate, to find the limit's value for the current run of the process.

lines — Command

Highly amusing board game
/usr/games/lines

lines is an interactive COHERENT version of a two-player board game by Claude Soucie called *Lines of Action*. The screen displays the game board with "X" and "O" characters marking the positions of the pieces. To see the rules of the game, type "r" and then press **<Enter>**. To see the available interactive commands, type "h" and press **<Enter>**.

Two players can use **lines** to keep track of a game between them by moving with the "M" command. Alternatively, one player can play against the computer by moving with the "m" command. The program uses a tree-search technique to consider possible moves; the player can vary the speed of the program's replies with commands that change the tree search width and depth.

For a more detailed description of *Lines of Action*, see *A Gamut of Games* by Sid Sackson (New York, Random House, 1969).

See Also

commands

link() — System Call (libc)

Create a link
#include <unistd.h>
link(*old, new***)**
char **old, *new***;**

A *link* to a file is another name for the file. All attributes of the file appear identical among all links.

link() creates a link called *new* to an existing file named *old*.

For administrative reasons, it is an error for users other than the superuser to create a link to a directory. Such links can make the file system no longer tree structured unless carefully controlled, posing problems for commands such as **find**.

Examples

The first example, called **lock.c**, demonstrates how **link()** can be used to perform intertask locking. With this technique, a program can start a process in the background and stop any other user from starting the identical process.

```
#include <unistd.h>
main()
{
      if(link("lock.c", "lockfile") == -1) {
            printf("Cannot link\n");
            exit(1);
      }

      sleep(50);  /* do nothing for 50 seconds */
      unlink("lockfile");
      printf("done\n");
      exit(EXIT_SUCCESS);
}
```

The second example demonstrates how to use **link()** and **unlink()** to rename a file.

```
#include <stdio.h>
#include <unistd.h>
main(argc, argv) int argc; char *argv[];
{
      register char *old, *new;

      if (argc != 3) {
            fprintf(stderr, "Usage: rename old new\n");
            exit(EXIT_FAILURE);
      }
      old = argv[1];
      new = argv[2];

      if (link(old, new) == -1) {
            fprintf(stderr, "rename: link(%s, %s) failed\n", old, new);
            exit(EXIT_FAILURE);
      } else if (unlink(old) == -1) {
            fprintf(stderr, "rename: unlink(%s) failed\n", old);
            exit(EXIT_FAILURE);
      }
      exit(EXIT_SUCCESS);
}
```

See Also

find, libc, ln, rename(), unlink(), unistd.h
POSIX Standard, §5.3.4

Diagnostics

link() returns zero when successful. It returns -1 on errors, e.g., *old* does not exist, *new* already exists, attempt to link across file systems, or no permission to create *new* in the target directory.

Notes

Because each mounted file system is a self-contained entity, links between different mounted file systems fail.

listen() — Sockets Function (libsocket)

Listen for a connection on a socket
#include <sys/socket.h>
int listen(*socket***,** *backlog***)**
int *socket***, int** *backlog***;**

Function **listen()** "listens" for a connection on *socket*. It also signals the system your process's willingness to accept a connection on that socket. This function applies only to sockets of type **SOCK_STREAM** or **SOCK_SEQPACKET**.

socket is a file descriptor that identifies the socket in question. It must have been returned by a call to **socket()**. *backlog* defines the maximum length to which the queue of pending connections may grow. As of this writing, *backlog* is limited to a maximum of five. If a connection request arrives with the queue full, the client may receive an error with an indication of **ECONNREFUSED**; or if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

If all goes well, **listen()** returns zero. If an error occurs, it returns -1 and sets **errno** to an appropriate value. The following lists the errors that can occur, by the value to which **listen()** sets **errno**:

EBADF *socket* is not a valid descriptor.

ENOTSOCK
socket does not identify a socket.

EOPNOTSUPP
socket is not of a type that supports **listen()**.

Example

For an example of this function, see the Lexicon entry for **libsocket**.

See Also

accept(), connect(), libsocket, socket()

lmail — Command

Deliver mail on your local system

Command **lmail** delivers mail on your local system. It receives messages from the mail-routing program **smail**, and copies each into the appropriate user's mailbox.

See Also

commands, mail (overview), mail (command), rmail, smail

ln — Command

Create a link to a file
ln [-f] *oldfile newfile*
ln [-f] *oldfile ... directory*

The COHERENT system knows a file by its i-node number. Each file is also *linked* to one or more file names, each name being stored in a directory. This system means that the same file can be known by multiple names in multiple directories. The command **ln** lets you create a new link to a file.

In its first form, **ln** links the name *newfile* to the file that is already named *oldfile*, provided that *newfile* does not already exist.

In the second form, **ln** links *oldfile* with an identical name in another *directory*. In effect, one file will "live" in two directories.

If *newfile* already exists, **-f** forces **ln** to unlink it and assign its name to *oldfile*.

See Also

commands, cp, ls, mv, rm

Notes

Links across file systems are impossible. For example, if your COHERENT system has two file systems, one mounted on **/f** and the other mounted on **/usr**, you cannot use **ln** to link a file in **/v** to one in **/usr**.

Note, too, that **ln** lets you link a directory to another file. This feature is permitted by POSIX Standard; however, because COHERENT does not yet support symbolic links, this feature at best is useless, and at worst is rather dangerous. *Caveat utilitor.*

localtime() — Time Function (libc)

Convert system time to calendar structure
#include <time.h>
#include <sys/types.h>
struct tm ***localtime(***timep**)**
time_t **timep**;

localtime() converts COHERENT's internal time into the form described in the structure **tm**, which is defined in the header file **<time.h>**.

timep points to the system time. It is of type **time_t**, which is defined in the header file **<sys/types.h>**.

localtime() returns a pointer to the structure **tm**. The function **asctime()** turns **tm** into an ASCII string.

Unlike its cousin **gmtime()**, **localtime()** returns the local time, including conversion to daylight saving time, if applicable. The daylight-saving time flag indicates whether daylight saving time is now in effect, *not* whether it is in effect during some part of the year. Note, too, that the time zone is set by **localtime()** every time the value returned by

```
getenv("TIMEZONE")
```

changes. See the Lexicon entry for **TIMEZONE** for more information on how COHERENT handles time zone settings.

Example

The following example recreates the function **asctime()**. It builds a string somewhat different from that returned by **asctime()** to demonstrate how to manipulate the **tm** structure.

```
#include <time.h>
#include <sys/types.h>

char *month[] = {
      "January", "February", "March", "April",
      "May", "June", "July", "August", "September",
      "October", "November", "December"
};

char *weekday[] = {
      "Sunday", "Monday", "Tuesday", "Wednesday",
      "Thursday", "Friday", "Saturday"
};

main()
{
      char buf[20];
      time_t tnum;
      struct tm *ts;
      int hour = 0;

      time(&tnum);      /* get time from system */

      /* convert time to tm struct */
      ts=localtime(&tnum);

      if (ts->tm_hour == 0)
            sprintf(buf,"12:%02d:%02d A.M.",
                  ts->tm_min, ts->tm_sec);

      else
            if(ts->tm_hour>=12) {
                  hour=ts->tm_hour-12;
                  if (hour==0)
                        hour=12;
                  sprintf(buf,"%02d:%02d:%02d P.M.",
                        hour, ts->tm_min,ts->tm_sec);

            } else
                  sprintf(buf,"%02d:%02d:%02d A.M.", ts->tm_hour,
                        ts->tm_min,ts->tm_sec);

      printf("\n%s %d %s 19%d %s\n",
            weekday[ts->tm_wday], ts->tm_mday,
            month[ts->tm_mon], ts->tm_year, buf);

      printf("Today is the %d day of 19%d\n",
            ts->tm_yday, ts->tm_year);

      printf("Daylight Saving Time %s in effect\n",
            ts->tm_isdst ? "is" : "is not");
}
```

See Also

gmtime(), libc, time [overview], TIMEZONE
ANSI Standard, §7.12.3.4
POSIX Standard, §8.1

LEXICON

Notes

localtime() returns a pointer to a statically allocated data area that is overwritten by successive calls.

lockf() — General Function (libc)

Lock a file or a section of a file
#include <unistd.h>
int
lockf(*fd***,** *cmd***,** *size***)**
int *fd*, *cmd*; **long** *size*;

The COHERENT library function **lockf()** allows a process to lock part or all of a file. If another process calls **lockf()** on the same file to request a lock that conflicts with a previous lock, the later **lockf()** call returns an error or sleeps until the file is unlocked by the first process.

fd gives a file descriptor of an open file; the file must have been opened with **O_WRONLY** or **O_RDWR** permission if **lockf()** is to succeed.

size specifies how many bytes should be locked or unlocked. The lock begins at the current file position and extend forward (if *size* is positive) or backwards (if it is negative). A *size* of zero locks or unlocks the entire file starting from the current position.

cmd specifies the action **lockf()** is to take. **lockf()** recognizes the following four commands, as specified in the header file **<unistd.h>**:

F_TEST Test whether a lock has already been set upon the specified section of the file.

F_LOCK Lock a section of the file, if possible. If the section cannot be locked, sleep until it becomes available for locking.

F_TLOCK Lock a section of the file, if possible. Unlike **F_LOCK, F_TLOCK** does not sleep if the section cannot be locked; rather, it returns -1 and sets **errno** to **EAGAIN** if the lock is not available.

F_ULOCK Unlock a currently existing lock.

Use **lockf()** with the unbuffered I/O routines (**open()**, **write()**, and so on) rather than with standard I/O library routines (**fopen()**, **fprintf()**, **fwrite()**, and so on). The buffering used by the standard I/O library may cause unexpected behavior with file locking.

See Also

creat(), fcntl(), libc, open()

Diagnostics

lockf() returns zero on success, -1 on failure. On failure, it also sets **errno** to an appropriate value. Possible errors include the following:

EINVAL Invalid file descriptor.

EAGAIN Requested section is already locked.

EDEADLK A deadlock would occur if the command slept, or the system lock table is full.

Notes

See the Lexicon entry for **fcntl()** for a fuller description of the COHERENT system's method of file locking.

log() — Mathematics Function (libm)

Compute natural logarithm
#include <math.h>
double log(*z***) double** *z*;

log() returns the natural (base *e*) logarithm of its argument *z*.

Example

The following example is by Sanjay Lal (sanjayl@tor.comm.mot.com). It returns the amount of a quantity of radioactive material that remains after the passage of a period of time. Use it when planning your next nuclear dump. It takes three arguments: the amount of material, in kilograms; the half life, in years; and the time passed, in years. These can be decimal fractions.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

main(argc, argv)
int argc; char *argv[];
{
      double num, thalf, time;

      if (argc != 4) {
            fprintf(stderr,"Usage: %s amount halflife time\n", argv[0]);
            exit (EXIT_FAILURE);
      }

      num = atof (argv[1]);
      thalf = atof (argv[2]);
      time = atof (argv[3]);
      printf("%f\n", num * exp ( -log(2.0) * (time / thalf)));
}
```

See Also

log10(), libm
ANSI Standard, §7.5.4.4
POSIX Standard, §8.1

Diagnostics

When a domain error occurs (*z* is less than or equal to zero), **log()** sets **errno** to **EDOM** and returns zero.

log10() — Mathematics Function (libm)

Compute common logarithm
#include <math.h>
double log10(*z***) double** *z***;**

log10() returns the common (base 10) logarithm of its argument *z*.

Example

The following example, called **fact.c**, uses **log10()** and **pow()** to compute an approximation of the factorial of an integer. Compile it with the command:

```
cc -f fact.c -lm
```

It is by Brent Seidel (brent_seidel@chthone.stat.com).

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
      int num, loop, exponent;
      double sum, fraction;
      char buffer[50];

      fprintf(stderr, "Enter number to compute factoral of: ");
      fflush(stderr);
      if (gets(buffer) == NULL)
            exit(EXIT_FAILURE);

      num = atoi(buffer);
      for (sum = 0, loop = 2; loop <= num; loop++) {
            sum += log10((double) loop);
      }

      exponent = (int) sum;
      fraction = sum - exponent;
      printf("The factoral of %d is %g X 10^%d\n",
            num, pow(10.0, fraction), exponent);
}
```

See Also

log(), libm
ANSI Standard, §7.5.4.5
POSIX Standard, §8.1

Diagnostics

A domain error in **log10()** (*z* is less than or equal to zero) sets **errno** to **EDOM** and returns zero.

login — Command

Log in a user
login [-p] [*login_id* **[***environ_var***[=***value***] ...]]**

The command **login** allows a user to identify himself to your system. A user can invoke it as a command, or the system itself can invoke it (usually through the command **getty**) when a user attempts to log in.

You can invoke **login** as a command. To do so, return to your lowest-level (login) shell, then type either

```
login
```

or:

```
exec /bin/login
```

This invocation replaces the shell with **login**, and so ensures a smooth transition from one user account to another.

If the user does not supply a *login_id* on the **login** command line, **login** prompts him for the login identifier to use. If the account for *login_id* is protected by a password, **login** then asks the user to enter that password. If possible, **login** turns off echoing during the entry of the password to ensure that bystanders (or "kibitzers") cannot see the password displayed on his terminal.

Switches

login executes the file **/etc/default/login**. This file sets switches that control **login**'s behavior. Each switch has the form

*SWITCH***=***VALUE*

where *SWITCH* is the switch being set and *VALUE* is the value to which it is being set. **login** exports some of these switches as environmental variables, to give the programs that **login** invokes a minimal working environment.

login recognizes the following switches by default:

ALTSHELL
If set to **YES**, the login shell's name is recorded in the environment. If set to **NO**, it is not. By default, **login** sets this to **YES**.

CONSOLE
The allowable terminal devices (from **/dev**) from which the superuser **root** can log into your system. If this names more than one device, you must separated them with colons. If this variable is not set, then **root** can log in from any device. A device name can also include the wildcard character '?'.

HZ Your computer's clock tick frequency, in Hertz. **login** does not set a default. **login** exports this switch as an environmental variable.

IDLEWEEKS
The number of weeks before a login is disabled for lack of use. **login** does not set this variable.

NEWUSER
This switch gives a shell command that is to be executed when the file **$HOME/.lastlogin** does not exist. By default, it displays a warning message is displayed. The installation script for COHERENT typically creates a setting for you that executes the file **/etc/default/welcome** instead. This works with the command **/etc/newusr** to provide a "friendly" environment for users who are using COHERENT for the first time.

PASSREQ
If set to **YES**, every user must have a password. If set to **NO**, some users may log in without a password. By default **login** sets this to **YES**.

PATH This variable names the directories that an interactive shell searches for executable files. By default, **login** sets this to **/bin:/usr/bin**. **login** exports this switch as an environmental variable.

SUPATH
The default path for the superuser **root**. By default, **login** sets this to **/bin:/usr/bin**. **login** exports this switch as an environmental variable.

TIMEOUT
The time, in seconds, that **login** waits before it silently terminates and returns control to **getty**. **login** gives the user five "chances" to log in during this time. **login** by default sets this to 120.

TIMEZONE
The current time zone. This variable has the same format as the COHERENT environmental variable **TZ**: that is, it uses the template *NSTHNDT*, where *NST* is a three-character abbreviation for your local standard time (e.g., **CST** for Central Standard Time), *H* gives the number of hours difference between your time zone and Greenwich Mean Time, and *NSD* gives a three-character abbreviation for your local daylight-saving time. **login** exports this switch as an environmental variable.

Note that this variable is set for the benefit of code imported from UNIX. Most COHERENT commands use the environmental variable **TIMEZONE**, which much more detailed information about your local time zone. For details on **TIMEZONE**, see its entry in the Lexicon.

Note, too, that the variable **TZ**, which is set in file **/etc/timezone**, should be set to exactly the same string as **/etc/default/TIMEZONE**; otherwise, much confusion will result.

ULIMIT
The maximum size, in 512-byte blocks, of a file that the user can create. **login** does not set a default. At present, COHERENT ignores this option.

UMASK
This gives the permissions that a shell sets by default for files that the user cretaes. **login** does not set a default value for this variable. **login** exports this switch as an environmental variable.

Logging Failed Attempts

If the user attempts and fails five times to log in, **login** records the erroneous attempts in file **/usr/adm/loginlog** (should that file exist), and it disables the terminal for a period of time. (Note that previous versions of COHERENT recorded failed attempts in file **/usr/adm/failed**.) **login** does not record when the user typed only `(¢)` in response to a prompt for a login identifier. If the user does not succeed in logging in within two minutes (120 seconds), **login** silently disconnects the terminal and returns control of the device to **getty**.

Restrictions on Logging In

If the file **/etc/nologin** exists, **login** refuses to let any users login in, except for the superuser **root** and the (presumably few) users named in file **/etc/trustme**. You can use this mechanism to stop users from logging in at an inopportune time, e.g., when the system is about to be shut down. In response to an attempt to log in, **login** displays the contents of that file, which should contain the system administrator's explanation of why logins are not permitted at that time.

login also reads file **/etc/usrtime**, if it exists. This file gives user identifiers; for each identifier, it gives the tty line from which that user can log in, and the day of the week and time of day during which that user can log in. **login** rejects the user's login if it is from a tty line forbidden to the user, or outside the day and time permitted. If a user's login identifier is not in this file, **login** assumes that that user can log in from any line and at any time. Additional options allow you to control globally all users, or interactive users, UUCP accounts, or SLIP users.

Passwords

login prompts the user for a password when he logs in. **login** takes its copy of the user's password from file **/etc/passwd**. If the password consists of a single asterisk '*', then **login** reads the password from file **/etc/shadow**. This file should be legible only by the superuser **root**. Once the passwords are in **/etc/shadow**, they can be read only by processes that have **root**-level permissions, such as **login**. This protects the encrypted passwords from being read by ordinary users, and perhaps decrypted by a "cracker." For details, see the Lexicon entry for **shadow**.

Note that if a user's password consists of '*' and file **/etc/shadow** does not exist, **login** assumes that the user's password encrypts to '*'. This effectively locks the user out of his account. The lesson is not to remove or modify **/etc/shadow** capriciously.

LEXICON

In addition, **login** reads the files **/etc/dialups** and **/etc/d_passwd**, which hold auxiliary passwords. You can set auxiliary passwords for users on selected tty lines to provide additional security. For details, see these files' entries in the Lexicon.

Success In Logging In

If the user succeeds in logging in, **login** displays on his terminal the date and time that he last logged in, as recorded in file **$HOME/.lastlogin**. **login** updates this file whenever the user logs in. If this file had been modified by a process other than **login**, **login** warns the user of a possible breach in his account's security.

login then prints the contents of the file **/etc/motd**, which holds the message of the day. It also sets the environmental variable **LOGNAME** to the user's login identifier.

As its last action, **login** invokes the user's shell, as set in the last field of his entry in **/etc/passwd**. Under COHERENT, this is either the Bourne shell **sh** or the Korn shell **ksh**. (**login** can also invoke a program in place of a shell, e.g., the command **uucico** for a UUCP account.) If **login** invokes an interactive shell, it does so with the first character of its **argv[0]** set to '-', so that the shell knows that it is a login shell. (For example, if **login** invokes **ksh**, its **argv[0]** is **-ksh**.)

When a shell starts up, it executes the script **/etc/profile**. This script executes the command **umask**, to set the permissions that the shell gives by default to files that that user creates; and then sets the following environmental variables:

HZ The default clock speed for your system. By default, COHERENT sets this to 100.

LOGNAME
The user's login identifier.

MAIL This names the user's mailbox. By default, it is set to **/usr/spool/mail/***login_id*.

PAGER The command used to "page" through files of text. By default, COHERENT sets this to **more**.

PATH The directories that the shell searches for executable files. By default, COHERENT sets these to **/bin** and **/usr/bin**.

TERM The type of terminal at which the user is working. By default, COHERENT reads file **/etc/ttytype** to read the default terminal type for a given port. For details, see the description of this command in the Lexicon.

Finally, **/etc/profile** calls the script **/etc/timezone**, which sets the following environmental variables:

TZ Your time zone, as interpreted by most UNIX software.

TIMEZONE
Your time zone, as interpreted by the COHERENT system. At present, it contains considerably more information about your time zone than does **TZ**. For details of this variable, see its description in the Lexicon.

The shell then executes the script **$HOME/.profile**, should one exist. The COHERENT command **newusr** creates this file when it installs a new user. The user can edit this file to set environmental variables, and to invoke commands for his amusement, e.g., **/usr/games/fortune**.

Command-line Options

If a user invokes **login** as a command, he can set one or more environmental variables on **login**'s command line. If *environ_var* contains an equal sign, then it and *value* are placed into the environment. If *environ_var* does not contain an equal sign, then **login** places it into the environment with the format:

environ_var=*n*

where *n* is a number from zero through the number of environmental variables being so set.

For security reasons, **login** refuses to set from its command line any of the following environmental variables:

CDPATH	**HOME**
HZ	**IFS**
LOGNAME	**MAIL**
PATH	**SHELL**
TZ	

login also recognizes the command-line option **-p**, which tells **login** to preserve the user's current environment when logging in as *login_id*. If it is *not* invoked with this option, **login** "empties" the current user's before it constructs the environment for user *login_id*. If it is invoked with this option **login** replaces existing environmental variables with those it sets during the login process, but it preserves all other environmental variables set in the

original environment.

Subsystem Logins

login supports virtual "subsystems" under COHERENT. If the user's shell as specified in **/etc/passwd** is '*', then **login** makes the user's **HOME** directory into the system's root directory, informs the user that it is executing a "Subsystem login," and then re-executes **login**. The new root directory must have its own versions of the commands **/etc/passwd**, **/bin/login**, and **/dev** files. Once so logged in, the user has, in effect, his own virtual version of the COHERENT system.

Files

/etc/d_passwd — Passwords for shells on dialup lines
/etc/default/login — Default parameters for **login**
/etc/dialups — List of dialup tty lines
/etc/group — File that defines user groups
/etc/nologin — Forbid all logins
/etc/passwd — Password file
/etc/profile — Script executed by **sh** and **ksh** upon invocation
/etc/shadow — Optional file of "shadow" passwords
/etc/trustme — Permit named users to log in despite **nologin**
/etc/ttytype — Default terminal type on a given tty line
/etc/utmp — Identifiers of users who are logged into your system
/etc/usrtime — Login restrictions for user *login_id*
/etc/wtmp — History of who has logged in, and when
/usr/adm/loginlog — Record of failed login attempts
/usr/spool/mail/*name* — Mailbox for *user*
$HOME/.lastlogin — Date of user's last login

See Also

Administering COHERENT, commands, ksh, lastlogin, mail, sh, newgrp, newusr, welcome

Notes

This version of **login** no longer recognizes the remote-access account **remacc**. To duplicate the function of this account, set the files **/etc/dialups** and **/etc/d_passwd**. For details, see their entries in the Lexicon.

This version of **login** was written by Tony Field (tony@ajfcal.cuc.ab.ca), with help from Uwe Doering (gemini@geminix.in-berlin.de). It was ported to COHERENT by Harry Pulley (hcpiv@snowhite.cis.uoguelph.ca), with help from Udo Munk (udo@umunk.gun.de).

login — System Administration

Set default values for logging in
/etc/default/login

The command **login** reads the file **/etc/default/login**, which gives **login**'s default settings. These settings dictate some of **login**'s behaviors. **login** exports some settings as environmental variables, to help control the behavior of some other programs within COHERENT.

For a list of the settings normally set by **/etc/default/login**, see the Lexicon entry for the command **login**.

See Also

Administering COHERENT, login

loginlog — System Administration

Log of failed login attempts
/usr/adm/loginlog

File **/usr/adm/loginlog** logs all failed attempts to log in. **login** places an entry into this log either when a login attempt does not succeed within the number of seconds set by the environmental variable **TIMEOUT** (default, 120), or fails to log in correctly with five times within that time.

See Also

Administering COHERENT, login

Notes

Earlier implementations of **login** logged failed attempts in file **/usr/adm/failed**.

logmsg — System Administration

Hold COHERENT Login Message
/etc/logmsg

The file **/etc/logmsg** holds the message that COHERENT displays to prompt the user to log in. The superuser **bin** can edit this message to whatever she prefers.

See Also

Administering COHERENT

Notes

The default message consists of the bell character **<ctrl-G>** followed by the text **Coherent login:**. If the bell annoys you, simply delete the **<ctrl-G>** from **/etc/logmsg**.

LOGNAME — Environmental Variable

Name user's identifier
LOGNAME=*user_identifier*

The environmental variable **LOGNAME** names your login identifier. For example, if your login identifier is **fwb**, then by typing **set** you will see the entry **LOGNAME=fwb**. **LOGNAME** is set in **/etc/profile**.

See Also

environmental variables, ksh, login, sh, USER

long — C Keyword

Data type

A **long** is a numeric data type. The ANSI standard states that **long** is the largest integer data type. It cannot be smaller than an **int**, although an **int** and a **long** can be the same size.

COHERENT defines an **long** to be four bytes long; that is, **sizeof long** equals 4 (four **char**s, or 31 data bits plus a sign bit). A **long** can hold any value from -2,147,483,647 to 2,147,483,647.

A **long** normally is sign extended when cast to a larger data type; an **unsigned long**, however, will be zero extended.

See Also

C keywords, data formats, int
ANSI Standard, §6.1.2.5

longjmp() — General Function (libc)

Perform a non-local goto
#include <setjmp.h>
int longjmp(*env***,** *rval***)**
jmp_buf *env***; int** *rval***;**

The function call is the only mechanism that C provides to transfer control between functions. This mechanism is inadequate for some purposes, such as handling unexpected errors or interrupts at lower levels of a program. To answer this need, **longjmp** provides a non-local *goto*.

longjmp() restores an environment that had been saved by a previous call to the function **setjmp()**. It returns the value *rval* to the caller of **setjmp()**, just as if the **setjmp()** call had just returned. Note that **longjmp()** must not restore the environment of a routine that has already returned. The type declaration for **jmp_buf** is in the header file **setjmp.h**. The environment saved includes the program counter, stack pointer, and stack frame. These routines do not restore register variables in the environment returned.

Example

For an example of this function, see the entry for **setjmp()**.

See Also

libc, setjmp(), siglongjmp()
ANSI Standard, §7.6.2.1
POSIX Standard, §8.1

Notes

Programmers should note that many user-level routines cannot be interrupted and reentered safely. For that reason, improper use of **longjmp()** and **setjmp()** can result in the creation of mysterious and irreproducible bugs. Do not attempt to use **longjmp()** within an exception handler.

look — Command

Find matching lines in a sorted file
look [-df] *string* [*file*]

The command **look** scans the sorted *file* and prints each line that begins with *string*.

The following options specify the order of the search:

-d Use dictionary order: the only characters tested are alphanumerics and blanks.

-f Convert all alphabetic characters to upper case.

If no *file* is specified, **look** uses **/usr/dict/words** with the **-df** option.

Example

For an example of how to use this command, see the entry for **spell**.

Files

/usr/dict/words — File of words (sorted with **sort -df**).

See Also

commands, sort

Notes

Because the file **/usr/dict/words** is quite large, you may not have installed it or uncompressed it when you installed your COHERENT system. If this is the case, **look** will not work correctly.

lp — Command

Spool a file for printing
lp [-d*printer***] [-t** *title***] [-n***copies***] [-R** *page* **[***page***]] [-Smws]** *file ...*

The command **lp** spools text for printing. If you name no *file* on its command line, **lp** spools what it receives from the standard input.

lp prefaces the spooled text with a header that describes, among other things, on what device you want to print the text; then it copies the text into directory **/usr/spool/mlp/queue**, where it remains until it is removed by the printer daemon **lpsched**. The spooled text, which may comprise multiple files, plus its header is called a *job*.

The following describes the header with which **lp** prefaces each file:

Offset	*Length*	*Description*
0	14	User who spooled the file
14	14	Name of the printer on which to print file
28	10	Type of file (application specific)
38	3	Length of output page (default, 66 lines)
41	4	Number of pages (maximum, 9,999)
45	2	Number of copies to print (default, one; maximum, 99)
47	1	Set life expectancy of job (see below)
48	1	If 'M', send user mail after printing
49	1	If 'W', write user after printing
50	14	Name of data base (application specific)
64	14	Name of program (application specific)
78	10	Date/time stamp (no. of seconds since 1/1/1970)
88	60	Description or title

Note that the fields marked "application specific" are not use by **lp** or **lpsched**. Rather, they are available to applications, such as filters, that may be used with **lp** to print files.

The "life expectancy" byte of the header defines how long the job remains alive in **/usr/spool/mlp/queue**. Jobs labeled **T** (temporary) live for 30 minutes after being spooled; those labeled **S** (short-term) live for 24 hours; and those labeled **L** (long-term) live for 72 hours. Once a job's life expectancy has expired, the printer daemon **lpsched** removes it. The default is life expectancy is **S**. To change the life expectancy of a job, use the command **chreq**. You can also change the above default "lifetimes" by editing the file **/usr/spool/mlp/controls**.

When **lp** creates a job, it gives the job a seven-character name. The name's first character gives the status of the job: **R** indicates that the file is being printed or is pending printing, whereas **r** indicates that the job has already been printed. The second character gives the job's priority status, from 0 through 9: zero gives highest priority, nine the lowest. The default priority is 2. The last five characters of the name give a zero-padded sequence number. To change a job's status or priority, use the command **chreq**; or the system administrator can alter either simply by renaming the file.

lp recognizes the following options:

-R *request* Print a job beginning from the first *page* and continuing either to the second *page* or to the end of the document (if no second *page* is specified). Note that the printer daemon **lpsched** identifies pages by counting lines of input, so this feature works only with straight text. It does *not* work correctly with "cooked" input, such as files of PostScript or PCL.

-S9 Shut down the spooler daemon **lpsched**.

-d*printer* Print the job on *printer*.

-m Send mail to the user once the spooled job has been printed.

-n*copies* Print *copies* copies of the job.

-s Silent — do not acknowlege submissions. Normally, **lp** writes on the standard output the sequence number of the job you just spooled. You can use that number to remove or abort a job, or otherwise manipulate it.

-t *title* Give this job *title*. This is the title that appears in the queue displayed by the command **lpstat**.

-w Write a message on the user's screen once the job has been printed.

lp sends you mail if one of your print jobs failed due to an error.

For more information on **lp** and its related commands, see the Lexicon entry **printer**.

See Also

chreq, commands, controls, lp [device driver], lpadmin, lpsched, pclfont, printer

Notes

Because most users find banners annoying rather than helpful, **lp** does not print banners. It ignores the option **-b**, which under orthodox implementations of **lp** prints a banner page. Applications that desire a banner page should make provision for it in the individual printer's control file. For details, see the Lexicon entry for the command **lpadmin**.

If you wish to use **lp** to download a PCL bitmapped font to your PCL printer, you must first process the font with the command **pclfont**. For details, see its Lexicon entry.

lp — Device Driver

Driver for parallel ports
/dev/lpt*N*

The device driver **lp** drives the parallel ports. It has major number 3.

This driver follows the IBM PC standard in that it can only send data out the port — it cannot receive data from the port.

The following script lets you install or de-install the parallel-port driver: To install or de-install a parallel printer, log in as the superuser **root**; then execute the following script:

```
cd /etc/conf
lp/mkdev
/conf/mlpconfig
bin/idmkcoh -o /kernel_name
```

kernel_name should name the new kernel to build. Then reboot to invoke the newly built *kernel_name*.

See Also

device drivers, printer

lpadmin — Command

Administer the lp print-spooler system
lpadmin [-d*printer***]**
lpadmin [-p*printer***] [-v***device***] [-m***backend***]**
lpadmin [-x*printer***]**

The command **lpadmin** administers the **lp** print-spooling system.

Under the **lp** spooler, the system administrator gives each printer a name. She also establishes a script for each class of printers; for example, she would prepare one script for all Epson printers, and another for all PostScript printers. The script lists commands that must be executed to print the text properly, such as setting the port into the correct mode or post-processing the text; Finally, she inserts into file **/usr/spool/mlp/controls** an entry that links a printer by name with its device and its script. When a user spools a job for printing, she selects the printer by name. (She can also use the command **route** establish a default printer for herself.) The print spooler **lpsched** reads the information established by the administrator to ensure that printing is managed correctly.

The command **lpadmin** is designed to make it easy for you to perform these tasks of administration. With **lpadmin**, you can add a new printer to your COHERENT system and link it to a device and a description script. You can also add or modify a description script, or drop a printer. **lpadmin** recognizes the following options:

-d*printer* Make *printer* the default printer for your system. This is the printer that is used when a user names no printer on the **lp** command line and has set no default printer for herself.

-m*script* Use *script* to preprocess all text sent to a given printer. *script* is stored in directory **/usr/spool/mlp/backend**. This option is always used with option **-p**.

-p*printer* Select *printer* for definition or change. This option is used with the options **-m** and **-v**.

-v*device* Associate *device* (a serial or parallel port) with the printer named in the option **-p**.

-x*printer* Remove *printer* from the system.

For detailed examples of how to modify the file **controls** and how to build a control script for a printer, see the Lexicon entry for **controls**.

See Also

commands, controls, lp, printer

lpd — System Administration

Spooler daemon for line printer
/usr/lib/lpd

lpd is the daemon that prints listings queued by the command **lpr**. All jobs are printed on the printer that is accessed through device **/dev/lp**. For information on this device, and on printer management in general, see the Lexicon entry **printer**.

lpr invokes **lpd** automatically. If there is no printing to do, or if another daemon is already running (indicated by the file **dpid**), **lpd** exits immediately. Otherwise, it searches the spool directory for control files of listings to print. A control file contains the names of files to print, the user name, banners, and files to be removed upon completion.

lpd does not print listings in any particular order. Priority is not given to any file, either by size or by requester.

Files

/dev/lp — Printer
/usr/spool/lpd — Spool directory

/usr/spool/lpd/cf* — Control files
/usr/spool/lpd/df* — Data files
/usr/spool/lpd/dpid — Lock and process id

See Also

Administering COHERENT, init, lpr, lpskip, printer

Notes

Beginning with release 4.2, COHERENT also includes the printer daemon **despooler**, which prints files spooled with the command **lp**. For details on how COHERENT manages printing, see the Lexicon entry for **printer**.

lpioctl.h — Header File

Definitions for line-printer I/O control
#include <sys/lpioctl.h>

lpioctl.h defines constants used by routines that control I/O on the line printer.

See Also

header files

lpr — Command

Spool a job for printing on the line printer
lpr [-cmnr] [-b *banner***] [***file ...***]**

The command **lpr** spools each *file* for printing on the line printer. If no *file* is named on the command line, **lpr** spools what it reads from the standard input.

lpr recognizes the following options:

-B Suppress printing of a banner.

-b *banner* Print *banner* on the banner page. The default banner is the user's login name.

-c Copy each *file* into the spooling directory, instead of reading the file from its home directory. This option lets you change a *file* before it has finished printing.

-m Write a message on the user's terminal when printing completes.

-n Do not send a message (default).

-r Remove the files when they have been spooled.

The command **lpskip** aborts or restarts printing of the file that is currently being printed. The command **epson** converts the output of **nroff** into a form usable by Epson-compatible dot-matrix printers.

Files

/dev/lp — Line printer
/usr/lib/lpd — Line printer daemon
/usr/spool/lpd — Spool directory
/usr/spool/lpd/dpid — Daemon lockfile

See Also

commands, hpr, lp, lpd, lpskip, printer

Notes

Beginning with release 4.2, COHERENT also includes the **lp** print spooler. **lp** offers a more sophisticated way to manage printers, especially on machines that support multiple printers of the same type. For details, see the Lexicon entries for **printer** and **lp**.

lpsched — Command

Print jobs spooled with command lp; turn on printer daemon
lpsched

The daemon **lpsched** prints jobs spooled with the command **lp**.

Typing the command **lpsched** by itself launches the daemon. The rest of this article describes how **lpsched**

manages printing.

Each file in directory **/usr/spooler/mlp/queue** is a print job spooled by the command **lp**. When **lp** spools a job, it copies the input (usually a file) into the spool directory and appends to the beginning of each job a 192-byte header that indicates how the job is to be printed. This header includes such information as the name of the printer on which to print the job and the date and time the job was spooled. For a detailed layout of this header, see the Lexicon entry for **lp**.

lp also assigned each job a seven-character name. The first character is **R** or **r**: the former indicates that the job is either being printed or is awaiting printing; whereas the latter indicates that the job has been printed. The second character is a digit, from zero to nine, that sets the job's priority: zero gives highest priority, nine lowest. (The default priority is **2**.) The last five characters give a zero-padded identification number.

lpsched awakens every 30 seconds or whenever the command **lp** spools a job for printing. **lpsched** then processes each file in **/usr/spooler/mlp/queue**. It reads each job each that is awaiting printing, the order being dictated first by the priority code and then by the identification number (which indicates the order in which the jobs were spooled).

When **lpsched** actually prints a job, it performs the following tasks:

- First, it opens the file that contains the job to be printed, and reads its header. The header gives the number of the job; the name of the user who spooled the job; the name of the printer device upon which the job is to be printed; the number of copies to print; and the title of the job, as set with the **lp** option **-t**. (NB, do not confuse an MLP "device," which is set in the file **/usr/spool/mlp/controls**, with the physical device into which the printer is plugged.)
- **lpsched** then finds the entry in file **/usr/spool/mlp/controls** that describes the printer device the user requested. An entry in **controls** is of the form

  ```
  printer = banner, /dev/hp, make_banner
  ```

  In this case, the MLP device is named **banner**; the output is to be printed on physical device **/dev/hp**; and the output is to be filtered through backend script **make_banner**, which is a script kept in directory **/usr/spool/mlp/backend**. (For details on how to describe an MLP printer device, see the Lexicon entry for **controls**).
- If the entry for this device does not name a backend script, **lpsched** copies the body of the job (that is, the text that you had spooled) without modification to the device by which the printer is accessed.
- If the entry for this device does name a backend script, **lpsched** invokes the script and redirects its output to the physical device.

When **lpsched** invokes a backend script, it passes it four arguments: (1) the number of the job to be printed, (2) the login identifier of the user who spooled the job, (3) the number of copies to be printed, and (4) the title of the job. The script can ignore these arguments, or use them in its filtration process; for example, it can use the fourth argument to construct a banner page that is printed before the job. For examples of backend script that perform various types of sophisticated processing, see the Lexicon entry for **controls**.

lpsched uses a system of lock files to ensure that each device is accessed in a disciplined manner. For details on COHERENT's system of building lock files, see the Lexicon entries for **UUCP** and **libmisc**.

To abort the printing of a job, invoke the command **cancel**. Note that this only affects jobs that are being spooled or waiting to be spooled. If a job has been downloaded to a printer, the only way to abort printing is to manipulate the printer itself through its front panel and switches.

When a job has printed successfully, **lpsched** changes the status character in its name to **r**. A file remains in the spool directory until its "lifetime" has expired. You can reprint a quiescent file by invoking the command **reprint**. To change a job's target printer, priority, or lifetime, use the command **chreq**. For details on these commands, see their Lexicon entries.

lpsched awakens whenever you use the command **lp** to spool a job for printing. It also awakens every five minutes, whether or not a job has been spooled, to see if anything needs to be printed and check quiescent files.

After it has processed every job that awaits printing, **lpsched** reads the header of every quiescent file. If a file's "lifetime" has expired, **lpsched** removes it. A file with a *temporary* lifetime survives 30 minutes after spooling; one with a *short-term* lifetime survives 24 hours; and one with a *long-term* lifetime survives 72 hours. You can change these defaults by editing **controls**; for details, see its entry in the Lexicon. By default, a job is given a short-term life expectancy. To change a job's life expectancy, use the command **chreq**.

LEXICON

The command **lpsched** turns on **lpsched**; and command **lpshut** turns it off. Jobs spooled while **lpsched** is turned off remain spooled until **lpsched** is reawakened.

See Also

Administering COHERENT, commands, lp, lpshut, printer

lpshut — Command

Turn off the printer daemon despooler
lpshut [-d]

The command **lpshut** turns off the printer daemon **lpsched**.

The option **-d** tells **lpshut** to finish the jobs that are currently printing before it shuts down the daemon. If jobs are interrupted while printing, the printer daemon **lpsched** reprints them when it restarts.

See Also

commands, lp, lpsched, printer

lpskip — Command

Abort/restart current job on line printer
lpskip [-r]

The command **lpskip** aborts or restarts the file being printed on the printer plugged into device **/dev/lp**. By default, it aborts the job and prints a message on the user's terminal.

When invoked with the **-r** option, **lpskip** restarts the printing of the current job. This is useful when a printing is spoiled due to, say, a paper jam.

lpskip works only with files that have been spooled with the command **lpr**.

Files

/usr/lib/lpd — Line printer daemon
/usr/spool/lpd — Spool directory
/usr/spool/lpd/dpid Daemon lockfile

See Also

commands, lpd, lpr, hpskip

Notes

To cancel jobs spooled with the command **hpr**, use the command **hpskip**. To cancel or reprint jobs spooled with the command **lp**, use the commands **cancel** and **reprint**. See the Lexicon entry **printer** for details.

lpstat — Command

Give status of printer or job
lpstat [-p*printer***] [-drqstv]**

The command **lpstat** gives information about the operation of the **lp** print-spooling mechanism. It recognizes the following options:

-p *printer*
Give the status of *printer*.

-d Name the system's default printer.

-r Give the status of the daemon **lpsched**.

-q Give a detailed report of jobs in the queue. The jobs are displayed in two groups, quiescent and active, with each group ordered by their priority — which, given **lp**'s conventions for naming jobs, is identical with their alphabetical order.

-s Summarize status of each request and status of each printer.

-t Like option **-s**, but in somewhat more detail.

-v List all available printers and the devices associated with them.

See Also

commands, lp, printer

lr — Command

List subdirectories' contents in columnar format
lr [*file ...***]**

lr is a link to the command **ls -CR**. It prints each *file* in columnar format, like the command **lc**. If a *file* is a directory, **lr** also prints its contents and that of each of its subdirectories. If no *file* is named, it lists the contents of the current directory by default.

See Also

commands, l, lc, lf, ls, lx

lrand48() — Random-Number Function (libc)

Return a 48-bit pseudo-random number as a non-negative long integer
long lrand48();

Function **lrand48()** generates a 48-bit random number, then returns its high 31 bits in the form of a non-negative **long**. The value returned is (or should be) uniformly distributed throughout the range of zero through 2^31.

See Also

libc, srand48()

ls — Command

List directory's contents
ls [-abCcdFfgilmnopqRrstux] [*file ...* **]**

The command **ls** prints information about each *file*. Normally, **ls** sorts its output by file name and prints only the name of each *file*. If a directory name is given as an argument, **ls** sorts and lists its contents, not including '.' and '..'. If no *file* is named, **ls** lists the contents of the current directory.

The following options control how **ls** sorts and displays its output:

-a Print all directory entries, including '.', '..', any hidden files, and volume ID's.

-b Print non-graphic characters in octal.

-C Print the output in multi-column format, sorted down the columns.

-c Print the time the files' attributes were last changed.

-d Treat directories as if they were files.

-F Print a '/' after the name of each directory, and print an '*' after each executable file.

-f Force each argument to be treated as a directory. This disables the **-lrst** options and sorting, and enables the **-a** option.

-i Print the i-number of each file.

-l Print information in long format. The fields give mode bits, link count, owner uid, owner gid, size in bytes, date, and file name. For special files, major and minor device numbers replace the size field.

-m "Stream" the output horizontally across the screen, with each file name separated by a comma.

-n Same as **-l**, except the group identifiers and user identifiers are numbers rather than names.

-o Same as **-l**, except that the group id is not printed.

-p Print a '/' after each directory name.

-q Print non-graphics characters as '?'.

-r Reverse the sense of the sort.

-R Recursively print directories.

-s Print the size in blocks of each file.

-t Sort by time, newest first.

-u Sort by the *access* time.

-x Print multicolumn output, sorted across the columns. This resembles the output of the command **lc**.

The date **ls** prints with the **-l** and **-t** options is the *modification* time, unless the **-c** or **-u** option is used as well.

The mode field in the long list format consists of ten characters. The first character will be one of the following:

- Regular file
b Block special file
c Character special file
d Directory
p Pipe
x Bad entry (remove it immediately!)

The remaining nine characters are permission bits, in three sets of three characters each. The first set pertains to the owner of the file, the second to users from the owner's group, and the third to users from other groups. Each set may contain three characters from the following.

r The file can be read
s Set effective user ID or group ID on execution
t Shared text is sticky
w The file can be written
x The file is executable
- No permission is given

Links

COHERENT includes several commands that are links to **ls** and its options, to make it easier for you to use the various features of **ls**. The following table gives each command and the form of **ls** to which it is linked:

l **ls -l**
lf **ls -CF**
lr **ls -CR**
lx **ls -x**

See Also

chmod, commands, l, lc, lf, lr, lx, stat

lseek() — System Call (libc)

Set read/write position
#include <unistd.h>
long lseek(*fd***,** *where***,** *how***)**
int *fd***,** *how***; long** *where***;**

lseek() changes the *seek position*, or the point within a file where the next read or write operation is performed. *fd* is the file's file descriptor, which is returned by **open()**.

where and *how* describe the new seek position. *where* gives the number of bytes that you wish to move the seek position. It is measured from the beginning of the file if *how* equals **SEEK_SET** (zero), from the current seek position if *how* equals **SEEK_CUR** (one), and from the end of the file if *how* equals **SEEK_END** (two). A successful call to **lseek()** returns the new seek position. For example,

```
position = lseek(fd, 100L, SEEK_SET);
```

moves the seek position 100 bytes past the beginning of the file; whereas

```
position = lseek(fd, 0L, SEEK_CUR);
```

returns the current seek position and does not change the seek position at all.

You can create a *sparse file* by seeking beyond the current size of the file and writing. The "hole" between the end of the file and where the write occurs is read as zero and will occupy no disk space. For example, if you **lseek()** 10,000 bytes past the current end of file and write a string, the data will be written 10,000 bytes past the old end of file and all intervening matter will be considered part of the file.

lseek() differs from its cousin **fseek()** in that **lseek()** is a system call and uses a file descriptor, whereas **fseek()** is a C function and uses a **FILE** pointer.

If all goes well, **lseek()** returns the new seek position. If an error occurs, such as seeking to a negative position, **lseek()** returns -1L and sets **errno** to an appropriate value.

See Also

libc, unistd.h
POSIX Standard, §6.5.3

Notes

lseek() is permitted on character-special files, but drivers do not generally implement it. As a result, seeking a terminal will not generate an error but will have no discernible effect.

ltol3() — General Function (libc)

Convert long integer to file system block number
ltol3(*l3p, lp, n***)**
char **l3p*;*
long **lp*;*
unsigned *n*;

To conserve space inside i-nodes in COHERENT file systems, the system stores block addresses in three bytes. Programs that reference or maintain file systems use the functions **l3tol()** and **ltol3()** to convert between the three byte representation and **long** numbers.

ltol3() converts *n* **long** integers at address *lp* to the more compact form at address *l3p*.

See Also

libc

lvalue — Definition

An **lvalue** is an expression that designates a region of storage. The name comes from the assignment expression **e1=e2;**, in which the left operand must be an lvalue.

An identifier has both an *lvalue* (its address) and an *rvalue* (its contents). Some C operators require lvalue operands; for example, the left operand of an assignment statement must be an lvalue. Some operators give lvalue results; for example, if *e* is a pointer expression, ***e* is an lvalue that designates the object to which *e* points.

A *variable* can be used as an lvalue, whereas a constant cannot. For example, you cannot say

```
6 = (foo+bar);
```

A pointer is a variable, and can be manipulated within limits. An array name, however, is a constant and cannot be altered legally. Thus, the code

```
int foo[10];
int *bar;
foo = bar;
```

will generate an error message when you attempt to compile it, whereas

```
int foo[10];
int *bar;
bar = foo;
```

will not.

The following example shows the use of both an lvalue and a rvalue:

```
int i, *ip;

ip = &i;        /* ip is an lvalue, i and &i are rvalues */
i = 3;          /* i is an lvalue, 3 is an rvalue */
*ip = 4;        /* *ip is an lvalue, 4 is an rvalue */
```

See Also

Programming COHERENT, rvalue
ANSI Standard, §6.2.2.1

lx — Command

List directory's contents in columnar format
lx [*file ...***]**

lx is a link to the command **ls -x**. It prints each *file* in columnar format, like the command **lc**, except that directories and file names are printed together in one listing. If a *file* is a directory, **lx** lists its contents. If no *file* is named, **lx** lists the contents of the current directory by default.

See Also

commands, l, lc, lf, lr, ls

m4 — Command

Macro processor
m4 [file ...]

The command **m4** processes macros. It allows you to define strings for which **m4** is to search, and strings to replace them; **m4** then opens *file*, reads its contents, replaces each macro with its specified replacement string, and writes the results into the standard output stream.

m4 can also manipulate files, make conditional decisions, select substrings, and perform arithmetic. The tutorial *Introduction to the m4 Macro Processor* introduces **m4** in detail.

m4 reads its *file*s in the order given; if no *file* is named, then it reads the standard input stream. The file name '-' indicates the standard input.

m4 copies input to output until it finds a potential *macro*. A macro is a string of alphanumerics (letters, digits, or underscores) that begins with a non-digit character and is surrounded by non-alphanumerics. If **m4** does not recognize the *macro*, it simply copies it to the output and continues processing. If **m4** recognizes the *macro* and the next character is a left parenthesis '(', an *argument set* follows:

```
macro(arg1,..., argn)
```

The arguments are collected by processing them in the same manner as other text (thus, an arguments may itself be another macro), and resulting output text is diverted into storage. **m4** stores up to nine arguments; any more will be processed but not saved. An argument set consists of strings of text separated by commas (commas inside quotation marks or parentheses do not terminate an argument), and must contain balanced parentheses that are free of quotation marks (i.e., that are *unquoted*). **m4** strips arguments of unquoted leading space (blanks, tabs, newline characters).

m4 then removes the *macro* and its optional argument set from the input stream, processes them, and replaces them in the input stream with the resulting value. The value becomes the next piece of text to be read.

Quotation marks, of the form ` ', inhibit the recognition of *macro*. **m4** strips off one level of quotation marks when it encounters them (quotation marks are nestable). Thus, `*macro*' is not processed, but is changed to *macro* and passed on.

m4 determines the *value* of a user-defined macro by taking the text that constitutes the macro's *definition* and replacing any occurrence within that text of '$*n*' (where *n* is '0' through '9') with the text of the *n*th argument. Argument 0 is the *macro* itself.

m4 recognizes the following predefined macros:

changequote[([*openquote***],[***closequote***])]**
: Changes the quotation characters. Missing arguments default to ` for open or ' for close. Quotation characters will not nest if they are defined to be the same character. Value is null.

decr[(*number***)]**
: Decrement *number* (default, 0) by one and returns resulting value.

define(*macro,definition***)**
: Define or redefine *macro*. If a predefined macro is redefined, its original definition is irrecoverably lost. Value is null.

divert[(*n***)]**
: Redirects output to output stream *n* (default is zero). The standard output is zero, and one through nine are maintained as temporary files. Any other *n* results in output being thrown away until the next **divert** macro. Value is null.

LEXICON

divnum
Value is current output stream number.

dnl Delete to newline: removes all characters from the input stream up to and including the next newline. Value is null.

dumpdef[(*macros***)]**
Value is quoted definitions of all *macro*s specified, or names and definitions of all defined macros if no arguments.

errprint(*text***)**
Print *text* on standard error file. Value is null.

eval(*expression***)**
Value is a number that is the value of evaluated *expression*. It recognizes, in order of decreasing precedence: parentheses, **, unary + -, * / %, binary + -, relations, and logicals. Arithmetic is performed in **long**s.

ifdef(*macro,defvalue,undefvalue***)**
Return *defvalue* if *macro* is defined, and *undefvalue* if not.

ifelse(*arg1,arg2,arg3...***)**
Compares *arg1* and *arg2*. If they are the same, returns *arg3*. If not, and *arg4* is the last argument, return *arg4*. Otherwise, the process repeats, comparing *arg4* and *arg5*, and so on. Like other **m4** macros, this takes a maximum of nine arguments.

include(*file***)**
Value is the entire contents of the *file* argument. If *file* is not accessible, a fatal error results.

incr[(*number***)]**
Increments given *number* (default, zero) by one and returns resulting value.

index(*text,pattern***)**
Value is a number corresponding to position of *pattern* in *text*. If *pattern* does not occur in *text*, value is -1.

len(*text***)**
Value is a number that corresponds to length of *text*.

maketemp(*filenameXXXXXX***)**
Value is *filename* with last six characters, usually **XXXXXX**, replaced with current process id and a single letter. Same as the COHERENT system call **mktemp()**.

sinclude(*file***)**
Value is the entire contents of *file*. If *file* is not accessible, return null and continue processing.

substr(*text***[,***start***[,***count***]])**
Value is a substring of *text*. *start* may be left-oriented (nonnegative) or right-oriented (negative). *count* specifies how many characters to the right (if positive) or to the left (if negative) to return. If absent, it is assumed to be large and of the same sign as *start*. If *start* is omitted, it is assumed to be zero if *count* is positive or omitted, or -1 if *count* is negative.

syscmd(*command***)**
Pass *command* to the shell for execution. Value is null. Same as system call **system**.

translit(*text,characters***[,***replacements***])**
Replaces *characters* in *text* with the corresponding characters from *replacements*. If the *replacements* is absent or too short, replace *characters* with a null character. Value is *text* with specified replacements.

undefine(*macro***)**
Remove macro definition. Value is null. If a predefined macro is redefined, its original definition is irrecoverably lost.

undivert[(*stream***[,...])]**
Dumps each specified *stream* into the current output stream. With no arguments, **undivert** dumps all output streams in numeric order. **m4** will not dump any output stream into itself. At the end of processing, **m4** automatically dumps all diverted text to standard output in numeric order. Value is null.

See Also

commands, mktemp(), system
Introduction to the m4 Macro Processor

machine.h — Header File

Machine-dependent definitions
#include <sys/machine.h>

machine.h defines macros, constants, and structures that are specific to the machine upon which COHERENT is being run.

See Also

header files

Notes

This header file is obsolete, and will be dropped from a future release of COHERENT. Its use is strongly discouraged.

macro — Definition

A **macro** is a body of text that is given a name. When the name is used in a program, it is replaced with the text to which it refers; this is called macro *expansion*. For example, **getchar()** is a macro that consists of the function call **getc(stdin)**. The C preprocessor recognizes two varieties of macros: *object-like* and *function-like*.

When the C compiler performs macro substitution, all escape sequences and trigraphs have been resolved. After a macro has been expanded, the expanded text is scanned again to see if the expansion itself contains any macros (not including the original macro that has already been expanded). This re-scanning continues until no further replacement is possible.

Most macros are defined in C headers. The C preprocessor, however, defines some others.

See Also

#define, C preprocessor, m4, manifest constant, Programming COHERENT
ANSI Standard, §3.8.3

madd() — Multiple-Precision Mathematics (libmp)

Add multiple-precision integers
#include <mprec.h>
void madd(*a, b, c***)**
mint **a, *b, *c;*

madd() sets the multiple-precision integer (or **mint**) pointed to by *c* to the sum of the the **mint**s pointed to by *a* and *b.*

See Also

libmp

mail — Command

Send or read mail
mail [-mpqrv] [-f *file***] [***user ...***]**

mail allows you to exchange electronic mail with other COHERENT system users, either on your own system or on other systems via UUCP. Depending upon its form, this command can be used either to send mail to other users or to read the mail that other users have sent to you.

Sending Mail

If you name one or more *user*s, **mail** assumes that you wish to send a mail message to each *user*. **mail** first prints the prompt

```
Subject:
```

on the screen, requesting that you give the message a title.

mail then reads what you type on the standard input. A message is terminated by **<ctrl-D>**, by a line that contains only the character '.', or by a line that contains only the character '?'. Ending with a question mark

prompts **mail** to feed the message into an editor for further editing. The editor used is the one named in the environmental variable **EDITOR**. If this variable is not defined, **mail** uses **ed**.

If you have defined environmental variable **ASKCC** to **YES**, **mail** asks you, after a message is ended, for a list of users to whom you wish to send a copy of the message.

Finally, **mail** prepends the date and the sender's name, and sends the result to each *user* named either on the command line or on the carbon-copy list with the **rmail** command.

Each *user* who has received mail is greeted by the message "You have mail." when she logs in. **mail** normally changes the contents of the mailbox as the user works with them; however, **mail** has options that allow the contents of the mailbox to remain unchanged if the user desires.

Reading Mail

If no *user* is named on its command line, **mail** reads and displays the user's mail, message by message. If environmental variable **PAGER** is defined, **mail** will "pipe" each message through the command it names. For example, the **.profile** command line:

```
export PAGER="exec /bin/scat -1"
```

invokes **/bin/scat** for each mail message with the command-line argument **-1** (the digit one).

While reading mail, the user can use any of the following commands to save, delete, or send each message to another user interactively.

d Delete the current message and print the next message.

m [*user* ...]
Mail the current message to each *user* given (default: yourself).

p Print the current message again.

q Quit, and update mailbox file to reflect changes.

r Reverse the direction in which the mailbox is being scanned.

s [*file* ...]
Save the current mail message with the usual header in each *file* (default: **$HOME/mbox**).

t [*user* ...]
Send a message read from the standard input, terminated by an end-of-file character or by a line containing only '.' or '?', to each *user* (default: yourself).

w [*file* ...]
Write the current message without the usual header in each *file* (default: **$HOME/mbox**).

x Exit without updating the mailbox file.

<newline>
Print the next message.

- Print the previous message.

EOF Quit, updating mailbox; same as **q**.

? Print a summary of available commands.

!*command*
Pass *command* to the shell for execution.

The following command line options control the sending and reading of mail.

-f *file* Read mail from *file* instead of from the default, **/usr/spool/mail/***user*.

-m Send a message to the terminal of *user* if he is logged into the system when mail is sent.

-p Print all mail without interaction.

-q Quit without changing the mailbox if an interrupt character is typed. Normally, an interrupt character stops printing of the current message.

-r Reverse the order of printing messages. Normally, **mail** prints messages in the order in which they were received.

-v Verbose mode. Show the version number of the **mail** program, and display expanded aliases.

If you wish, you can create a signature file, **.signature**, in your home directory. **mail** appends the contents of the signature file to the end of every mail message you send, as a signature. A signature can be your system's path name (for **uucp** messages), your telephone number, an amusing *bon mot*, or what you will.

Files

$HOME/dead.letter — Message that **mail** could not send
$HOME/mbox — Default saved mail
$HOME/.signature — Signature file
/etc/domain — Name of your system's domain
/etc/uucpname — Name of your system
/tmp/mail* — Temporary and lock files
/usr/spool/mail — Mailbox directory, filed by user name

See Also

aliases, ASKCC, commands, EDITOR, .forward, mkfnames, msg, nptx, PAGER, paths, rmail, smail, uux

Notes

Note that before you can send mail, either locally or to a remote site, you must run the program **uuinstall** and use its 'S' option to set the name of your local site and domain. Your local system must, of course, also have permission to log into any remote site to which you wish to send mail. See the tutorial and Lexicon articles on UUCP for details on using UUCP to exchange mail and files with remote sites.

mail — Overview

Electronic mail system

The COHERENT system includes a full-featured, UNIX-style mail system. This system consists of commands with which your system can send, receive, and forward mail; and configuration files, with which you can describe potential recipients of mail, either on your system or other systems. This article describes the design of the COHERENT mail system, and introduces the commands and files that compose it.

The COHERENT mail system has three major components: the *user agent* (also called the *mailer*); the *routing agents* (the commands **smail** and **rmail**); and the *delivery agents* (the commands **lmail** and **uux**).

This structure may seem overly complex (you may ask why all of this functionality could not be bundled into one program); however, experience has shown that it is best to organize each set of functions within its own program. One advantage this gives you is that you can replace one part of the mail system with another, superior program, without disturbing the operation of the system as a whole. For example, you may wish to replace the mailer **mail** with another mailer, such as **elm**. Because the mailer is its program, you can replace **mail** with **elm** without affecting the delivery or routing agents at all. You may never need to modify how the routing or delivery agents work, but studying the overall structure of the mail system will help you to decide intelligently on whether to replace a part of the mail system, and will also help you diagnose any problems that may crop up.

The following describes each set of agents in turn.

The User Agent

The user agent (also called the *mailer*) presents to you the messages that have been delivered into your mailbox. It also collects messages from you and hands them to the routing agent for delivery. This is the program you invoke when you wish to read mail or send a mail message.

COHERENT comes with one mailer, called **mail**. When you invoke it without any arguments, it reads the contents of file **/usr/spool/mail/***user* (where *user* is your login identifier), breaks its contents into individual messages, and presents the messages to you one by one. (File **/usr/spool/mail/***user* is also called your "mailbox", because that is where the mail system deposits your messages.) You can read a mail message; then, by giving commands to **mail**, you can reply to the message if you wish, then copy it into a file for archiving or throw it away.

If you wish to send mail, type the command

```
mail user
```

where *user* identifies the user to whom you wish to send the message. *user* can either be a user on your system, or on a remote system; if the latter, you must also name the site on which *user* resides. The syntax for naming a remote sites is described in further detail below. When you invoke **mail** to send a mail message, **mail** presents the prompt **Subject:**, upon which you should type the subject of the message. When you have typed the subject, press (¢); you can then type the body of the message. Press (¢) when you come to the end of a line; the cursor jumps to the beginning of the next line on your screen, and you can continue typing. When you have finished typing, type **<ctrl-D>** or a '.' at the beginning of a line; this signals **mail** that you have finished entering the message. **mail** then passes the message to the routing agents for delivery, as described in the next section.

If, while you are typing the body of your message, you type the letter '?' at the beginning of a line, **mail** invokes an editor and copies your message into it. The editor it invokes is the set named by the environmental variable **EDITOR**. You can then use the editor to finish typing your message. When you have finished typing your message, exit from the editor; the message will then dispatched.

When the mail dispatches your message, it checks your home directory for a file named **.signature**. This is your *signature* file: the mailer appends the contents of this file onto the end of your message, as your signature. A signature can be any mass of text that you wish; usually, it gives a user's name and e-mail address, and sometimes includes a joke, motto, or slogan as a form of self-expression. It's generally considered bad form to have a signature that exceeds five lines of text, or that uses vulgar, obscene, or abusive language.

To mail a file to another user, use the shell's redirection operator '<'. For example, the command

```
mail stephen < bug.report
```

mails file **bug.report** to user **stephen**. The file will be prefixed with your address, and suffixed with your mail "signature", should you have one.

For details on how to use the mailer and its commands, see the Lexicon entry for the command **mail**.

Other mailers are also available for COHERENT; the most popular one is named **elm**. This mailer uses a visual interface to display the messages that are in your mailbox; you can use the arrow keys on your keyboard to move a cursor and select the message you want. Sources and binaries for **elm** are available on the MWC BBS and on other sites on the Internet.

Routing Agents

The routing agent, as term implies, figures out how to deliver a message to its destination, and dispatches it appropriately.

The routing agent **rmail** ("route mail") receives mail from another system. If the mail is intended for your local system, **rmail** passes the mail to the delivery agent **lmail** (described below) for delivery on your system. If, however, the mail is intended for forwarding to another system, **rmail** forwards it appropriately. **rmail** does most of its work in the background; you will seldom if ever will need to work with it directly.

The routing agent **smail** ("send mail") receives mail from you and dispatches to its target user, either on your system or on another system. **smail** actually is a large, complex program that handles mail correctly under a great variety of conditions. Under COHERENT, **rmail** actually is a link to **smail**.

When you mail a message to *user*, **smail** performs the following steps to route the message:

- **smail** first looks up *user* in the file **/usr/lib/mail/aliases**. For details on aliases, see the Lexicon entry **aliases**.
- If **smail** finds *user* in one of these files, it substitutes the alias for *user*. If the alias is of the form

  sys!*user*

  or

  sys! . . . !*user*

  or

  user@sys[.domain]

  smail treats it as a remote destination, and invokes command **uux** to spool the message to *sys*. When **uux** has delivered the message, it becomes the responsibility of command **uuxqt** on *sys* to pass the message to *user*.

- If **smail** finds no match in **/usr/lib/mail/aliases**, it looks up *user* in the file **/etc/passwd**, to see if she is a user on your local system. If **smail** does not find *user* in **/etc/passwd**, it throws away the message and mails an error message to the sender.
- If **smail** does find *user* in **/etc/passwd**, it then looks for file **.forward** in *user*'s home directory. This file normally contains a list of addresses to which mail is to be forwarded.
- If *user*'s home directory does not contain a file named **.forward**, **smail** passes the message to **lmail** which writes the message into the file **/usr/spool/mail/***user* (the user's "mailbox").
- If, however, *user* does have file **.forward** in her home directory, **smail** reads it, and forwards the message to the address or addresses that that file contains.

For further information on **smail**, and on its suite of configuration files, see the Lexicon entry for **smail**.

Before you can send mail to a remote site, you must have set up a UUCP connect to that site, either directly or indirectly. That is, you must have set up UUCP to send mail to that site, or to a system that can forward the mail to some other that may have permission to access the site in question. See the tutorial and Lexicon articles on UUCP for details on using UUCP to exchange mail and files with remote sites.

Please note that the routing agents are the only components of the mail system that must run **setuid** to assume the privilege of the superuser **root**.

Delivery Agents

The delivery agents actually move messages to their destination.

The delivery agent **lmail** ("local mail") places messages into users' mailboxes. To discourage the forging of mail, **lmail** does not use **setuid**. It must be run by a privileged user (generally **root**) so that it will have permission to write mail into every user's mailbox. As a rule, **lmail** is invoked only by the routing agent; you seldom, if ever, will work directly with **lmail**.

The UUCP **uux** queues commands for execution on a remote system. The mail system uses **uux** as a delivery agent; in fact, on most systems, the delivery of mail is **uux**'s principal task. When a message is to be forwarded to a remote site, **smail** invokes **uux** to create two files in directory **/usr/spool/uucp/***site* (where *site* names the site to which mail is being sent). One file, which has the prefix 'C', contains the **rmail** command to be executed on the remote site; the other file, which has the prefix 'D', contains the body of the message that **rmail** is to route. The next time your system polls *site* (or is polled by *site*), those files are copied to *site*, where they are executed by *site*'s copy of the command **uuxqt**.

You can use **uux** directly to spool commands for execution. For details, see the Lexicon entries for **uux** and **uuxqt**.

uux uses **setuid** to assume the identity of user **uucp** in order to write into the necessary spool directories. Please note that it is very easy to use **uux** to forge messages to remote systems. Keep this in mind if you plan to use electronic mail for any kind of authorization system.

Setting Up a Mail Feed

One of the most useful tasks a personal computer can perform for you is let you exchange electronic mail with users on remote systems. The following describes how you can set up your mail system so you can plug into the Internet and begin to exchange mail with the outside world.

To begin, the COHERENT system at present can exchange mail with remote systems only via UUCP. To receive mail, you must find a site that has a connection to the Internet — either direct or indirect — and is willing to act as a UUCP feeder for you. Such a site may be a local college or university, or a commercial "Internet provider."

Once you have located such a site, set up a UUCP connection with that site, as described in this manual's tutorial on UUCP. If you do not have experience in setting up a UUCP site, read this tutorial carefully, as this can be rather tricky.

Next, edit file **/etc/domain**, and set your system's domain to the name of the system from which you will be receiving your feed. For example, if you have purchased Internet service from site **acme.com**, then **/etc/domain** should read:

```
acme.com
```

Finally, you must edit file **/usr/lib/mail/config** to tell **smail** to forward to the feeder system all messages that are bound for the outside world. You must change two attributes within this file:

domains
This attribute sets the domain name that **smail** writes into each mail message's header. This is required so that other users can reply to your messages. Set it to the name of your feeder system; it should be identical to the name you wrote into **/etc/domain**. For example, if you are purchasing your Internet service from system **acme.com**, then set the attribute **domains** to:

```
domains=acme.com
```

smart_path
Set this to the name of the remote site as you have set it in file **/usr/lib/uucp/sys**. For example, if you are purchasing Internet service from site **acme.com**, you may have used the name **acme** to name the site within file **/usr/lib/uucp/sys**. In this instance, you set the **smail** attribute **smart_path** as follows:

```
smart_path=acme
```

That's all there is to it. **smail**'s default configurations will handle the rest. Once you have the UUCP connection working properly, then any mail to a user who is not on your local system will be forwarded to the system that is providing your mail feed, and from there forwarded to the remote site to which you addressed it.

For details on setting up a UUCP feed, see the UUCP tutorial that appears earlier in this manual; also see the Lexicon entries for **UUCP**, **sys**, **dial**, and **port**. For more information on modifying **smail**'s configuration file, see the Lexicon entry for **config**.

Mailing to Networks

The following gives directions on how to send mail to users on popular networks:

America Online
Send mail to *user***@aol.com**.

Applelink
Send mail to *user***@applelink.apple.com**.

ATTMail
Send mail to *user***@attmail.com**.

BITNET
Send mail to *user*@*host***.bitnet** or to *user***%***host***.bitnet@***gateway*.

Compuserve
Send mail to *number***.***number***@compuserve.com**. Note that Compuserve addresses are usually given as *number*,*number*; you must convert the comma to a period.

FidoNet
This network uses an unusual addressing scheme. To send mail to John Doe at 1:123/456.0, use the following domain address:

```
f456.n123.z1.fidonet.org.
```

The **z1** comes from the **1:** at the front of the FidoNode address. Then, put the person's name in front of this, with the at-sign between them:

```
john.doe@f456.n123.z1.fidonet.org
```

If the host label does not end in **.0**, as in 1:123/456.4, use that digit with **p** prefixed to it, as follows:

```
john.doe@p4.f456.n123.z1.fidonet.org
```

MCIMail
Send mail to *user***@mcimail.com**. **MCIMail** usually includes a hyphen in *user*'s name; be sure to remove it.

UUnet
Send mail to *user***@host.uucp**, or to *user***%***host***.uucp@***gateway*, or to *user*@*domain*.

These directions assume that you have a UUCP link to another system that gives you access to the Internet or other intelligent network. For more information on sending mail to remote systems via UUCP, see the Lexicon entry for **UUCP**.

Files

$HOME/.forward — Forwarding instructions for inbound mail
$HOME/.signature — Personal signature
$HOME/dead.letter — Message that **mail** could not send
/etc/domain — Name of your system's domain
/etc/passwd — User identities

/etc/uucpname — Name of your system
/tmp/mail* — Temporary and lock files
/usr/lib/mail/aliases — Aliases of users
/usr/lib/mail/config — **smail** configuration file
/usr/lib/mail/fullnames — Short full name aliases of users
/usr/lib/mail/paths — Mail routing control file
/usr/lib/mail/routers — Information on routing mail to remote sites
/usr/lib/mail/transports — Information on mail-transportation programs
/usr/spool/mail — Mailbox directory, filed by user name

See Also

aliases, commands, config, cvmail, .forward, mail, mkfnames, msg, nptx, paths, rmail, smail, UUCP
Krol, E.: *The Whole Internet: User's Guide & Catalog.* Sebastopol, Ca.: O'Reilly & Associates, Inc., 1992. *Highly recommended.*

mailq — Command

Display information about spooled mail
mailq [-v]

Command **mailq** displays information about the mail currently spooled in directory **/usr/spool/mail** and its subdirectories. Command-line option **-v** tells **mailq** to display a per-message transaction log for each message, which shows what has happened to the message so far.

See Also

commands, mail [overview], smail

Notes

mailq is a link to command **smail**.

main() — C Language

Introduce program's main function

A C program consists of a set of functions, one of which must be called **main()**. This function is called from the runtime startup routine after the runtime environment has been initialized.

Programs can terminate in one of two ways. The easiest is simply to have the **main()** routine **return()**. Control returns to the runtime startup; it closes all open file streams and otherwise cleans up, and then returns control to the operating system, passing it the value returned by **main()** as exit status.

In some situations (errors, for example), it may be necessary to stop a program, and you may not want to return to **main()**. Here, you can use the library function **exit()**; it cleans up the debris left by the broken program and returns control directly to the operating system.

The system call **_exit()** quickly returns control to the operating system without performing any cleanup. This routine should be used with care, because bypassing the cleanup will leave files open and buffers of data in memory.

Programs compiled by COHERENT return to the program that called them; if they return from **main()** with a value or call **exit()** with a value (e.g., **EXIT_SUCCESS** or **EXIT_FAILURE**), **main()** returns that value to the program that invoked it (e.g., the shell). Programs that invoke other programs through the function **system()** check the returned value to see if these secondary programs terminated successfully. If you exit from **main()** without explicitly returning a value (e.g., by just letting **main()** simply conclude, or by invoking **exit()** without a return status, or by invoking **return** without a return value), **main()** returns whatever random value happens to have been in the register EAX.

See Also

_exit(), argc, argv, C language, envp, exit(), EXIT_FAILURE, EXIT_SUCCESS
ANSI Standard, §5.1.2.2.1
POSIX Standard, §3.1.2.2

major number — Definition

Device numbering

A *major number* specifies the device driver associated with a given device name found in the directory **/dev**. COHERENT uses a device's the major number as an index into an internal table of device-driver pointers.

Every COHERENT device has a device number associated with it. This device number is of type **dev_t**, as defined in **<sys/types.h>**. The macro **major()** in **<sys/stat.h>** extracts the major number from a given device number.

See Also

device drivers, minor number, stat.h

make — Command

Program-building discipline
make [*option …***] [***argument …***] [***target …***]**

make helps you build things that consist of more than one file of source code. A "thing" can be a program, a report, a document, or anything else that is constructed out of something else.

Complex programs often consist of several *object modules*, each of which is the product of compiling a *source file*. A source file may refer to one or more **include** files, which can also be changed. Some programs may be generated from specifications given to program generators, such as **yacc**. Recompiling and relinking complicated programs can be difficult and tedious.

make regenerates programs automatically. It follows a specification of the structure of the program that you write into a file called **makefile**. **make** also checks the date and time that COHERENT has recorded for each source file and its corresponding object module; to avoid unnecessary recompilation, **make** will recompile a source file only if it has been altered since its object module was last compiled.

Options

The following lists the options that can be passed to **make** on its command line.

-d (Debug) Give verbose printout of all decisions and information going into decisions.

-e Force macro definitions in environment to override those in the **makefile**.

-f *file* *file* contains the **make** specification. If this option does not appear, **make** uses the file **makefile**, which is sought first in the directories named in the **PATH** environmental variable, and then in the current directory. If *file* is '-', **make** uses the standard input; note, however, that the standard input can be used *only* if it is piped.

-i Ignore all errors from commands, and continue processing. To invoke this behavior for an individual action within a **makefile**, prefix the action with the '-' flag. By default, **make** exits if a command returns an error.

-k Continue to update other targets that do not depend upon the current target if a non-ignored error occurs while executing the commands to bring a target up to date.

-n Test only; suppress execution of commands. To override this behavior for an individual action within a **makefile**, prefix the action with the '+' flag.

-p Print all macro definitions and target descriptions.

-q Return a zero exit status if the targets are up to date. Do not execute any commands.

-r Do not use the built-in rules that describe dependencies.

-S Terminate **make** if an error occurs while executing the commands to bring a target up to date. This is true by default, and the opposite of **-k**.

-s Do not print command lines when executing them. Commands preceded by '@' are not printed, except under the **-n** option.

-t (Touch option) Force the dates of targets to be the current time, and bypass actual regeneration. However, if the target is out-of-date, **make** will still execute an individual action if that action is prefixed with the '+' flag.

The Makefile

A **makefile** consists of three types of instructions: *macro definitions*, *dependency definitions*, and *commands*.

A macro definition simply defines a macro for use throughout the **makefile**; for example, the macro definition

```
FILES=file1.o file2.o file3.o
```

Note the use of the equal sign '='.

A dependency definition names the object modules used to build the target program, and source files used to build each object module . It consists of the *target name*, or name of the program to be created, followed by a colon ':' and the names of the object modules that build it. For example, the statement

```
example:  $(FILES)
```

uses the macro **FILES** to name the object modules used to build the program **example**. Likewise, the dependency definition

```
file1.o:  file1.c macros.h
```

defines the object module **file1.o** as consisting of the source file **file1.c** and the header file **macros.h**.

Finally, a command line details an action that **make** must perform to build the target program. Each command line must begin with a space or tab character. For example, the command line

```
cc -o example $(FILES)
```

gives the **cc** command needed to build the program **example**. The **cc** command lists the *object modules* to be used, *not* the source files.

Note that if you prefix an action with a hyphen '-', **make** will ignore errors in the action. If the action is prefixed by '@', it tells **make** to be silent about the action — that is, do not echo the command to the standard output. The '+' flag is described below.

Finally, you can embed comments within a **makefile**. **make** ignores a pound sign '#' and all text that follows it. COHERENT's implementation of **make** recognizes the presence of quotation marks, and and does not treat a '#' as a comment if it appears between apostrophes or quotation marks, or prefixed by a backslash. Many other versions of **make** do not permit this, including the one specified by POSIX.2: *caveat utilitor*.

make searches for **makefile** first in directories named in the environmental variable **PATH**, and then in the current directory.

make Without a Makefile

Beginning with release 4.2 of COHERENT, you can also invoke **make** to build an object for which no **makefile** exists. In this case, **make** uses its default suffix rules to identify the objects it should construct and how it should construct them. If, for example, you type

```
make foo
```

make will search the local directory for any file named **foo** that has any of the suffices that **make** recognizes by default. If the local directory contains a file named **foo.c**, **make** invokes **cc** to compile it; whereas if it contains a file named **foo.o**, it invokes the linker **ld** to link it.

Note that if no **makefile** exists, **make** by default creates an executable named after the C source file, just as the command **cc** does.

Dependencies

The **makefile** specifies which files depend upon other files, and how to recreate the dependent files. For example, if the target file **test** depends upon the object module **test.o**, the dependency is as follows:

```
test: test.o
      cc -o test test.o
```

make knows about common dependencies, e.g., that **.o** files depend upon **.c** files with the same base name. The target **.SUFFIXES** contains the suffixes that **make** recognizes. (Note that you can use the command **makedepend** to build such a list dynamically. For details, see its entry in the Lexicon.)

make also has a set of rules to regenerate dependent files. For example, for a source file with suffix **.c** and a dependent file with the suffix **.o**, the target **.c.o** gives the regeneration rule:

LEXICON

```
.c.o:
      cc -c $<
```

The **-c** option to the **cc** commands tells **cc** not to link or erase the compiled object module. **$<** is a macro that **make** defines; it stands for the name of the file that causes the current action. The default suffixes and rules are kept in the files **/usr/lib/makemacros** and **/usr/lib/makeactions**.

Macros

To simplify the writing of complex dependencies, **make** provides a *macro* facility. To define a macro, write

NAME **=** *string*

string is terminated by the end-of-line character, so it can contain blanks. To refer to the value of the macro, use a dollar sign '$' followed by the macro name enclosed in parentheses or braces, e.g.:

$(*NAME*)
${*NAME*}

If the macro name is one character, parentheses are not necessary. **make** uses macros in the definition of default rules:

```
.c.o:
      $(CC) $(CFLAGS) -c $<
```

where the macros are defined as

```
CC=cc
CFLAGS=-V
```

The built-in macros are as follows:

$* The target's name, minus a '.'-delimited suffix.

$@ For regular targets, the target's full name. For targets that are library dependencies of the form *library***(***object***)**, this macro expands to the *library* part of the target.

$% For targets that are library dependencies of the form *library***(***object***)**, this macro expands to the *object* part of the target.

$? This expands to prerequisite files that are newer than the target.

$< For suffix-rules, this macro expands to the name of the prerequisite file that **make** chose as the implicit prerequisite of the target. Do not use this macro outside a suffix rule.

You can specify macro definitions in the **makefile**, in the environment, or as a command-line argument. A macro defined as a command-line argument always overrides a definition of the same macro name in the environment or in the **makefile**. Normally, a definition in a **makefile** overrides a definition of the same macro name in the environment; however, with the -e option, a definition in the environment overrides a definition in the **makefile**.

Each command line *argument* should be a macro definition of the form

```
OBJECT=a.o b.o
```

Arguments that include spaces must be surrounded by quotation marks, because blanks are significant to the shell **sh**.

Source File Path

make first looks for the file with the name given, which may be relative to the current directory when make was invoked. If it does not find the file, and if the name of the file is not an absolute path name, **make** removes any leading path information from the name and looks for the file in the current directory. If the file is not found in the current directory, **make** then searches for the file in the list of directories specified by the macro **$(SRCPATH)**. This allows you to compile a program in an object directory separate from the source directory. For example

```
export SRCPATH=/usr/src/local/me
make
```

or alteratively

```
make SRCPATH=/usr/src/local/me
```

builds objects in the current directory as specified by the **makefile** and sources in **/usr/src/local/me**. To test changes to a program built from several source files, copy only the files you wish to change to the current directory; make will use the local sources and find the other sources on the **$(SRCPATH)**.

Note that **$(SRCPATH)** can be a single directory, as in the above example, or a ':'-separated list of directories, as described in the Lexicon entry for the function **path()**.

Macros and Environmental Variables

The environmental variable **MAKEFLAGS** provides an alternative method of passing parameters to **make**. If this variable is defined, **make** processes the switches that it contains as if they were specified on the command line. **make** processes **MAKEFLAGS** before it processes any actual command-line parameters.

Either of the following two formats can be used for **MAKEFLAGS**:

```
MAKEFLAGS="-n -d"
MAKEFLAGS=nd
```

Either of the above passes to **make** the options **n** and **d**.

After it processes the switches named in **MAKEFLAGS**, **make** processes all options set on its command line. **make** then re-defines **MAKEFLAGS** to contain the full set of switches passed to it, and marks the macro for export. This means that recursive invocations of **make** are passed the same switch settings as the highest-level invocation of **make**. (See also the description of the '+' flag, below.)

make takes all other environment-variable settings passed to it and uses them to set the values of corresponding macros internally.

When **make** executes a command, it exports to that command all the environmental variables **make** imported from the initial environment, the **MAKEFLAGS** environmental variable, and the macros defined on the **make** command line.

Always Actions

If an action for rebuilding a target begins with the '+' flag, **make** executes the action even if the command line specifies the option **-n**. This is useful when dealing with recursive **makefiles**: when you pass the options **-p**, **-d**, or **-n** to the top-level invocation of **make**, the top-level **makefile** can still invoke the sub-**makefiles**, and pass them the same flags via the environmental variable **MAKEFLAGS**, as described above. This simplifies the debugging of **makefile**s for complex projects. This flag mainly affects **make**'s usage with the options **-q**, **-n**, and **-t**.

Library Dependencies

make interprets targets of the form *library***(***object***)** as referring to members of an archive created with the archiver **ar**. **make** can examine the archive's contents to determine whether the named member is present and what date it possesses.

When building such a target, **make** looks for suffix rules for use in building *object*, but with a target suffix of **.a** rather than the actual suffix of *object*.

For example, with the default **make** rules in effect, the target

```
libc.a(clock.o)
```

would be rebuilt from a source file **clock.c** by the suffix-rule **.c.a**. The default suffix rule (as supplied in file **/usr/lib/makeactions**) deals with building the *object* file and then uses the macros **AR** and **ARFLAGS** to move the resulting object file into the target archive.

Actions for library targets use macro definitions that differ slightly from those for normal actions. When it builds a library target, **make** sets the macro **$@** to the name of the *library* part of the target, and sets the special macro **$%** (defined only for use with library targets) to the name of the *object* part of the target.

Single-Suffix Rules

make can use an inference rule of the form:

```
suffix:
        actions
```

to infer an action from a target that does not have a suffix. When you use a target that has no explicit rule and no known suffix, **make** appends onto the target every known suffix in turn, and for each suffix searches for a file or rule for building the target. If **make** discovers a file that matches one of file names that it has built, it then tries to

use a single-suffix rule to generate the target from **target***suffix*, with the actions given in the single-suffix rule.

For example, the default rules for **make** contain a single-suffix rule:

```
.c:
        $(CC) $(CFLAGS) $@ $<
```

Given the above rule and a file in the current directory or source path named **clock.c**, the target

```
clock:
```

results in the executable file **clock** being built by compiling the single source file **clock.c** and linking it.

Suffix-Rewriting Macro Expansion

You can use a special form of macro expansion

$(*macro***:***suffix***[=***value***])**

to simplify the use of macros that involve long lists of files names. When you request the above form of expansion, **make** searches the expansion of *macro*; for every word that ends in *suffix* it replaces *suffix* with the optional *value*.

For example, consider the following:

```
SOURCES = parse.c interpret.c builtin.c
OBJS = $(SOURCES:.c=.o)
```

This expansion of the macro **OBJS** is:

```
parse.o interpret.o builtin.o
```

When a **makefile** uses long lists of files, this facility not only saves typing, but eases maintenance because you need to change only one list of files.

Path-Oriented Macro Expansions

The following special-macro expansion forms perform path processing on the macro's contents:

$(*special***)**	Normal expansion
$(*special***F)**	Expand only file-name part
$(*special***D)**	Expand only directory part without trailing slash

where *special* is one of the following: **@**, **?**, **<**, *, or **%**. These expansion forms allow rules (especially inference rules) to deal easily with path-oriented operations, without resorting to complex shell operations involving backquoting and the command **basename**. In particular, when expanding a macro with a file list such as **${?D}**, **make** processes all the entries in the file list as specified; otherwise, this would be extremely cumbersome.

Files

makefile
Makefile — List of dependencies and commands
/usr/lib/makeactions — Default actions
/usr/lib/makemacros — Default macros

See Also

as, cc, commands, ld, makedepend, srcpath, touch
The make Programming Discipline, tutorial

Diagnostics

make returns the following error messages:

; after target or macroname *(error)*
A semicolon appeared after a target name or a macro name.

Bad macro name *(error)*
A macro includes an illegal character, e.g., a control character.

= in or after dependency *(error)*
An equal sign '**=**' appeared within or followed the definition of a macro name or target file. For example, **OBJ=atod.o=factor.o** will produce this error.

Incomplete line at end of file *(error)*
: An incomplete line appeared at the end of the **makefile**.

Macro definition too long *(error)*
: The macro definition exceeds the limited designed into the preprocessor.

Multiple actions for *name (error)*
: A target is defined with more than one single-colon target line.

Multiple detailed actions for *name (error)*
: A target is defined with more than one single-colon target line.

Must use "::" for *name (error)*
: A double-colon target line was followed by a single-colon target line.

Newline after target or macroname *(error)*
: A newline character appears after a target name or a macro name.

'::' not allowed for *name* (error)
: A double-colon target line was used illegally; for example, after single-colon target line.

::: or : in or after dependency list *(error)*
: A triple colon is meaningless to **make**, and therefore illegal wherever it appears. A single colon may be used only in a target line (which is also called the *dependency list*), and nowhere else.

Out of core (adddep) *(error)*
: This results from a system problem. Try reducing the size of your **makefile**.

Out of range number input. *(warning)*
: You attempted to use a numeric value that is out of range.

Out of space *(error)*
: System problem. Try reducing the size of your **makefile**.

Out of space (lookup) *(error)*
: System problem. Try reducing the size of your **makefile**.

Syntax error *(error)*
: The syntax of a line is faulty.

Too many macro definitions *(error)*
: The number of macros you have created exceeds the capacity of your computer to process them.

= without macro name or in token list *(error)*
: An equal sign '**=**' can be used only to define a macro, using the following syntax: "MACRO=*definition*". An incomplete macro definition, or the appearance of an equal sign outside the context of a macro definition, will trigger this error message.

: without preceding target *(error)*
: A colon appeared without a target file name, e.g., :*string*.

Notes

Prior to release 4.2, COHERENT's implementation of **make** permitted users to use the macro **$<** outside of suffix rules. This non-standard behavior is no longer supported.

The order of items in **makemacros/.SUFFIXES** is significant. The consequent of a default rule (e.g., **.o**) must *precede* the antecedent (e.g., **.c**) in the entry **.SUFFIXES**. Otherwise, **make** will not work properly.

makeboot — Command

Create a bootable floppy disk
makeboot

The script **makeboot** automatically builds a bootable floppy disk. To use it, insert a scratch floppy disk into your system's drive 0 (drive A:), log in as the superuser **root**, and then type **makeboot**.

makeboot automatically formats the floppy disk, builds a file system on it, and copies onto the disk all files it needs to boot COHERENT. To abort **makeboot** at any time, press **<ctrl-C>**.

Because **makeboot** is a script you can — and should — edit it to suit your preferences. For example, by default

makeboot copies both the editors **vi** and **me** onto the floppy disk; you may wish to save space by copying just one or the other.

See Also

commands, floppy disk

Notes

makeboot reads file **/bin/mount** to discover whether floppy-disk drive 0 is 3.5 inches or 5.25 inches. This file was initialized when you last installed or updated COHERENT. If you have changed your A drive since then, **makeboot** may think it is working with one size of floppy disk when in fact it is working with another. To correct this error, abort **makeboot**, then edit **/etc/mount** so that it reflects your system's true configuration of disk drives.

makedepend — X Utility

Generate list of dependencies for a makefile
makedepend [-D*name***=***def***] [-D***name***] [-I***includedir***] [-Y***includedir***] [-a] [-f***makefile***] [-o***objsuffix***] [-p***objprefix***] [-s***string***] [-w***width***] [-v] [-m] [--***otheroption ...***--]** *sourcefile ...*

makedepend reads each *sourcefile*, and parses it as the C preprocessor does. It processes every **#include**, **#define**, **#undef**, **#ifdef**, **#ifndef**, **#endif**, **#if**, and **#else** directives so that it can correctly tell which **#include** directives should be used in a compilation. Any **#include** directive can reference a file that has other **#include** directives, and **makedepend** parses these files as well.

Every file that a *sourcefile* includes, directly or indirectly, is what **makedepend** calls a *dependency*. It writes these dependencies into a **makefile** in such a way that **make** will know which object files must be recompiled when a dependency has changed.

By default, **makedepend** writes its output into a file named **makefile**, if it exists; otherwise, it writes its output into **Makefile**. You can specify an alternate makefile with the option **-f**. **makedepend** first searches the makefile for the line

```
# DO NOT DELETE THIS LINE -- make depend depends on it.
```

or one provided with the option **-s** as a delimiter for the dependency output. If it finds the line, it deletes everything after after this line to the end of the makefile, and writes its output after this line. If **makedepend** does not find this line, it appends the string to the end of the makefile and writes the output after that. For each *sourcefile*, **makedepend** writes into the makefile a line of the form

```
sourcefile.o: dfile ...
```

where **sourcefile.o** is the name of the *sourcefile* with its suffix replaced **.o**, and **dfile** is a dependency that **makedepend** discovered in a **#include** directive as it parsed *sourcefile* or one of the files it includes.

Command-line Options

makedepend ignores any option it does not understand, so you can use the same arguments that you would for **cc**. It does recognize the following command-line options:

-D*symbol***[=***def***]**
Define *symbol* within **makedepend**'s internal symbol table. Without **=***def*, **makedepend** defines it as **1**.

-I*includedir*
Tell **makedepend** to prefix *includedir* onto the list of directories to search when it encounters a **#include** directive. By default, **makedepend** only searches only **/usr/include**.

-Y[*includedir***]**
Search *includedir* for header files instead of all of the standard header-file directories. If you omit to name an *includedir*, this option prevents searching of the standard header-file directories.

-a Append the dependencies to the end of the file instead of replacing them.

-f*file* Write output into *file* instead of into **makefile**.

-o*objsuffix*
Append *objectsuffix* to a *filename* instead of the default **.o**.

-p*objprefix*
Prefix the name of each object file with *objprefix*. This usually is used to designate a different directory for the object file. The default is the empty string.

-s*string* Use *string* as the starting-string delimiter within a **makefile**.

-w*width*
Set the width of a line of output to *width*. By default, **makedepend** limits a line of output to 78 characters.

-v Verbose: tell **makedepend** to write onto the standard output the list of files that each input file includes.

-m Warn about multiple inclusion. This option tells **makedepend** to warn if any input file includes another file more than once. In previous versions of **makedepend**, this was the default behavior; the default has been changed to better match the behavior of the C compiler, which does not consider multiple inclusion to be an error. This option is provided for backward compatibility, and to aid in debugging problems related to multiple inclusion.

-- option ... --
makedepend ignores every *option* that it does not recognize and that is bracked by two hyphens '--'. In this way, you can safely tell **makedepend** to ignore esoteric compiler arguments that might normally be found in a **CFLAGS** macro. **makedepend** processes normally all options between the pair of double hyphens that recognizes.

Algorithm

To speed its performance, **makedepend** makes two assumptions about the programs whose dependency it is mapping: first, that all files compiled by a single **makefile** will be compiled with roughly the same **-I** and **-D** options; and second, that most files in a directory include largely the same files. Given these assumptions, **makedepend** expects to be called once for each makefile, with all source files that that **makefile** maintains appearing on its command line.

makedepend parses each source and header file exactly once, and maintains an internal symbol table for each. Thus, the first file on the command line will take an amount of time proportional to the amount of time that a normal C preprocessor takes. However, on subsequent files, if **makedepend** encounters a header file that it has already parsed, it does not parse it again.

For example, imagine you are compiling two files, **file1.c** and **file2.c**. Assume, further, that each includes the header file **header.h**, and **header.h** in turn includes the files **def1.h** and **def2.h**. When you run the command

```
makedepend file1.c file2.c
```

makedepend parses **file1.c** and, therefore, **header.h** followed by **def1.h** and **def2.h**. It then decides that the dependencies for this file are

```
file1.o: header.h def1.h def2.h
```

When **makedepend** parses **file2.c** and discovers that it, too, includes **header.h**, it does not re-parse that file, but simply adds **header.h**, **def1.h**, and **def2.h** to the list of dependencies for **file2.o**.

Example

makedepend normally is used within a **makefile** target, so that typing the command

```
make depend
```

brings the dependencies up to date for the **makefile**. For example,

```
SRCS = file1.c file2.c ...
CFLAGS = -O -DHACK -I../foobar -xyz
depend:
        makedepend -- $(CFLAGS) -- $(SRCS)
```

See Also

cc, commands, make
X Windows Manual: **imake**

Notes

makedepend was written by Todd Brunhoff of Tektronix, Inc., and MIT Project Athena.

LEXICON

malloc() — General Function (libc)

Allocate dynamic memory
#include <stdlib.h>
char *malloc(*size***) unsigned** *size***;**

malloc() helps to manage a program's free-space arenas. It uses a circular, first-fit algorithm to select an unused block of at least *size* bytes, marks the portion it uses, and returns a pointer to it. The function **free()** returns allocated memory to the free memory pool.

Each arena allocated by **malloc()** is rounded up to the nearest even number and preceded by an **unsigned int** that contains the true length. Thus, if you ask for three bytes you get four, and the **unsigned** that precedes the newly allocated arena is set to four.

When an arena is freed, its low order bit is turned on; consolidation occurs when **malloc()** passes over an arena as it searches for space. The end of each arena contains a block with a length of zero, followed by a pointer to the next arena. Arenas point in a circle.

The most common problem with **malloc()** occurs when a program modifies more space than it allocates with **malloc()**. This can cause later **malloc()**s to crash with a message that indicates that the arena has been corrupted. You can use the function **memok()** to isolate these problems.

Example

This example reads from the standard input up to *NITEMS* items, each of which is up to *MAXLEN* long, sorts them, and writes the sorted list onto the standard output. It demonstrates the functions **qsort()**, **malloc()**, **free()**, **exit()**, and **strcmp()**.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define NITEMS 512
#define MAXLEN 256
char *data[NITEMS];
char string[MAXLEN];

main()
{
      register char **cpp;
      register int count;
      extern int compare();

      for (cpp = &data[0]; cpp < &data[NITEMS]; cpp++) {
            if (gets(string) == NULL)
                  break;
            if ((*cpp = malloc(strlen(string) + 1)) == NULL)
                  exit(1);
            strcpy(*cpp, string);
      }

      count = cpp - &data[0];
      qsort(data, count, sizeof(char *), compare);

      for (cpp = &data[0]; cpp < &data[count]; cpp++) {
            printf("%s\n", *cpp);
            free(*cpp);
      }
      exit(0);
}

compare(p1, p2)
register char **p1, **p2;
{
      extern int strcmp();
      return(strcmp(*p1, *p2));
}
```

See Also

alloca(), arena, calloc(), free(), libc, memok(), realloc(), setbuf(), stdlib.h
ANSI Standard, §7.10.3.3

POSIX Standard, §8.1

Diagnostics

malloc() returns NULL if insufficient memory is available.

Notes

The function **alloca()** allocates space on the stack. The space so allocated does not need to be freed when the function that allocated the space exits.

malloc.h — Header File

Definitions for memory-allocation functions
#include <sys/malloc.h>

malloc.h defines constants, structures, and macros used with COHERENT's memory-allocation functions. Note that this header does not declare the library's memory-allocation functions.

See Also

header files

Notes

This header file is obsolete, and will be dropped from a future release of COHERENT. Its use is strongly discouraged.

man — Technical Information

Manual macro package
nroff -man *file ...*

The macro package **tmac.an** formats UNIX-style manual pages. To invoke this package, use the argument **-man** with either **nroff** or **troff**

tmac.an includes the following macros:

.B [*string ...***]**
Use **boldface** font. If used with one or more *string*s, prints them in boldface. Otherwise, print all subsequent text in boldface, up to the next explicit change of font.

.BI boldtext *italictext* **boldtext** *italictext ...*
This macro prints its arguments in alternating **boldface** and *italic* fonts. It takes up to six arguments.

.BR boldtext romantext **boldtext** romantext ... **]**
This macro prints its arguments in alternating **boldface** and Roman fonts. It takes up to six arguments.

.CO Print the string "COHERENT".

.DE End a display. It is always used with the macro **.DS**, described below.

.DS Start a display. The text that follows, up to the macro **.DE**, is read into a diversion. It is not adjusted. When the display is closed, **nroff** checks whether the present page has enough space left to hold the text. If the page does not, **nroff** jumps to the next page and prints the text there.

.DT Set the default tab stops. **tmac.an** by default set a tab stop every five characters (half-inch).

.HE Help end — close a section of help messages.

.HP Hanging paragraph. The new paragraph is separated by a space from the text that came above it; however, unlike the macro **.PP**, the new paragraph keeps the current level of indentation.

.HS Help start. All text from here up to the macro **.HE** is assumed to form a special help message, and is ignored.

.I [*string ...***]**
Use **italic** font. If used with one or more *string*s, prints them in italic. Otherwise, print all subsequent text in italic, up to the next explicit change of font.

.IB *italictext* **boldtext** *italictext* **boldtext ...**
This macro prints its arguments in alternating *italic* and **boldface** fonts. It takes up to six arguments.

.IP [*string* **[***indentation***]]**
Indented paragraph. If it has no arguments, it drops a space and indents subsequent text to the current level of indentation. If the macro has one argument, it uses that argument as a stub, and indents the following text by another five characters (one-half inch). If it has two arguments, it uses the first as a stub, and indents the subsequent text by the value given in the second argument.

.IR *italictext* romantext *italictext* romantext ... **]**
This macro prints its arguments in alternating *italic* and Roman fonts. It takes up to six arguments.

.PD [*distance***]**
Set the default interparagraph distance to *distance*. If invoked without an argument, it resets the interparagraph distance to the default, which is kept in the number register **PD**.

.PP Paragraph. The macro inserts a space into the output, and indent subsequent text by the default amount, which is the value kept in the number register **IN**.

.R Use Roman font. If used with one or more *string*s, prints them in Roman. Otherwise, print all subsequent text in Roman, up to the next explicit change of font.

.RB romantext **boldtext** romantext **boldtext ...**
This macro prints its arguments in alternating Roman and **boldface** fonts. It takes up to six arguments.

.RE End relative indentation. Subsequent text is printed at the previous level of indentation.

.RI romantext *italictext* romantext *italictext ...*
This macro prints its arguments in alternating Roman and *italic* fonts. It takes up to six arguments.

.RS [*indentation***]**
Start relative indentation. The indentation of subsequent text is increased by *indentation*. If invoked without an argument, indentation is increased by the default amount, as set by the number register **IN**.

.SH [*text***]**
Section heading. Set *text* in bold as the title of the section. If it is invoked without an argument, this macro uses the first line of the subsequent text as the section's title. Subsequent text is indented by the default amount, as set by the number register **IN**.

.TH [*first second third fourth fifth***]**
Header. This is the first macro to appear in any manual page. Its optional arguments are used in the header and footer of the manual page, as follows:

first The name of the manual page. It appears in the left and right corners of each page's header.

second This argument gives the section of the UNIX manual that holds the manual page.

third This argument appears in the center of each page's footer. It usually names the category of item that this manual page is documenting.

fourth This appears in the lower-left corner of each page.

fifth This appears in the center of each page's header.

.TP [*indentation***]**
Tagged paragraph. This macro resembles the macro **.IP**, except that it uses first line of subsequent text as the paragraph's stub.

tmac.an uses the following number registers to control its behavior. These are defined in the macro **.TH**; if you wish to reset them, do so *after* you have invoked macro **.TH**:

IN The default indentation.

LL The default line length.

PD The default distance between paragraphs.

Finally, **tmac.an** sets the following strings:

R The registered trademark symbol. This is equivalent to the special character **\(rg**.

Tm The trademark symbol. This is equivalent to the special character **\(tm**.

Files

/usr/lib/tmac.an — Macro package

See Also

ms, nroff, troff, Using COHERENT
nroff, The Text Processing Language, tutorial

man — Command

Display Lexicon entries
man [-dw] [*page ...***]**

man prints each manual *page* onto the standard output. This normally is an entry from the COHERENT Lexicon, although it can be a manual page from any other source as well.

When used with the option **-w**, it prints the path name of the file instead of printing the document itself. When used with option **-d**, it dumps a list of all available manual pages to the standard output device, for your perusal.

By default, **man** uses the pager **more** to display text. To use another pager, e.g., **scat**, define the environmental variable **PAGER**:

```
export PAGER="/bin/scat"
```

man normally searches for manual pages in the directory **/usr/man**. However, if the environmental variable **MANPATH** is set, **man** searches for manual pages in each directory that it names. **MANPATH** must name one or more directories, with directories separated by a colon ':'.

Index Files

To locate a manual page, **man** reads index files. It assumes that every file **/usr/man/*.index** is an index file; it then opens these files, and searches them for the manual entry you have requested.

Prior to release 4.2, an index file consisted of entries that had the format:

relative-path-name article-name

where *relative-path-name* gave the subdirectory and file in **/usr/man** that held the manual-page entry, *article_name* gave the name of the article as it appears in the Lexicon. Beginning with release 4.2, **man** uses index entries of the form:

relative-path-name article_name description

description gives a brief summary of the article. Fields must be separated by one more white-space characters. For example, entries

```
COHERENT1/bc          bc            Interactive calculator with arbitrary precision
LOCAL/chess           chess         Interactive chess program
```

associate manual-page file **/usr/man/COHERENT1/bc** with the Lexicon entry for the command **bc**. Likewise, rules for the user-written chess game **chess** are found in file **/usr/man/LOCAL/chess**.

man can read index entries prepared in either the "old" or the "new" form. We encourage you to use the new form, because this format also allows the index entries to be used by the command **apropos**.

Adding Manual-Page Entries

When writing new manual-page entries for COHERENT, we recommend that you place them into a subdirectory of **/usr/man**. This subdirectory should be uniquely named to avoid possible name-space collisions. A good rule of thumb is to name the subdirectory after the application with which it is associated. Also, when all manual-pages associated with a given application reside in a specific subdirectory, you can update the manual pages easily.

You should also add a uniquely named index file to directory **/usr/man** that identifies each of the newly added manual pages. This index file should use the "new" format described above; and its name should end with the suffix **.index**.

Files

/usr/man/* — Directories that hold manual pages **/usr/man/*.index** — Index files

See Also

apropos, commands, help, install, PAGER, Using COHERENT

Notes

The manual pages that are included with your release of the COHERENT system may include entries that have been corrected and updated since your COHERENT manual was printed. If there is a discrepancy between an on-line manual page and the printed COHERENT manual, you should assume that the on-line manual page is correct.

manifest constant — Overview

A **manifest constant** is a constant that is given a name so it can be defined differently under different computing environments. An example is **EOF**, the end-of-file marker, which has wildly different representations under different operating systems. Note, too, that numerals are manifest constants by definition.

The use of manifest constants in programs helps to ensure that code is portable by isolating the definition of these elements in a single header file, where they need to be changed only once.

The header file **limits.h** defines a set of macros that express certain numeric limits of COHERENT's implementation of C.

See Also

__DATE__, __FILE__, __LINE__, __STDC__, __TIME__, C preprocessor, EOF, EXIT_FAILURE, EXIT_SUCCESS, limits.h, macro, MB_CUR_MAX, NULL, RAND_MAX, portability, Programming COHERENT

math.h — Header File

Declare mathematics functions
#include <math.h>

math.h is the header file to be included with programs that use any of COHERENT's mathematics routines. It includes the following: definitions for mathematical functions; error return values, as used by the **errno** function; definitions of mathematical constants, e.g., **HUGE_VAL**; the definition of structure **cpx**, which describes complex variables; definitions of internal compiler functions; and, finally, prototypes of all mathematical functions.

See Also

header files, libm
ANSI Standard §7.5

MB_CUR_MAX — Manifest Constant

Largest size of a multibyte character in current locale
#include <stdlib.h>

MB_CUR_MAX is a manifest constant that is defined in the header **stdlib.h**. It expands into an expression that indicates the maximum number of bytes contained in a multibyte character in the current locale.

See Also

manifest constant
ANSI Standard, §7.10.7

mboot — Device Driver

Master boot block for hard disk

To be bootable, a COHERENT file system must contain a boot block (either **boot** or **mboot**). In addition, all hard disks must contain the master boot block **mboot** or an equivalent.

mboot is the master boot block for a hard-disk drive. It is compatible with, and therefore can replace, the IBM master boot block installed by the MS-DOS command **FDISK**. It must be installed in the first sector of the hard disk, as follows:

```
/etc/fdisk -b /conf/mboot /dev/at0x
/bin/sync
```

mboot searches its internal partition table (updated by the command **fdisk**) for an active partition. You can select an alternate partition by pressing 0 through 7 before the system selects the active partition. If the selected partition is of non-zero size with a valid partition boot block, COHERENT executes that partition's boot block. Otherwise, the prompt

```
Select partition [0-7]
```

appear, and the system waits for you to select the partition you want.

Files

/conf/mboot — Hard-disk master boot block

See Also

boot, device drivers, fdisk, mkfs

mcd — Device Driver

Device driver for Mitsumi CD-ROM drives

mcd is a device driver for a Mitsumi CD-ROM drive, models FX001, FX001 high speed, FX001D, or LU005, that is plugged into the Mitsumi controller card. It has major number 16.

Normally, this device driver is included in the kernel when you install or update COHERENT. To configure this driver, log in as the superuser **root**, and execute script **/etc/conf/mcd/mkdev**. Then run the command

```
/etc/conf/bin/idmkcoh -o coh.test
```

to build a test kernel that includes the driver.

Files

/dev/cdrom — Device applications read for CD-ROMs by default
/dev/rmcd0 — Character-special device for accessing Mitsumi CD-ROM

See Also

CD-ROM, device drivers, hai

mcmp() — Multiple-Precision Mathematics (libmp)

Compare multiple-precision integers
#include <mprec.h>
int mcmp(*a, b***)**
mint **a, *b;*

mcmp() compares the multiple-precision integers (or **mint**s) pointed to by *a* and *b*. It returns a signed integer less than, equal to, or greater than zero according to whether the value pointed to by *a* is less than, equal to, or greater than that pointed to by *b*.

See Also

libmp

mcopy() — Multiple-Precision Mathematics (libmp)

Copy a multiple-precision integer
#include <mprec.h>
void mcopy(*a, b***)**
mint **a, *b;*

mcopy() sets the multiple-precision integer (or **mint**) pointed to by *b* to the value pointed to by *a*.

See Also

libmp

mdevice — System Administration

Describe drivers that can be linked into kernel
/etc/conf/mdevice

File **mdevice** describes each device driver that can be linked into the COHERENT kernel. The command **idmkcoh** uses the information in this file when it builds and configures a new kernel.

mdevice contains one line for each driver or kernel component that can be linked into the kernel. Each line, in turn, consists of ten fields. The following describes the ten fields in order, from left to right:

1. *Name*
This field gives the name of the driver or component. Each name must uniquely identify the driver or component within the kernel. This field cannot be longer than eight characters.

2. *Function Flags*
This field holds a flag for each function (that is, entry point) within the driver or component. This field is used only by drivers or components that use the STREAMS or DDI/DKI interfaces; drivers that use the internal-kernel interface should place a hyphen '-' in this field. The legal flags are as follows:

o Open
c Close
r Read
w Write
i Ioctl
s Startup
x Exit
I Init
h Halt
p Poll — that is, **chpoll()**

3. *Miscellaneous Flags*
These flags give information about the device. They are set by most varieties of driver; the only exception is a STREAMS driver, for which only the flag **S** matters. The legal flags are as follows:

c Character device
b Block device
f Driver conforms to the DDI/DKI
o Driver has only one entry in **/etc/conf/sdevice**
r Driver is required in all configurations of the kernel
S STREAMS module; or STREAMS device when used with the 'c' flag
H Device driver controls hardware
C Driver uses interal-kernel (**CON**) interface

Note that the 'C' flag is unique to COHERENT, and cannot be used under other operating systems.

4. *Code Prefix*
This gives the "magic prefix" by which the kernel identifies the entry-point routines for this driver. In most instances, this is identical with the driver's name.

5. *Block Major-Device Number*
This gives the major-device number of this driver when it is accessed in block mode. In most instances, this and the following field are identical.

6. *Character Major-Device Number*
This gives the major-device number of this driver when it is accessed in character (raw) mode. In most instances, this and the preceding field are identical.

7. *Minor Device Numbers, Minimum*
This gives the smallest number that can be held by a minor-device number under this driver. Most drivers set this field to the lowest legal value, which is zero.

8. *Minor Device Numbers, Maximum*
This gives the largest number that can be held by a minor-device number under this driver. Most drivers set this field to the highest legal value, which is 255.

9. *DMA Channel*
This gives the DMA channel by which the device is accessed. Under COHERENT, this is always set to -1.

10. *CPU ID*
This gives the CPU that controls this driver, should the driver be running in a multiprocessor environment and be dedicated to a particular processor. Under COHERENT, this is always set to -1.

For an example of modifying this file, see the entry for **device drivers**.

Example

The following gives some example entries from **mdevice**:

```
1     2     3     4     5     6     7     8     9     10
###
# Example of an kernel components:  floating-point emulator and STREAMS
###
em87  -     -     em87  0     0     0     0     -1    -1
streams     I     -     streams     0     0     0     0-1 -1

###
# Example of a STREAMS driver:  note flags 'c' and 'S' both set in field 3
###
echo  -     cSf   echo  0     33    0     255   -1    -1

###
# Example DDI/DKI character driver:  Note that field 2 is initialized.
###
trace ocriI cfo   tr    0     34    0     255   -1    -1

###
# Example IK driver:  Note flag 'C' in field 3
###
at    -     CGHo  at    11    11    0     255   -1    -1
```

See Also

Administering COHERENT, device drivers, idmkcoh, mtune, sdevice, stune

mdiv() — Multiple-Precision Mathematics (libmp)

Divide multiple-precision integers
#include <mprec.h>
void mdiv(*a, b, q, r***)**
mint **a, *b, *q, *r;*

mdiv() divides the multiple-precision integer (or **mint**) pointed to by *a* with that pointed to by *b*. It writes the quotient and remainder into, respectively, *q* and *r*. *b* must not be zero. The results of the operation are defined by the following conditions:

1. $a = q * b + r$

2. The sign of *r* equals the sign of *q*

3. The absolute value of *r* is greater than the absolute value of *b*.

See Also

libmp

me — Command

MicroEMACS screen editor
me [-e *errorfile***] [-f** *bindfile***] [***textfile ...***]**

me is the command for MicroEMACS, the screen editor for COHERENT. With MicroEMACS, you can insert text, delete text, move text, search for a string and replace it, and perform many other editing tasks. MicroEMACS reads text from files and writes edited text to files; it can edit several files simultaneously, while displaying the contents of each file in its own screen window.

Screen Layout

Before you can use MicroEMACS, you must set the environmental variable **TERM** in your environment. If you do not set this variable explicitly in your **.profile** file, COHERENT sets it by default to **ansipc**. See the Lexicon entry **TERM** for details.

If the command **me** is used without arguments, MicroEMACS opens an empty buffer. If used with one or more file name arguments, MicroEMACS will open each of the files named, and display its contents in a window. If a file cannot be found, MicroEMACS will assume that you are creating it for the first time, and create an appropriately named buffer and file descriptor for it.

The last line of the screen is used to print messages and inquiries. The rest of the screen is portioned into one or more *windows* in which text is displayed. The last line of each window shows whether the text has been changed, the name of the buffer, and the name of the file associated with the window.

MicroEMACS notes its *current position*. It is important to remember that the current position is always to the *left* of the cursor, and lies *between* two letters, rather than at one letter or another. For example, if the cursor is positioned at the letter 'k' of the phrase "Mark Williams", then the current position lies *between* the letters 'r' and 'k'.

Commands and Text

The printable ASCII characters, from ' ' to '~', can be inserted at the current position. Control characters and escape sequences are recognized as *commands*, described below. A control character can be inserted into the text by prefixing it with **<ctrl-Q>** (that is, hold down the **<control>** key and type the letter 'Q').

There are two types of commands to remove text. *Delete* commands remove text and throw it away, whereas *kill* commands remove text but save it in the *kill buffer*. Successive kill commands append text to the previous kill buffer. Moving the cursor before you kill a line will empty the kill buffer, and write the line just killed into it.

Search commands prompt for a search string terminated by **<return>** and then search for it. Case sensitivity for searching can be toggled with the command **<esc>@**. Typing **<return>** instead of a search string tells MicroEMACS to use the previous search string.

Some commands manipulate words rather than characters. MicroEMACS defines a word as consisting of all alphabetic characters, plus '_' and '$'. Usually, a character command is a control character and the corresponding word command is an escape sequence. For example, **<ctrl-F>** moves forward one character and **<esc>F** moves forward one word.

MicroEMACS can handle blocks of text as well as individual characters, words, and lines. MicroEMACS defines a block of text as all the text that lies between the *mark* and the current position of the cursor. For example, typing **<ctrl-W>** kills all text from the mark to the current position of the cursor; this is useful when moving text from one file to another. When you invoke MicroEMACS, the mark is set at the beginning of the file; you can reset the mark to the cursor's current position by typing **<ctrl-@>**.

Using MicroEMACS with the Compiler

MicroEMACS can be invoked automatically by the compiler command **cc** to help you repair all errors that occur during compilation. The **-A** option to **cc** causes MicroEMACS to be invoked automatically when an error occurs. The compiler error messages are displayed in one window, the source code in the other, and the cursor is at the line on which the first error occurred. You can correct the errors one by one. To move to the next error in the list, type **<ctrl-X>>**; to move the previous error, type **<ctrl-X><**.

When have finished making corrections, exit from MicroEMACS by typing **<ctrl-Z>**, as usual; the compiler will automatically be re-invoked to re-compile the corrected source code. If more errors are found, MicroEMACS will be re-invoked with the new list of errors. This cycle will continue either until the file compiles without error, or until you break the cycle by typing **<ctrl-U> <ctrl-X> <ctrl-C>**.

The option **-e** to the **me** command allows you to invoke the error buffer by hand. For example, the commands

```
cc myprogram.c 2>errorfile
me -e errorfile myprogram.c
```

divert the compiler's error messages into **errorfile**, and then invokes MicroEMACS to let you correct them interactively.

The MicroEMACS Help Facility

MicroEMACS has a built-in help facility. With it, you can ask for information either for a word that you type in, or for a word over which the cursor is positioned. The MicroEMACS help file contains the bindings for all library functions and macros included with COHERENT.

For example, consider that you are preparing a C program and want more information about the function **fopen**. Type **<ctrl-X>?**. At the bottom of the screen will appear the prompt

```
Topic:
```

Type **fopen**. MicroEMACS will search its help file, find its entry for **fopen**, then open a window and print the following:

```
Open a stream for standard I/O
#include <stdio.h>
FILE *fopen (name, type) char *name, *type;
```

If you wish, you can kill the information in the help window and copy it into your program, to ensure that you prepare the function call correctly.

Consider, however, that you are checking a program written earlier, and you wish to check the call for a call to **fopen**. Simply move the cursor until it is positioned over one of the letters in **fopen**, then type **<esc>?**. MicroEMACS will open its help window, and show the same information it did above.

To erase the help window, type **<ctrl-X>1**.

Options

The following list gives the MicroEMACS commands. They are grouped by function, e.g., *Moving the cursor*. Some commands can take an *argument*, which specifies how often the command is to be executed. The default argument is 1. The command **<ctrl-U>** introduces an argument. By default, it sets the argument to four. Typing **<ctrl-U>** followed by a number sets the argument to that number. Typing **<ctrl-U>** followed by one or more **<ctrl-U>**s multiplies the argument by four.

Moving the Cursor

<ctrl-A> Move to start of line.

<ctrl-B> (Back) Move backward by characters.

<esc>B Move backward by words.

<ctrl-E> (End) Move to end of line.

<ctrl-F> (Forward) Move forward by characters.

<esc>F (Forward) Move forward by words.

<esc>G Go to an absolute line number in a file. Same as **<ctrl-X>G**.

<ctrl-N> (Next) Move to next line.

<ctrl-P> (Previous) Move to previous line.

<ctrl-V> Move forward by pages.

<esc>V Move backward by pages.

<ctrl-X>= Print the current position.

<ctrl-X>G Go to an absolute line number in a file. Can be used with an argument; otherwise, it will prompt for a line number. Same as **<esc>G**.

<ctrl-X>[Go to matching C delimiter. For example, if the cursor is positioned under the character '{', then typing **<ctrl-X>[** moves the cursor to the next '}'. Likewise, if the cursor is positioned under the character }, then typing **<ctrl-X>[** moves the cursor to the first preceding '{'. MicroEMACS recognizes the delimiters [,], {, }, (,), /*, and */.

<ctrl-X>] Toggle reverse-video display of matching C delimiters. For example, if reverse-video displaying is toggled on, then whenever the cursor is positioned under a '}' MicroEMACS displays the first preceding '{' in reverse video (should it be on the screen). MicroEMACS recognizes the delimiters [,], {, }, (,), /*, and */.

<esc>! Move the current line to the line within the window given by *argument*; the position is in lines from the top if positive, in lines from the bottom if negative, and the center of the window if zero.

<esc>< Move to the beginning of the current buffer.

<esc>> Move to the end of the current buffer.

Killing and Deleting

<ctrl-D> (Delete) Delete next character.

<esc>D Kill the next word.

<ctrl-H> If no argument, delete previous character. Otherwise, kill *argument* previous characters.

<ctrl-K> (Kill) With no argument, kill from current position to end of line; if at the end, kill the newline. With argument set to one, kill from beginning of line to current position. Otherwise, kill *argument* lines forward (if positive) or backward (if negative).

<ctrl-W> Kill text from current position to mark.

<ctrl-X><ctrl-O>
Kill blank lines at current position.

<ctrl-Y> (Yank back) Copy the kill buffer into text at the current position; set current position to the end of the new text.

<esc><ctrl-H>
Kill the previous word.

<esc><DEL>
Kill the previous word.

<DEL> If no argument, delete the previous character. Otherwise, kill *argument* previous characters.

Windows

<ctrl-X>1 Display only the current window.

<ctrl-X>2 Split the current window into two windows. This command is usually followed by **<ctrl-X>B** or **<ctrl-X><ctrl-V>**.

<ctrl-X>N (Next) Move to next window.

<ctrl-X>P (Previous) Move to previous window.

<ctrl-X>Z Enlarge the current window by *argument* lines.

<ctrl-X><ctrl-N>
Move text in current window down by *argument* lines.

<ctrl-X><ctrl-P>
Move text in current window up by *argument* lines.

<ctrl-X><ctrl-Z>
Shrink current window by *argument* lines.

Buffers

<ctrl-X>B (Buffer) Prompt for a buffer name, and display the buffer in the current window.

<ctrl-X>K (Kill) Prompt for a buffer name and delete it.

<ctrl-X><ctrl-B>
Display a window showing the change flag, size, buffer name, and file name of each buffer.

<ctrl-X><ctrl-F>
(File name) Prompt for a file name for current buffer.

<ctrl-X><ctrl-R>
(Read) Prompt for a file name, delete current buffer, and read the file.

<ctrl-X><ctrl-V>
(Visit) Prompt for a file name and display the file in the current window.

Saving Text and Exiting

<ctrl-X><ctrl-C>
Exit without saving text.

<ctrl-X><ctrl-S>
(Save) Save current buffer to the associated file.

<ctrl-X><ctrl-W>
(Write) Prompt for a file name and write the current buffer to it.

<ctrl-Z> Save current buffer to associated file and exit.

Compilation Error Handling

<ctrl-X>> Move to next error.

<ctrl-X>< Move to previous error.

Search and Replace

<ctrl-R> (Reverse) Incremental search backward; a pattern is sought as each character is typed.

<esc>R (Reverse) Search toward the beginning of the file. Waits for entire pattern before search begins.

<ctrl-S> (Search) Incremental search forward; a pattern is sought as each character is typed.

<esc>S (Search) Search toward the end of the file. Waits for entire pattern before search begins.

<esc>% Search and replace. Prompt for two strings; then search for the first string and replace it with the second.

<esc>/ Search for next occurrence of a string entered with the **<esc>S** or **<esc>R** commands; this remembers whether the previous search had been forward or backward.

<esc>@ Toggle case sensitivity for searches. By default, searches are case insensitive.

Keyboard Macros

<ctrl-X>(Begin a macro definition. MicroEMACS collects everything typed until the next **<ctrl-X>)** for subsequent repeated execution. **<ctrl-G>** breaks the definition.

<ctrl-X>) End a macro definition.

<ctrl-X>E (Execute) Execute the keyboard macro.

<ctrl-X>M Bind a newly created keyboard macro to a given keystroke or set of keystrokes.

Flexible Key Bindings

<ctrl-X>R Replace one binding with another.

<ctrl-X>X Rebind the prefix (meta) keys, and the multiple-execution key **<ctrl-U>**.

<ctrl-X>S Prompt for a file name, and write all flexible keybindings and macros into it. This command also saves information about how you have configured MicroEMACS; for example, it notes whether you have turned on word-wrapping.

<ctrl-X>L Prompt for a file name, and read all flexible keybindings and macros from it.

<ctrl-X>I Rebind current macro to the initialization macro.

By default, MicroEMACS checks for the existence of file **$HOME/.emacs.rc** and executes it if found. The **-f** option lets you specify an alternate file of keybindings macros from the **me** command line. After loading the file, MicroEMACS then executes the initialization macro, if one exists. For example, to load the keybindings file **bindings** and edit file **textfile**, use the command:

```
me -f bindings textfile
```

Change Case of Text

<esc>C (Capitalize) Capitalize the next word.

<ctrl-X><ctrl-L>
(Lower) Convert all text from current position to mark into lower case.

<esc>L (Lower) Convert the next word to lower case.

LEXICON

<ctrl-X><ctrl-U>
(Upper) Convert all text from current position to mark into upper case.

<esc>U (Upper) Convert the next word to upper case.

White Space

<ctrl-I> Insert a tab. Default behavior is to move the cursor to the nearest 8's boundary; for example, if the cursor is in the 62nd column on the screen, pressing **<ctrl-I>** moves it to column 64.

When used with a positive argument, change the behavior of the tab key. For example, **<ctrl-U>4<ctrl-I>** commands MicroEMACS to insert enough spaces for a tab key to reach a four's boundary.

When used with a negative argument, change the behavior of the tab character. For example, **<ctrl-U>-4<ctrl-I>** says that a tab character on a file will take you to the nearest 4's boundary. Thus, if you have a file with tabs in it and you use '-4', the appearance of the file on the screen will change; but if you use '4' the appearance of the file on the screen will not change.

To change the default size of a tab, set the environmental variable **TABSIZE** to a value other than eight.

<ctrl-J> Insert a new line and indent to current level. This is often used in C programs to preserve the current level of indentation.

<ctrl-M> (Return) If the following line is not empty, insert a new line; if empty, move to next line.

<ctrl-O> Open a blank line; that is, insert newline after the current position.

<tab> With argument, set tab fields at every *argument* characters. An argument of zero restores the default of eight characters. Setting the tab to any character other than eight causes space characters to be set in your file instead of tab characters.

Send Commands to Operating System

<ctrl-C> Suspend MicroEMACS and execute a subshell. Typing **<ctrl-D>** returns you to MicroEMACS and allows you to resume editing.

<ctrl-X>! Prompt for a shell command and execute it.

These commands recognize the shell variable **SHELL** to determine the shell to which it should pass the command.

Setting the Mark

<ctrl-@> Set mark at current position.

<esc>. Set mark at current position.

Help Window

<ctrl-X>? Prompt for word for which information is needed.

<esc>? Search for word over which cursor is positioned.

<esc>2 Erase help window.

Miscellaneous

<ctrl-G> Abort a command.

<ctrl-L> Redraw the screen.

<ctrl-Q> (Quote) Insert the next character into text; used to insert control characters.

<esc>Q Quote a character by numeric value. When you type this command, MicroEMACS prompts you for a numeric value, in decimal. It then inserts into your text the character whose value you type. This command is useful when you wish to enter characters with the high bit set.

<ctrl-T> Transpose the characters before and after the current position.

<ctrl-U> Specify a numeric argument, as described above.

<ctrl-U><ctrl-X><ctrl-C>
Abort editing and re-compilation. Use this command to abort editing and return to COHERENT when you are using the **-A** option to the **cc** command.

<ctrl-X>H Use word-wrap on a region.

<ctrl-X>F Set word wrap to *argument* column. If argument is one, set word wrap to cursor's current position.

<ctrl-X><ctrl-X>
Mark the current position, then jump to the previous setting of the mark. This is useful when moving text from one place in a file to another.

Diagnostics

MicroEMACS prints error messages on the bottom line of the screen. It prints informational messages (enclosed in square brackets '[' and ']' to distinguish them from error messages) in the same place.

MicroEMACS manipulates text in memory rather than in a file. The file on disk is not changed until you save the edited text. MicroEMACS prints a warning and prompts you whenever a command would cause it to lose changed text.

See Also

commands, ed, elvis, ex, sed, TERM, vi

Notes

Because MicroEMACS keeps text in memory, it does not work for extremely large files. It prints an error message if a file is too large to edit. If this happens when you first invoke a file, you should exit from the editor immediately. Otherwise, your file on disk will be truncated. If this happens in the middle of an editing session, however, delete text until the message disappears, then save your file and exit. Due to the way MicroEMACS works, saving a file after this error message has appeared will take more time than usual.

MicroEMACS is based upon the public domain editor by David G. Conroy.

mem — Device Driver

Physical memory file

The special file **/dev/mem** permits a program to read and write to the physical memory of the host computer, just as it reads and writes into an ordinary file. The location where I/O will occur can be positioned to any valid byte address by a call to **lseek()**. Note that **ps** and related commands use **/dev/kmem**, which manipulates the kernel's data space.

Commands may examine or change addresses in physical memory. Addresses to use when changing the system itself normally are obtained from the system load module (**/coherent**) name list, so that they always reflect the currently running version of the system.

Files

/dev/mem

See Also

clock, cmos, core, device drivers, lseek, ps

Diagnostics

On an error, such as nonexistent memory location, **mem** returns -1.

memccpy() — String Function (libc)

Copy a region of memory up to a set character
#include <string.h>
char ***memccpy(***dest**,** src**,** c**,** n**)**
char **dest***,* ***src***; **unsigned int** c**,** n**;**

memccpy() copies characters from *src* to *dest*, stopping when either it finds the first occurrence of character *c* or it has copied *n* characters. Unlike the routines **strcpy()** and **strncpy()**, **memcpy()** copies from one region to another. Therefore, it will not halt automatically when it encounters NUL.

memccpy() returns a pointer to the first location after character *c* in *dest*, or NULL if character *c* was not found.

See Also

libc, memcpy(), strcpy(), strncpy(), string.h

Notes

memccpy() is not part of the ANSI Standard. Use of this library routine may restrict portability.

If *dest* and *src* overlap, the behavior of **memccpy()** is undefined. *dest* should point to enough reserved memory to hold *n* bytes of data; otherwise, data corruption may result.

memchr() — String Function (libc)

Search a region of memory for a character
#include <string.h>
char *memchr(*region***,** *character***,** *n***)**
char ****region*; int** *character***; unsigned int** *n***;**

memchr() searches the first *n* characters in *region* for *character*. It returns the address of *character* if it is found, or NULL if it is not.

Unlike the string-search function **strchr()**, **memchr()** searches a region of memory. Therefore, it does not stop when it encounters a null character.

Example

The following example deals a random hand of cards from a standard deck of 52. The command line takes one argument, which indicates the size of the hand you want dealt. It uses an algorithm published by Bob Floyd in the September 1987 *Communications of the ACM*.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#define DECK 52

main(argc, argv)
int argc; char *argv[];
{
      char deck[DECK], *fp;
      int  deckp, n, j, t;

      if(argc != 2 ||
         52 < (n = atoi(argv[1])) ||
         1 > n) {
               printf("usage: memchr n # where 0 < n < 53\n");
            exit(EXIT_FAILURE);
      }

      /* exercise rand() to make it more random */
      srand((unsigned int)time(NULL));
      for(j = 0; j < 100; j++)
            rand();

      deckp = 0;
      /* Bob Floyd's algorithm */
      for(j = DECK - n; j < DECK; j++) {
            t = rand() % (j + 1);
            if((fp = memchr(deck, t, deckp)) != NULL)
                  *fp = (char)j;
            deck[deckp++] = (char)t;
      }

      for(t = j = 0; j < deckp; j++) {
            div_t card;
```

```
            card = div(deck[j], 13);
            t += printf("%c%c  ",
                  /* note useful string addressing */
                  "A23456789TJQK"[card.rem],
                  "HCDS"[card.quot]);

            if(t > 50) {
                  t = 0;
                  putchar('\n');
            }
      }

      putchar('\n');
      return(EXIT_SUCCESS);
}
```

See Also

libc, strchr(), string.h
ANSI Standard, §7.11.5.1

memcmp() — String Function (libc)

Compare two regions
#include <string.h>
int memcmp(*region1***,** *region2***,** *count***)**
char **region1***; **char** **region2***; **unsigned int** *count*;

memcmp() compares *region1* with *region2* character by character for *count* characters.

If every character in *region1* is identical to its corresponding character in *region2*, then **memcmp()** returns zero. If it finds that a character in *region1* has a numeric value greater than that of the corresponding character in *region2*, then it returns a number greater than zero. If it finds that a character in *region1* has a numeric value less than less that of the corresponding character in *region2*, then it returns a number less than zero.

For example, consider the following code:

```
char region1[13], region2[13];
strcpy(region1, "Hello, world");
strcpy(region2, "Hello, World");
memcmp(region1, region2, 12);
```

memcmp() scans through the two regions of memory, comparing **region1[0]** with **region2[0]**, and so on, until it finds two corresponding "slots" in the arrays whose contents differ. In the above example, this will occur when it compares **region1[7]** (which contains 'w') with **region2[7]** (which contains 'W'). It then compares the two letters to see which stands first in the character table used in this implementation, and returns the appropriate value.

memcmp() differs from the string comparison routine **strcmp()** in a number of ways. First, **memcmp()** compares regions of memory rather than strings; therefore, it does not stop when it encounters a NUL.

Also, you can use **memcmp()** to compare an **int** array with a **char** array, because **memcmp()** simply compares areas of data.

See Also

libc, strcmp(), string.h
ANSI Standard, §7.11.4.1

memcpy() — String Function (libc)

Copy one region of memory into another
#include <string.h>
char ***memcpy(***region1***,** *region2***,** *n***)**
vaddr_t *region1*;
vaddr_t *region2*;
unsigned int *n*;

memcpy() copies *n* characters from *region2* into *region1*. Unlike the routines **strcpy()** and **strncpy()**, **memcpy()** copies from one region to another. Therefore, it will not halt automatically when it encounters NUL.

memcpy() returns *region1*.

Example

The following example copies a structure and displays it.

```
#include <string.h>
#include <stdio.h>

struct stuff {
      int a, b, c;
} x, y;

main()
{
      x.a = 1;
      /* this would stop strcpy or strncpy. */
      x.b = 0;
      x.c = 3;

      /* y = x; would do the same */
      memcpy(&y, &x, sizeof(y));
      printf("a =%d, b =%d, c =%d\n", y.a, y.b, y.c);
      return(EXIT_SUCCESS);
}
```

See Also

libc, strcpy(), string.h
ANSI Standard, §7.11.2.1

Notes

If *region1* and *region2* overlap, the behavior of **memcpy()** is undefined. *region1* should point to enough reserved memory to hold *n* bytes of data; otherwise, code or data will be overwritten.

memmove() — String Function (libc)

Copy region of memory into area it overlaps
#include <string.h>
char ***memmove(***region1**,** region2**,** count**)**
char **region1**, **char** **region2**, **unsigned int** count**;**

memmove() copies *count* characters from *region2* into *region1*. Unlike **memcpy()**, **memmove()** correctly copies the region pointed to by *region2* into that pointed by *region1* even if they overlap. To "correctly copy" means that the overlap does not propagate, not that the moved data stay intact. Unlike the string-copying routines **strcpy()** and **strncpy()**, **memmove()** continues to copy even if it encounters a NUL.

memmove() returns *region1*.

Example

The following example rotates a block of memory by one byte.

```
#include <string.h>
#include <stddef.h>
#include <stdio.h>

char *
rotate_left(region, len)
char *region; size_t len;
{
      char sav;

      sav = *region;
      /* with memcpy this might propagate the last char */
      memmove(region, region + 1, --len);
      region[len] = sav;
      return(region);
}
```

```
char nums[] = "0123456789";
main(void)
{
      printf(rotate_left(nums, strlen(nums)));
      return(EXIT_SUCCESS);
}
```

See Also

libc, string.h
ANSI Standard, §7.11.2.2

Notes

region1 should point to enough reserved memory to hold the contents of *region2*. Otherwise, code or data will be overwritten.

memok() — General Function (libc)

Test if the arena is corrupted
int
memok();

The library function **memok()** checks to see if the arena has been corrupted. It returns one if the arena is sound, and zero if it has been corrupted.

Example

The following example purposely corrupts the arena, to demonstrate **memok()**. Please note that this is not a recommended programming practice.

```
extern char *malloc();
main()
{
      char *p;

        p = malloc(2);                      /* get 2 bytes of memory */
        printf("Arena is %s\n", memok() ? "OK" : "bad");
        strcpy(p, "too long");              /* clobber memory */
        printf("Arena is %s\n", memok() ? "OK" : "bad");
}
```

See Also

arena, calloc(), libc, malloc(), realloc()

memset() — String Function (libc)

Fill an area with a character
#include <string.h>
char ***memset(***buffer**,** character**,** n**)**
char **buffer**; **int** character**; unsigned int** n**;**

memset() fills the first *n* bytes of the area pointed to by *buffer* with copies of *character*. It casts *character* to an **unsigned char** before filling *buffer* with copies of it.

memset() returns the pointer *buffer*.

Example

The following example fills an area with 'X', and prints the result.

```
#include <stdio.h>
#include <string.h>
#define BUFSIZ 20

main()
{
      char buffer[BUFSIZ];
```

```
        /* fill buffer with 'X' */
        memset(buffer, 'X', BUFSIZ);

        /* append null to end of buffer */
        buffer[BUFSIZ-1] = '\0';

        /* print the result */
        printf("%s\n", buffer);
        return(EXIT_SUCCESS);
}
```

See Also

libc, string.h
ANSI Standard, §7.11.6.1

mesg — Command

Permit/deny messages from other users
mesg [ny]

Normally, a user can communicate with other users by using the commands **msg** and **write**.

In certain situations, it is useful to suppress messages from other users. Therefore, COHERENT supplies the command **mesg**, which, lets you permit or suppress messages from other users. The argument **y** allows messages, whereas argument **n** disallows messages. With no argument, **mesg** tells you whether you can receive messages (as **yes** or **no**) without changing the message state.

Files

/dev/*

See Also

commands, msg, write

Notes

The owner-execute mode bit of the user's **tty** indicates whether messages are allowed.

min() — Multiple-Precision Mathematics (libmp)

Read multiple-precision integer from stdin
#include <mprec.h>
void min(*a***)**
mint ******a***;**

min() reads a multiple-precision integer (or **mint**) from the standard input and writes it at the address held by *a*. The base of the **mint** is indicated by the value held in the external variable **ibase**.

min() accepts leading blanks and an optional leading minus sign; the number is terminated by the first non-legal digit.

See Also

libmp

minit() — Multiple-Precision Mathematics (libmp)

Condition global or auto multiple-precision integer
#include <mprec.h>
void minit(*a***)**
mint ******a***;**

minit() helps to create a multiple-precision integer (or **mint**). If a new **mint** is declared to be global or automatic, you must call **minit()** before using the variable. This prevents garbage values in the newly created **mint** structure from causing chaos. A **mint** conditioned by **minit()** has no value; however, it may be used to receive the result of an operation.

See Also

libmp

minor number — Definition

Device numbering

A *minor number* specifies the device or type of device to use. COHERENT uses the minor number of a given device in a driver-specific manner. For example, a hard-disk driver may use the minor number to select a disk drive and partition.

Every COHERENT device has a device number associated with it. It is of type **dev_t**, as defined in **<sys/types.h>**. The macro **minor()** in **<sys/stat.h>** extracts the minor number from a given device number.

See Also

device drivers, major number, stat.h

mintfr() — Multiple-Precision Mathematics (libmp)

Free a multiple-precision integer
#include <mprec.h>
void mintfr(*a***)**
mint ******a***;**

mintfr() frees the memory used by a **mint**.

See Also

libmp

mitom() — Multiple-Precision Mathematics (libmp)

Reinitialize a multiple-precision integer
#include <mprec.h>
void mitom(*n, a***)**
mint ******a***; int** *n***;**

mitom() reinitializes the existing multiple-precision integer (or **mint**) pointed to by *a* to *n*.

See Also

libmp

mkdbm — Command

Build a data base for smail
/usr/lib/mail/mkdbm [-d] [-f] [-n] [-o *output-file***] [-v] [-y] [***file ...***]**

The command **mkdbm** generates a data base for **smail**.

It forms the data key from the characters up to, but not including, a colon (':') or white-space character. The data after the colon or white-space character forms the value associated with the key. You can use **mkdbm** to produce data-base files that can then be read by a **smail** router **pathalias** or its director **alias-file**. By default, the router and director are configured to use the DBM file-access protocol. (For information on routers and directors, see the Lexicon entries for **routers** and **directors**.)

For some data bases, you can use **mkline** to form single-line records whose comments and extra white space are removed. The generated data base contains a single NUL character at the end of each key and value. It also generates a single record that contains a '@' as a key and value; it does so for compatibility with the Berkeley **sendmail** command's alias files.

mkdbm recognizes the following command-line options:

-d Suppress writing the extra '@' record.

-f Fold the key to lower case before storing it within the data base.

-n Suppress writing a NUL character at the end of each line. Please note that this option is incompatible with **smail**'s method of accessing the data-base file.

-o *output-file*
Write output into *output-file*. This option also sets the name for the data base. If you do not use this option, **mkdbm** names the output data base after its first *file* argument. If, in addition, the command line does not name an input *file*, **mkmf** names the output file **dbm**.

-v Write statistics to the standard output.

-y Create an output file that is compatible with the Sun Yellow Pages (YP) system. This obviates the need for keeping a copy of **sendmail** on your system to maintain a YP-alias data base.

If its command line does not name an input *file*, **mkdbm** reads the standard input. **mkdbm** also reads the standard input if a *file* is named '-'; in this way, it can mix data read from the standard input with material read from files.

Calling **mkdbm** with the arguments **-ynd** generates a data base that is compatible with regular YP data bases. Using just the argument **-y** generates a data base that is compatible with the YP **mail.alias** data base.

As it creates the data base, **mkdbm** builds temporary files in the same directory in which it eventually builds the output files. When it has completed its work, **mkdbm** removes all data-base files that have the target name, sleeps for one or two seconds, then moves the newly written temporary data-base files to the target names. This method of writing a data-base is not compatible with the locking method used by Berkeley command **sendmail**.

Example

As an example of the use of **mkdbm** consider a file named **paths**, which contains the routing information:

```
.COM sun!%s
Stargate.COM ames!cmcl2!uiucdcs!stargate!%s
ames ames!%s
.ATT.COM mtune!%s
mtune mtune!%s
```

Given this file, the command

```
mkdbm -f paths
```

produces a data base in the files **paths.pag** and **paths.dir** that contains the above entries but with the keys shifted into lower case. For example, one entry will contain the key **stargate.com** with an associated value of:

```
ames!cmcl2!uiucdcs!stargate!%s
```

Files

dbmXXXXXX.pag
dbmXXXXXX.dir
Temporary files, created in the same directory as the output files.

See Also

commands, libgdbm, mail [overview], mkline, pathalias, smail

Notes

mkdir — Command

Create a directory
mkdir [-rp] *directory*

mkdir creates *directory*. Files or directories with the same name as *directory* must not already exist. **directory** will be empty except for the entries '.', the directory's link to itself, and '..', its link to its parent directory.

Option **-r** creates directories recursively. For example, the command

```
mkdir -r /foo/bar/baz
```

creates directory **foo** in **/**; then creates directory **bar** in the newly created directory **foo**; and finally creates directory **baz** in the newly created directory **bar**.

Option **-p** behaves exactly the same as **-r**. COHERENT includes it for use by scripts imported from other operating systems.

See Also

commands, mkdir(), rmdir

Diagnostics

mkdir fails and prints an error message if you do not have permission to write into directory in which you are attempting to create a new directory or if the directory in which you attempted to create a new directory does not exist.

mkdir() — System Call (libc)

Create a directory
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(*path***,** *mode***)**
char **path***;*
int *mode***;**

The COHERENT system call **mkdir()** creates the directory specified by *path* and gives it the file mode specified by *mode*. If *path* is relative (that is, it doesn't begin with a '/' character), **mkdir()** creates the directory relates to the current directory of the process that calls **mkdir()**. If *path* is absolute (i.e., begins with a '/'), **path** specifies a directory to be created relative to the root directory for this process. See Lexicon article **chroot()** for details. If *path* specifies more than one directory level, all parent names specified must exist, must be accessible by the calling process, and actually must be directories.

Argument *mode* is formed by logically OR'ing permissions constants found in header file **<sys/stat.h>**. These constants begin with **S_** and determine the permissions for the directory. See the Lexicon article **stat.h** for details.

If the directory is successfully created, **mkdir()** returns zero. If an error occurs, **mkdir()** returns -1 and sets **errno** to an appropriate value.

See Also

libc, mkdir, rmdir, rmdir(), stat.h
POSIX Standard, §5.4.1

mkfifo() — System Call (libc)

Create a FIFO
#include <sys/types.h>
#include <unistd.h>
int mkfifo(*path***,** *mode***)**
const char **path***;* **mode_t** *mode***;**

mkfifo() calls **mknod()** to create a FIFO. *path* points to the full path name of the FIFO to create. *mode* gives the mode into which the FIFO is to be opened. **mkfifo()** ignores the bits in *mode* other than the file-permission bits. The file permission bits of *mode* are modified by the process's file-creation mask; for details, see the Lexicon entry for **umask()**.

mkfifo() sets the ownership of the file FIFO's to the process's effective user identifier, and sets the FIFO's group identifier to the process's effective group identifier.

If all goes well, **mkfifo()** returns zero. If an error occurs, it returns -1 and sets **errno** to an appropriate value.

See Also

libc, libsocket, named pipe, pipe(), unistd.h
POSIX Standard, §5.4.2

mkfnames — Command

Generate data base of user names
/usr/bin/mkfnames [*namefile ...***]**

The script **mkfnames** generates a data base of users' names and addresses. It reads the contents of *namefile*, which contains each user's names and her e-mail address; invokes the command **nptx** to generate permutations of

the users' names; then sorts the output of **nptx** and writes the sorted output onto the standard output. If no *namefile* is named on the command line, **mkfnames** reads the file **/etc/passwd**, and parses its contents into the form required by command **nptx**.

mkfnames is usually used to generate the file **/usr/lib/mail/fullnames**, which the mail system uses to translate a person's name into her e-mail address. If more than one login account has the same part of a name (i.e., the same last name), the first login name in alphabetical order will be used.

See Also

commands, mail [overview], nptx, smail

mkfs — Command

Make a new file system
/etc/mkfs [-b *boot***] [-d] [-f** *name***] [-i** *inodes***] [-m** *arg***] [-n** *arg***] [-p** *pack***]** *filesystem proto*

mkfs makes a new file system. *filesystem* names the file (normally a block special file) where the new file system will reside. The contents of the newly created file system are described in *proto*. *proto* can be either a number or a file name.

If *proto* is a number, **mkfs** creates an empty file system (containing only a root directory) of the size in blocks given by *proto*. The number of i-nodes is calculated as a percentage of this number. The command

```
/etc/mkfs /dev/fha0 2400
```

creates a file system on a high-density, 5.25-inch diskette in drive 0. If the disk is a high-density, 3.5-inch diskette, use the command:

```
/etc/mkfs /dev/fva0 2880
```

If *proto* is a file name, however, the contents of that file will be used as a prototype for modeling the new file system. This prototype file must be laid out in the following manner:

bootstrap_file_name file_system_name device_name
no._of_blocks no._of_i-nodes n m
%b XX XX XX
...
directory_name
    directory_name mode user_id group_id contents
    ...
    $
$

Each line is described below.

The first line has three fields. Field 1, *bootstrap_file_name*, contains the name of a file that holds the boot strap, which must fit into block 0 of the disk. Field 2, *file_system_name*, gives the name of the file system; and field 3, *device_name*, gives the name of file system's physical device (for example, **/dev/hd1**). Only the first six characters in field 2 and the first 11 in field 3 are significant; all characters after them are ignored.

The second line contains four fields. Field 1, *no._of_blocks*, gives the size of the file system in blocks; field 2, *no_of_i-nodes*, gives the number of i-nodes in the file system. Because each file or directory requires one i-node, this number represents the limit on the number of files that may be created in the file system. A ratio of seven blocks per i-node generally works well.

Fields 3 and 4 control free list interleaving on your disk. *n* is the size of a "virtual cylinder": **fsck** allocates all the blocks on one virtual cylinder before it advances to the next virtual cylinder. The value of *n* must be less than or equal to 255, and should evenly divide the actual size of a cylinder on the device. *m* tells the system how many blocks to skip each time it increments a free list block number, i.e., the free list "interleave"; *n* mod *m* must be zero. Choosing an optimal interleave value may improve system performance for the device. The optimal values for *n* and *m* are hardware-specific and can be determined by experimentation.

Next, the third line and following begin with **%b**. These list the bad blocks on your storage device. One or more block numbers may appear on each line, separated by white space. These blocks are allocated to the bad block file (i-node 1).

The remaining lines in the *proto* file define the names, modes, and contents of the directories and files in the file system. These lines are divided into fields separated by white space (blanks or tabs) as follows:

- The first field names the file or directory to be created. This field is missing on the first line, which describes the root directory of the file system.
- The second field describes the mode of the file, which is six characters long. The first character gives the file type, that is, whether the file is ordinary ('-'), directory ('d'), block special ('b'), or character special ('c'). The second character is 'u' for set user id on execution, and '-' otherwise. The third character is 'g' for set group id on execution, and '-' otherwise. Characters 4 through 6 specify permissions in octal; for example, **644** specifies read and write permission for the owner, read permission for other users from the same group, and read permission for users from other groups.
  If the above file type were a directory, subsequent files are recursively defined under that directory, until the current level of directory is terminated by a line containing a '$' character.
- The next two fields specify the owner's numerical user id and group id.
- The last field describes file contents. For a directory, it is not needed. For an ordinary file, it is the name of a COHERENT file that will be copied into the newly created file. For block or character-special files, there are two fields that specify the numbers of the major and minor devices.

Finally, each directory's description and the entire *proto* file must terminate with dollar signs '$'.

The *proto* file need not contain all of the above fields. However, it must contain the name of the boot block (line 1), the number of blocks and the number of i-nodes (line 2), the list of bad blocks, the name of at least one directory, and the dollar sign that ends the file.

Command-line Options

mkfs recognizes the following command-line options:

-b *boot*
Specifies the file to use as the "bootstrap" for the file system.

-d Preserve file dates and times on the new file system.

-f *name*
Label the file system with the given *name*. *name* must be less than seven characters in length.

-i *inodes*
Use *inodes* as the number of inodes for the file system.

-m *arg*
Set the number of blocks to skip when incrementing virtual block number. This is the same as the *m* option as set on line 2 of the prototype file. You can use this option if you choose not to use a prototype file.

-n *arg*
Set the size of a "virtual cylinder". This is the same as the *n* option as set on line 2 of the prototype file. You can use this option if you choose not to use a prototype file.

-p *pack*
Set the file system "pack name" to *pack*. *pack* must be less than seven characters in length.

Example

The following example specifies a proto file for a high-density, 5.25-inch floppy disk; note that this floppy disk is faulty and contains a number of bad blocks:

```
/conf/boot.fha
2400 100
%b 55
%b 185 86
d--755 3 1
       coherent ---644 3 1 /coherent
       tmp         d--777 3 1
       $
       bin         d--755 3 1
                   mail       -u-755 0 1 /bin/mail
       $
       dev         d--755 3 1
                   tty30      c--644 0 1 3 0
                   tty35      c--644 0 1 3 5
                   mt0        b--600 0 1 12 0
       $
$
```

You can use the command **badscan** to draw up the list of bad blocks on your disk and create a skeleton *proto* file.

See Also

badscan, chmod, commands, fsck, mount, restor

Notes

When the command **fsck** checks a file system, it stores files that it cannot decypher into directory **lost+found**. However, **fsck** cannot modify a file system during its work. This rule was adopted to prevent **fsck** from attempting to modify a corrupt file system, and so making matters worse. However, this means that (among other things) **fsck** cannot change the size of directory **lost+found**. Thus, if more files are detached from the file system than **lost+found** can hold, **fsck** must delete them outright. If your newly created file system will hold a large number of transient files (e.g., a news system), you should increase the size of **lost+found** so that it has a fighting chance of holding all detached files that **fsck** finds. For example, the following script expands **/lost+found** so it can hold up to 500 files:

```
su root
for i in `from 1 to 500`
do
      touch /lost+found/$i
done
rm /lost+found/*
```

Run this script for each file system whose **lost+found** directory you wish to expand. For example, if you have a file system mount on directory **/u**, run this script for directory **/u/lost+found** instead of for **/lost+found**.

mkhpath — Command

Build a pathalias data base from a hosts table
/usr/lib/mail/mkhostpath [-d] [-c *cost***] [-g** *gateway***] [-n** *netname***] [- |** *filename* **]**

The script **mkhpath** reads a hosts table and constructs routes to that network. *filename* holds the hosts table; if it is '-', **mkhpath** reads the standard input. If the command line does not name a *filename*, **mkhpath** reads file **/etc/hosts**.

mkhpath assumes its input to be in the format of the data-base file **/etc/hosts**. It ignores the first field (Internet address) and any domain-based name (any field containing a '.'). It also ignores the hosts **localhost** and **loghost**, and all comment lines (those that begin with a '#').

mkhpath recognizes the following command-line options:

-c *cost* Set the cost of accessing the gateway to be *cost*. *cost* can be any cost expression recognized by the command **pathalias**. **mkhpath** ignores this option if you do not also use option **-n**.

-d Print a line only if it contains a domain host name. This is useful for ignoring test lines.

-g *gateway*
Make *gateway* the gateway to the hosts.

-n *netname*
Form a network map instead a list of path aliases, and name the network *netname*. If you do not use this option, **mkhpath** assumes that your local host is within the network, and therefore inserts your local host into the network list. It also assumes that the cost of routes within the network is **LOCAL**, unless you use option **-c** to set the cost of the routes explicitly.

By default, **mkhpath** builds the route table in the same format as that generated by command **pathalias** with its option **-i**. Flag **-n** overrides this default.

If you use neither option **-g** nor **-n**, **mkhpath** constructs direct routes from your local host to each remote host.

If you use option **-g** but do not use option **-n**, **mkhpath** prefixes every route to the network (except for the route to the gateway) with the route *gateway***!**.

If you use the option **-n**, **mkhostpath** builds a pathalias map to the network this option names. The format of the map depends on whether you also use option **-g**.

If you use both options **-g** and **-n**, **mkhpath** establishes the route from your local host to *gateway*, and inserts the gateway into the network list. It does not add your local host to the network list, even if it appears as a site within the hosts table. It sets the cost of the link between your local host and the gateway is **LOCAL**; you can override this by using option **-c**. It fixes at **LOCAL** the cost of routes inside the network; this is not affected by the option **-c**.

See Also

commands, hosts, mail [overview], smail

Notes

Please note that because COHERENT does not yet support networking, this command is never used.

mkline — Command

Fold an alias file, paths file, or mailing list into one-line records
/usr/lib/mail/mkline [-ltn] [*file ...* **]**

Command **mkline** takes alias file, path file, or mailing-list file as input, and generates output records that contain one complete entry per line, and removes all comments and white space.

mkline recognizes the following command-line options:

-l Generate a list of addresses. Use this to generate a mailing list. If you use this option, **mkline** ignores options **-n** and **-t**.

-n Do not extract keys from the input. **mkline** passes all token through unchanged, although it still removes all comments and as much white space as it can without creating ambiguous output.

-t Separate the key from the data with a single tab character. The default is to use a colon ':'.

If its command line does not name an input *file*, **mkdbm** reads the standard input. **mkline** also reads the standard input if a *file* is named '-'; in this way, it can mix data read from the standard input with material read from files.

Examples

Consider the following alias file:

```
Postmaster:hustead                                      # Ted Hustead, jr.
UUCP-Postmasters: tron, chongo                          # namei contacts
yamato                                                  # kremvax contact
tron: tron@namei.uucp (Ronald S. Karr)
yamato: yamato@kremvax.ussr.comm (Yamato T. Yankelovich)
chongo: chongo@eek.uts.amdahl.com (Landon Curt Noll)
```

When it reads this file, **mkline** generates:

```
Postmaster:hustead
UUCP-Postmasters:tron,chongo yamato
tron:tron@namei.uucp
yamato:yamato@kremvax.ussr.comm
chongo:chongo@eek.uts.amdahl.com
```

As an example of using **mkline** to compress mailing lists, consider the mailing list:

```
tron@namei.uucp,tron@uts.amdahl.com                    # Ronald S. Karr
yamato@kremvax.ussr.comm                               # Yamato T. Yankelovich
chongo@eek.uts.amdahl.com                              # Landon Curt Noll
Wilt . (the Stilt) Chamberlain@NBA.US                  # RFC822 doc example
```

The command **mkline -l** generates the following:

```
tron@namei.uucp
tron@uts.amdahl.com
yamato@kremvax.ussr.comm
chongo@eek.uts.amdahl.com
Wilt.Chamberlain@NBA.US
```

See Also

commands, mail [overview], mkdbm, mksort, pathalias, smail

Notes

mkline leaves one space character if the concatenation of two tokens would otherwise cause ambiguity.

mkline frequently is used with the command **mksort**. For an example of using these commands together, see the Lexicon entry for **mksort**.

mklost+found — Command

Make an enlarged lost+found directory
/etc/mklost+found *directory* **[***slots***]**

When the command **fsck** checks your file system, it copies all "orphaned" files into directory **lost+found** in the root directory of the file system being checked. Normally, this works well; however, if a great number of files are orphaned, directory **lost+found** may not be able to hold them all. This is because a directory is itself a file that holds information about the files it contains; normally, COHERENT expands the size of a directory file when it needs more space to hold files, but because **fsck** is forbidden to modify any file, it cannot enlarge **lost+found**. Thus, orphaned files that cannot be copied into **lost+found** are deleted.

Script **mklost+found** lets you build an enlarged **lost+found** directory within *directory*. It initializes the **lost+found** directory to be able to hold *slots* files. If you do not specify how many files you want **lost+found** to be able to hold, **mklost+found** initializes it to hold 250 files.

Example

The following command creates a **lost+found** directory that can hold 1,000 files for the file system that is mounted on directory **/news**:

```
/etc/mklost+found /news 1000
```

See Also

commands, fsck

Notes

Only the superuser **root** can run this script.

mknod — Command

Make a special file or named pipe
/etc/mknod [-f] *filename type major minor*
/etc/mknod [-f] *filename* **p**

In the first form, **mknod** creates a *special file,* which provides access to a device by the *filename* specified. Special files are conventionally stored in the **/dev** directory.

type can be either 'b' (for block-special file) or 'c' (for character-special file). Block-special files tend to be devices such as disks or magnetic tape, upon which COHERENT uses an elaborate buffering strategy. Character-special

files are unstructured (character at a time) devices such as terminals, line printers, or communications devices. Character-special files may also be random-access devices; this circumvents system buffering, allowing transfers of arbitrary size directly between the user and the hardware.

The *major* device number uniquely identifies a device driver to COHERENT. The *minor* device number is a parameter interpreted by the driver; it might specify the channel of a multiplexor or the unit number of a drive.

The caller must be the superuser.

In the second form, **mknod** creates a named pipe with the given *filename.* Named pipes can be used for communication between processes.

The **-f** option to **mknod** forces the creation of a new node, even if one of the same name already exists.

Files

/dev/*

See Also

commands, mount

mknod() — System Call (libc)

Create a special file
#include <sys/ino.h>
#include <sys/stat.h>
mknod(*name, mode, addr***)**
char **name***name***; int** *mode, addr***;**

mknod() is the COHERENT system call that creates a special file. A *special file* is one through which a device is accessed, or a named pipe.

mode gives the type of special file to be created. It can be set to **IFBLK**, for a block-special device, such as a disk driver; to **IFCHR**, for a character-special device, such as a serial-port driver; to **IFDIR**, for a directory; or to **IFPIPE**, for a named pipe. *mode* also contains permission mode bits.

address is a parameter interpreted by the driver; it might specify the channel of a multiplexor or the unit number of a drive. Note that this is not used with named pipes.

If all goes well, **mknod()** returns zero. If an error occurs, it returns a negative value and sets **errno** to an appropriate value.

See Also

libc, device drivers, named pipe, pipe()

Notes

Only the superuser **root** can use **mknod()**. This is a security feature.

mkpath — Command

Create a pathalias output file
**/usr/lib/mail/mkpath [-v] [-V] [-x] [-e] [-n] **
[-t *trace***] [***path_config***]**

The script **mkpath** is a wrapper for the command **pathalias**, which generates a set of paths among computers.

File *path_config* holds the data that are passed to **pathalias**; if it is set to '-', then **pathalias** reads the standard input. If no *path_config* is named on the command line, **mkpath** by default uses file **/usr/lib/mail/maps/mkpath.conf**.

mkpath recognizes the following command-line options:

-e Tell **mkpath** to stop when it encounters a syntax error, or if a command that it invokes exits with a non-zero status.

-n Disable the execution of any commands useful with the Bourne shell's option **-v**, and disables its own options **-e**, **-t**, **-V**, and **-x**.

-t *tracefile*
Copy into *tracefile* all data passed to **pathalias**.

-V Invoke command **pathalias** with its option **-v**.

-v Verbose mode. Its commands are executed with the Bourne shell's option **-v**, which echoes each command as it is read.

-x Verbose mode. Its commands are executed with the Bourne shell's option **-x**, which echoes each command as it is executed.

See Also

commands, mail [overview], pathalias, smail

mksort — Command

Sort the standard input, allowing arbitrarily long lines
/usr/lib/mail/mksort [-f] [*file ...* **]**

The command **mksort** reads lines of text, sorts them by the first field in each line, then writes them to the standard output. It usually is used by system administrators to help prepare the data files used by **smail**. Unlike the COHERENT command **sort**, **mksort** can read and process an arbitrarily long line of text.

The first field within a line of input is delimited either by a white-space character or a colon ':'. A line can be of any length, as long as the entire input can be stored in memory. Command-lind option **-f** (for "fold") tells **mksort** to ignore case when it sorts its input; with this option, the letter 'A' equals the letter 'a', and 'a' is always less than 'B'.

If its command line does not name an input file, **mksort** reads the standard input. A file name of '-' indicates the standard input; this permits **mksort** to mingle the contents of one or more files with what it reads from the standard input.

Example

The following example demonstrates how to use **mksort** with **mkline**. Consider file **aliases**, which contains the following aliasing information:

```
Postmaster:hustead # Ted Hustead, jr.
UUCP-Postmasters: tron, chongo   # namei contacts
    yamato  # kremvax contact
tron:      tron@namei.uucp (Ronald S. Karr)
yamato:    yamato@kremvax.ussr.comm (Yamato T. Yankelovich)
chongo:    chongo@eek.uts.amdahl.com (Landon Curt Noll)
```

Given this file, the command

```
mkline aliases | mksort -f
```

yields:

```
chongo:chongo@eek.uts.amdahl.com
Postmaster:hustead
tron:tron@namei.uucp
UUCP-Postmasters:tron,chongo yamato
yamato:yamato@kremvax.ussr.comm
```

See Also

commands, mail [overview], mkline, mkdbm, pathalias, smail

Notes

This command is not used by COHERENT's implementation of **smail**.

mktemp() — General Function (libc)

Generate a temporary file name
char *mktemp(*pattern***) char ***pattern***;**

mktemp() generates a unique file name. It can be used, for example, to name intermediate data files. *pattern* must consist of a string with six **X**'s at the end. **mktemp** replaces these **X**'s with the five-digit process id of the requesting process and a letter that is changed for each subsequent call. **mktemp** returns *pattern*. For example, the call

```
char template[] = "/tmp/sortXXXXXX";
mktemp(template);
```

might return the name **/tmp/sort01234a**.

It is normal practice to write temporary files into the directory **/tmp**. The start of the file name identifies the originator of the file.

See Also

libc

Notes

Because **mktemp()** writes on the argument template, passing it a quoted string causes a segmentation violation. To avoid this, either compile the module that contains the call to **mktemp()** with the compiler option **-VPSTR** (to put the quoted string into segment **.data** rather than into segment **.text**) or, preferably, move the string into the data segment. For example, use

```
char template[] = "/tmp/sortXXXXXX";
mktemp(template);
```

rather than:

```
mktemp("/tmp/sortXXXXXX");
```

mktime() — Time Function (libc)

Turn broken-down time into calendar time
#include <time.h>
time_t mktime(*timeptr***)**
struct tm ***timeptr*;**

mktime() reads broken-down time from the structure pointed to by *timeptr* and converts it into calendar time of the type **time_t**. It does the opposite of the functions **localtime()** and **gmtime()**, which turn calendar time into broken-down time.

mktime() manipulates the structure **tm** as follows:

1. It reads the contents of the structure, but ignores the fields **tm_wday** and **tm_yday**.
2. The original values of the other fields within the **tm** structure are not restricted. This allows you, for example, to increment the member **tm_hour** to discover the calendar time one hour hence, even if that forces the value of **tm_hour** to be greater than 23, its normal limit.
3. When calculation is completed, the values of the fields within the **tm** structure are reset to within their normal limits to conform to the newly calculated calendar time. The value of **tm_mday** is not set until after the values of **tm_mon** and **tm_year**.
4. The calendar time is returned.

If the calendar time cannot be calculated, **mktime** returns -1 cast to **time_t**.

Example

This example gets the date from the user and writes it into a **tm** structure.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define BAD_TIME ((time_t)-1)

/* ask for a number and return it. */
int askint(msg)
char *msg;
{
      char buf[20];

      printf("Enter %s ", msg);
      fflush(stdout);

      if(gets(buf) == NULL)
            exit(EXIT_SUCCESS);
      return(atoi(buf));
}

main()
{
      struct tm t;

      for(;;) {
            t.tm_mon  = askint("month") - 1;
            t.tm_mday = askint("day");
            t.tm_year = askint("year") - 1900;
            t.tm_hour = t.tm_min = t.tm_sec = 1;

            if(BAD_TIME == mktime(&t)) {
                  printf("Invalid date\n");
                  continue;
            }

            printf("Day of week is %d\n", t.tm_wday);
            break;
      }
      return(EXIT_SUCCESS);
}
```

See Also

clock(), difftime(), libc, time [overview]
ANSI Standard, §7.12.2.3
POSIX Standard, §8.1

Notes

The above description may appear to be needlessly complex. However, the Committee intended that **mktime()** be used to implement a portable mechanism for determining time and for controlling time-dependent loops. This function is needed because not every environment describes time internally as a multiple of a known time unit.

MLP_COPIES — Environmental Variable

Set default number of copies to print

When the command **lp** spools a job for printing, it reads the environmental variable **MLP_COPIES** to find how many copies are to be printed. The default is one copy; the maximum is 99.

See Also

environmental variables, lp, lpadmin, lpsched, printer

MLP_FORMLEN — Environmental Variable

Set default page length

When the command **lp** spools a job for printing, it reads the environmental variable **MLP_FORMLEN** to find the length, in lines, of the form on which the job is to be printed. In the United States, a *line* is defined to be one pica high (that is, one sixth of an inch). The default is length 66 lines (11 inches). (NB, the COHERENT command **units** gives a handy way to convert from picas or inches into metric units.)

The printer daemon **lpsched** uses this information to help it count pages of input — so you can specify the range of pages that it should print. Unfortunately, **lpsched** identifies a page by counting lines of input, so this feature works only it prints "straight" text. It does *not* work correctly with "cooked" input, such as files of PostScript or PCL.

See Also

environmental variables, lp, lpadmin, printer

MLP_LIFE — Environmental Variable

Set default life for print jobs

When the command **lp** spools a job for printing, it reads the environmental variable **MLP_LIFE** to set the job's "life expectancy". **MLP_LIFE** must be one of the following:

T Temporary: live in the queue for two hours.

S Short-term: live in the queue for 48 hours.

L Long-term: live in the queue for 72 hours.

The default life expectancy is **S**.

To change the default values for life-expectancies, edit the file **/usr/spool/mlp/control**. For details, see the Lexicon article **controls**.

See Also

environmental variables, lp, lpadmin, printer

MLP_PRIORITY — Environmental Variable

Set default priority for print spooling

When the command **lp** spools a job for printing, it reads the environmental variable **MLP_PRIORITY** to find what priority it is to give the job. **MLP_PRIORITY** must a numeral, from **0** to **9**: **0** assigns highest priority, **9** lowest. The default priority is **2**.

See Also

environmental variables, lp, lpadmin, printer

MLP_SPOOL — Environmental Variable

Pass user-specific information to print spooler

When the command **lp** spools a job for printing, it reads the environmental variable **MLP_SPOOL** to find user-specific information for this job. **MLP_SPOOL** must have the following layout:

Offset	*Length*	*Description*
0	10	Type of document (user-specific)
10	3	Page length, lines per page (default, 66)
13	14	Name of data base (user-specific)
28	14	Name of program (user-specific)
42	60	Title (user-specific)

With the exception of page length, **lp** uses none of these fields itself; rather, it makes them available to whatever program the user (or system administrator) has selected to process text before it is printed.

See Also

environmental variables, lp, lpadmin, printer

mmu.h — Header File

Definitions for memory-management unit
#include <sys/mmu.h>

The header file **mmu.h** defines functions that manipulate the memory-management unit (MMU) of the Intel 80X86 family of microprocessors.

See Also

header files

mneg() — Multiple-Precision Mathematics (libmp)

Negate multiple-precision integer
#include <mprec.h>
void mneg(*a, b***)**
mint **a, *b;*

mneg() negates the value of the multiple-precision integer (or **mint**) pointed to by *a*, and writes the result into the **mint** pointed to by *b*.

See Also

libmp

mnttab — System Administration

Mount table
/etc/mnttab

File **/etc/mnttab** holds the COHERENT system's mount table. It consists of an array of type **mnttab**, which is defined in header file **mnttab.h**.

See Also

Administering COHERENT, mnttab

mnttab.h — Header File

Structure for mount table
#include <mnttab.h>

mnttab.h defines the structure for the mount table maintained by the functions **/etc/mount** and **/etc/umount**.

File **/etc/mnttab** is an array of these structures.

See Also

header files, mount, umount

modem — Technical Information

The word *modem* is an abbreviation for "modulation/demodulation device". With the COHERENT system, you can attach a modem to your computer either to dial out for remote communication, to let others dial into your COHERENT system, or both. With your modem, too, you can use COHERENT's UUCP commands to exchange mail and files with remote sites automatically, and to download news and files from networks.

This article gives a summary of how to connect your modem to your computer, describe it to the COHERENT system, and set it up for UUCP connections. It also discusses some problems that may crop up when you attempt to use your modem.

Internal vs. External Modems

You can use internal and external modems with COHERENT. You must plug an external modem into a serial port on your system, whereas you must jumper an internal modem to use one of your system's COM ports. Be sure to use a COM port that is not already used on your system, or problems will result. See the Lexicon entry for **asy** for details on how COHERENT handles COM ports.

It is more difficult to diagnose problems with an internal modem because you have no status lights to indicate operation; otherwise, they operate almost identically. The rest of this article assumes that you are working with an external modem.

Plugging in an External Modem

A modem must be hooked up to a serial port on your computer. To plug your modem into the computer, simply take a normal serial-port cable, one with an RS-232 plug of the appropriate gender at each end, plug one end into your modem and the other into the serial port you wish to use. The Lexicon article **RS-232** describes the wiring of the RS-232 plug in detail; but if you are not skilled with a soldering iron, you are well advised simply to purchase a cable from your local electronics store and be done with it.

Serial Ports

The COHERENT system supports up to four serial ports; the devices for these are named **/dev/com1r** through **/dev/com4r**. If you are not sure which port you have plugged your modem into, perform the following test: First, turn on the modem. Then, type the following command:

```
echo FOO >/dev/com1l
```

If the **TX** light on the modem blinks, then you know the modem is plugged into **com1**. If it does not, try the command again for **/dev/com2l**, and so on through **com4l** until you find the appropriate port. If no command works, check the wiring on your cable and make sure that the plugs are securely inserted.

Edit /etc/ttys

If you intend to use your modem with UUCP, you must edit file **/etc/ttys** to tell COHERENT how you want it to handle that serial port. You must know (1) whether you want the port enabled or disabled; (2) the baud rate of the port (as set by your modem); and (3) the name of the port (which you just determined).

If a port is enabled, remote users can log into the system, either via a terminal directly plugged into the port or via a modem. COHERENT sends a login prompt to every enabled port. The COHERENT system also restricts permissions on all enabled serial ports, so that only the superuser **root** can read and write to the port. This prevents other users who may be using the system from accessing the serial port. If a port is disabled, you can dial out or use a direct-connect UUCP connection via that disabled port. To dial out on an enabled port, you must first use the command **disable** to disable the port. When you have finished dialing out, run the command **enable** to re-enable the port. (Note that UUCP automatically disables and re-enables a port when it dials out to poll a remote system.) Before you can use these commands with a port, the port must first be described in the file **/etc/ttys**.

See the Lexicon article on **ttys** for details on how to edit this file. Note that a modem is a remote device, and must be so described in **/etc/ttys**, or it will not work correctly.

After you have made your changes, type the command

```
kill quit 1
```

to make COHERENT re-read **/etc/ttys** and implement your changes.

Remote-Access Passwords

If you intend to let people dial into your computer, you are well advised to set the remote-access password. This will require that people who dial in know a special password in addition to whatever password their personal account may have.

If you wish, you can set a different remote-access password for each group of users who log into your system, as organized by the program invoked upon logging in. For example, you can give one password to the users who log in and invoke **uucico**; and another to the users who log in and use the interactive shells **ksh** or **sh**. For details on how to do this, see the Lexicon entries for **d_passwd** and **dialups**.

Edit /usr/lib/uucp/dial

Once you have edited file **/etc/ttys** and have set the remote-access password, check the file **/usr/lib/uucp/dial** and see if it holds a description that matches your modem. The commands **cu** and **uucico** read the descriptions in **dial** to control how they talk to modems. **dial** already contains descriptions for many commonly used modems; but you may find that you must edit an existing entry to match your modem's features exactly; for example, the existing entry may assume that you have a Touch-Tone telephone whereas you actually have a pulse telephone. The Lexicon entry on **dial** will walk you through this process.

When you have completed editing this entry, write it down, for you will need to insert it elsewhere.

Edit Port

If you intend to use your modem with UUCP, you must insert an entry for it into your the file **/usr/lib/uucp/port**. This file links a modem, as described in file **/usr/lib/uucp/dial**, with a port on your system. This arrangement permits UUCP to use one description with several modems of the same type, each plugged into a different port.

See the Lexicon entry **port** for details.

Walking Through UUCP Configuration

The following description walks you through the task of configuring your modem to handle UUCP. It is adapted from a posting to **comp.os.coherent** by Rob Schofield (schofld@mebv.mhs.compuserve.com).

LEXICON

First, decide whether you want outsiders (including outside UUCP sites) to log into your COHERENT system. If you do, then you must add to file **/etc/ttys** the name the incoming device — that is, the device that the remote users will log into. If you do not want incoming logins, you do not need to have an incoming device installed in **/etc/ttys** and you can safely omit it.

As described above, an entry in **/etc/ttys** consists of three one-character fields, followed by the name of the device:

- The first field indicates whether the device is enabled (that is, gets a login prompt) or disabled (that is, does not get a login prompt).
- The second field indicates whether the device is in "raw" mode or whether it "cooks" its input (that is, handles backspaces correctly, and so on). You should use 'l' (for cooked input).
- The third field gives the speed of the port; see the Lexicon entry **/etc/ttys** for a list of recognized codes.
- The device has the name **/dev/com?r**. The '?' in this name stands for the number of the COM port into which you've plugged your modem, from '1' to '4'. The 'r' in the device name stands for the "remote" (i.e., modem) device. If your modem is high speed (i.e., faster than 9600 baud) then use the hardware-handshaking version of the remote device (i.e., **/dev/com?fr**).

For example, if you have plugged a 14.4-kilobaud modem into serial port 3, insert the following line into file **/etc/ttys**:

```
1lQcom3fr
```

Once you have inserted this line into **/etc/ttys**, type the command:

```
kill quit 1
```

This forces COHERENT to re-read **/etc/ttys** and so recognize your change.

If you wish to dial out on your modem via programs **cu** or **ckermit**, or if you wish to have your UUCP system dial other, remote sites, those systems must use the *local* **/dev/com?l** on the same port number as your modem. If it is high speed, again use the 'f' version **/dev/com?fl**, which enables hardware handshaking. This sounds may sound strange (after all, why use a terminal-type device on a modem?), but there's a reason for it. When you use the UNIX or COHERENT system call **open()** on a **com?r** port, the function call does not return until it detects a "true" value on DCD — and that occurs only when someone has dialed in and the modems have connected. By using a **com?r** device, you are only setting up the system for a **getty** to detect someone dialing in; if you're dialing out, you do not need to detect DCD, hence the use of a terminal device. Hence, **cu**, UUCP, and **ckermit** should all be used with the outgoing port device, and not the incoming.

Do *not* add this port to **/etc/ttys**; rather, add it to the configuration files used by the applications. In the case of **ckermit**, use its command **set speed**. You can type this command either by hand, when you invoke **ckermit**; or you can add it to file **.kermrc** in your home directory. For details, see the Lexicon entry for **ckermit**. In the case of **cu** and UUCP, the device must be named in the file **/usr/lib/uucp/port**. For example, to dial out via our 14.4-kilobaud modem plugged into COM 3, add the following entry to **/usr/lib/uucp/port**:

```
port exampleport
type modem
device /dev/com3fl
baud 19200
dialer exampledialer
```

The device is **/dev/com3fl**, not the device **/dev/com3fr** we added to **/etc/ttys**. The 'r' version of a port is used exclusively for dialing in; the 'l' version for dialing out.

Last little trick is to link the device you are using to a pseudo device used by a few communication packages:

```
ln -f /dev/com?fl /dev/modem
```

Be sure to substitute the number of the port you're using (from '1' through '4') for the '?' in the above example.

Modem Maladies

This section discusses problems that have arisen with remote login via modem, as diagnosed by the technical support staff of Mark Williams Company.

Difficulty in logging in from a remote site via modem can be the result of problems in one or more of the following: cabling; enabling/disabling the port; flaws in the contents of file **/etc/ttys**; incorrect configuration of the modem; and setting the port to an incorrect state. See Lexicon articles **terminal** and **UUCP** for additional information. The

following paragraphs discuss the above-named items in detail.

RS-232 Cabling

When attaching an external modem to your computer, it is important to use a modem cable that supports "full modem control". COHERENT relies on modem-control signals when operating a modem for remote access purposes. When attaching a terminal directly to a serial port, a "null modem" cable must be used. When attaching a modem, a "straight through" cable must be used. See Lexicon articles **RS-232** and **terminal** for further details on cabling.

Enabled vs. Disabled Ports

A serial port can be either enabled or disabled for remote access. Enabling a port allows a user on a remote terminal or modem to log into your COHERENT system. Disabling a port permits a user to dial out or use a direct connect UUCP connection via that disabled port.

If a port is enabled for remote logins and you will use it to call out, you must use the command **disable** to disable the port before you access the port. UUCP automatically disables and re-enables a port.

The port name supplied to an **enable** or **disable** command must *exactly* match the last part of a line in the **/etc/ttys** file (see below). For example, for the command **enable com2pr** to work, there must be an entry in the file **/etc/ttys** which ends with **com2pr**.

When a port is enabled, the first character for the port in file **/etc/ttys** is set to a '1' (one), the permissions for the port are changed so that only the superuser **root** can read and write to the port (to prevent other users on the system from accessing the port while a remote session is in progress), and a login prompt is sent to the port.

ttys *Problems*

This file should have permissions of 644 (-rw-r--r--) and belong to owner and group **root**. Review the Lexicon entry for **ttys** to ensure that the format of your version of **/etc/ttys** is correct.

Leaving blanks at the end of a line in **/etc/ttys** usually results in error messages stating that a device could not be found.

You do not need to edit the initial '0' or '1' in entries in **/etc/ttys**; this digit is updated by the commands **enable** and **disable**. See the Lexicon entries for **enable** and **disable** for more information.

Constant Flickering

Another problem is a constant flickering of send/receive LEDs and an unexplained continual access of the hard drive. This occurs when the port is enabled and the modem is set in echo mode: COHERENT sends the login prompt to the modem, the modem echoes it back to COHERENT, COHERENT then thinks the modem is trying to talk to it and sends the password prompt, and so on *ad infinitum*.

To fix this problem, place the modem into no-echo mode, and turn off the display of result codes. The following section discusses this in more detail.

Modem Configuration

A modem that fails to answer an incoming call, hangs up before locking onto the remote carrier, becomes stuck in a loop echoing characters sent to it from the computer, or fails to operate at the expected baud rate probably is configured improperly. To remedy this situation, send the appropriate control string to the modem.

We offer some guidelines here for modem settings. Be warned, however, that modems from different manufacturers usually behave differently, regardless of claims of Hayes compatibility: you must check the manual for your modem.

- Echo should be OFF (usually by setting "E0").
- Result codes should be OFF (usually by setting "Q1").
- Modem status "DCD" should follow true carrier detect status, rather than being always on (usually by setting "&C1").
- Auto answer should be ON (usually obtained by setting register S0 to a nonzero value equal to the number of rings before answer).
- The delay value for "Wait for Carrier/Dial Tone" (usually register S7) should not be too short.

The scripts below show typical initialization for a "Hayes-compatible" modem that runs at 2400 baud and is plugged into port **/dev/com3r**. It is only an example; your modem may need something different. Please note that

the commands **sleep** and **stty** are necessary in the first example so that the command string will be sent to the modem at 2400 baud; otherwise, the string is sent at the default port speed, which is 9600 baud.

```
# initialize 2400-baud Hayes-compatible modem
sleep 3 > /dev/com3l &
stty 2400 < /dev/com3l
echo 'AT E0 Q1 V0 S0=1 &C1 M3' > /dev/com3l
sleep 3
```

The following gives a similar script for a Trailblazer modem that runs at 9600 baud and is plugged into port **/dev/com2r**:

```
# initialize 9600 baud internal Trailblazer on com2
/etc/disable com2r
sleep 3 > /dev/com2l &
stty 9600 < /dev/com2l
echo 'AT E0 T V0 X3 H0' > /dev/com2l
echo 'AT S0=1 S7=60 S48=1 S51=252 S52=0 S54=3 S58=2' > /dev/com2l
/etc/enable com2r
```

Modem Control

This section describes the modem-control protocol used by the driver **asy**, which COHERENT uses to control serial ports. *Modem control* describes how COHERENT handles RS-232 signals other than "Receive Data" and "Transmit Data".

Many processes can open a device at the same time. *First open* occurs if a process opens a device when no process has opened the device. *Last close* occurs when a process closes the port and no other remaining process has the port open. On first open, RTS and DTR are asserted by the computer, regardless of whether the specified device used modem control. If modem control is used (the high-order bit in minor number set to zero), **open()** does not complete until CD is true. Once an **al[01]** device has been opened with modem control, loss of CD to that port causes **SIGHUP** to be sent to all processes in the group keeping the port open.

See Also

Administering COHERENT, dial, RS-232, terminal, UUCP

Notes

One final bit of hard-won wisdom: once you have something working, write down what you did, and store it in a place where you won't lose it.

modf() — General Function (libc)

Separate integral part and fraction
#include <math.h>
double modf(*real***,** *ip***)**
double *real, *ip***;**

modf() is the floating-point modulus function. It returns the fractional part of its argument *real*, and stores the integral part in the location to which *ip* points. These numbers satisfy the equation *real = f + *ip*.

Example

This example prompts for a number from the keyboard, then uses **modf()** to calculate the number's fractional portion.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

main()
{
      double real, fp, ip;
      char string[64];

      for (;;) {
            printf("Enter number: ");
            if (gets(string) == 0)
                  break;
```

```
            real = atof(string);
            fp = modf(real, &ip);
            printf("%lf is the integral part of %lf\n",
                  ip, real);
            printf("%lf is the fractional part of %lf\n",
                  fp, real);
      }
}
```

See Also

atof(), ceil(), fabs(), floor(), frexp(), ldexp(), libc
ANSI Standard, §7.5.4.6
POSIX Standard, §8.1

Notes

In releases prior to version 4.0, the COHERENT implementation of **modf()** handled negative numbers by returning a integral part less than *real*, and a positive fraction. Now, it returns an integral part greater than *real*, and a negative fraction. For example, the old version of **modf()** would transform -1.9 into an integer of -2.0 and a fraction of 0.1; whereas the current version transforms -1.9 into an integer of -1.0 and a fraction of -0.9.

The behavior of **modf()** was changed to conform to the ANSI Standard.

modulus — Definition

Modulus is the operation that returns the remainder of a division operation. For example, 12 modulus four equals zero, because when 12 is divided by four it leaves no remainder. The term "modulo" also refers to the product of a modulus operation; in the above example, the modulo is zero. In C, the modulus operation is indicated with a percent sign '%'; therefore, 12 modulus 4 is written **12%4**.

The modulus operation often is used to trim numbers to a preset range. For example, if you wanted to create a list of single-digit random numbers, you would use the command:

```
rand()%10
```

This is demonstrated by the following example.

Example

This example prints a list of 20 single-digit random numbers. The random-number table is seeded with a portion of the current system time.

```
#include <stdio.h>
#include <stdlib.h>

main()
{
      long nowhere;     /* place to put unused data */
      int counter;

      srand((int)time(&nowhere));
      for (counter = 0; counter <20; counter++)
            printf("%d\n", rand()%10);
}
```

See Also

operator, Programming COHERENT

Notes

The implementation of C defines how a modulus operator behaves when it operates upon numbers with different signs. On the i8086,

```
10 % -4
```

yields -2. This is not mathematical modulus, which is +2.

mon.h — Header File

Read profile output files
#include <mon.h>

mon.h is used with programs that read the profiling routines' output files.

See Also

header files

moo — Command

Greatly amusing numeric guessing game
/usr/games/moo [*numdigits* **]**

moo is a guessing game of numbers, typically four digits, all different.

The game randomly selects a number that consists of *numdigits* unique digits. Obviously, *numdigits* cannot exceed ten; the default is four. **moo** then prompts you to guess the number it has selected. When you type your guess, **moo** responds with one of two possible answers. If you guess the number correctly, i.e., win, **moo** responds with "Right!". If any of the digits that you guessed were correct digits, but in the wrong place, you get a "cow." If you guess a digit correctly and in the correct place, you get a "bull." If the number of "bulls" is the same as the number of digits in the guess, you win. **moo** typically responds with a count of "bulls" and "cows," as in:

```
2 bulls, 1 cow.
```

See Also

commands

Notes

moo is sometimes also called **mastermind**.

It will never replace **DOOM**.

more — Command

Display text one page at a time
more [-cdflsu] [-*window_size* **] [+***line_number* **] [+/***pattern* **] [** *file ...* **] [-]**

more is a filter for paging through text one screenful at a time. *file* is a text file; the operator **-** tells **more** to read and display the standard input.

Command-line Options

more reads options from the command line and from the environmental variable **MORE**. In case of a conflict, the options given on the command line take precedence. Every cluster of options must be preceded with a hyphen '-', even if passed via the environmental variable **MORE**.

more recognizes the following options:

-c Paint the screen from the top line down. **more** normally repaints the screen by scrolling from the bottom of the screen.

-d Prompt the user at the end of each screen with the message:

```
[Press space to continue, 'q' to quit.]
```

The default is to not issue a prompt.

-f Count actual lines from the input file rather than screen lines. This option is useful when the input contains escape sequences that **more** does not recognize.

-l Do not treat the formfeed character **<ctrl-L>** as special. By default, **more** pauses at each formfeed character, as if a full screen had been displayed.

-s Squeeze consecutive blank lines into one blank line. This is useful for looking at **nroff** output, such as manual pages.

-u Display backspaces as control characters and leave the carriage return-linefeed (CR-LF) pair alone. By default, **more** displays backspaces that appear adjacent to an underscore character as underlined text; backspaces that appear between two identical characters as emboldened text; and compresses CR-LF sequences.

+/*pattern*
Search for *pattern* before displaying a file. *pattern* is a regular expression, as recognized by commands **ed** or **egrep**. *pattern* should be escaped to avoid being processed by the shell.

-*window_size*
Set the size of the window that **more** displays to *window_size*, which is a decimal integer less than or equal to the number of lines on your terminal. The default window size is read from the **termcap** description for your terminal.

+*line_number*
Make *line_number* the beginning line to display in *file*. *line_number* is a decimal integer less than the number of lines in *file*.

Commands

The following describes **more**'s interactive commands. These commands are based on those for the UNIX editor **vi**. Some commands may optionally be preceded by a decimal number. If you enter an invalid command, **more** will beep at you.

h
? Help: display a summary of these commands.

*[N]***<space>**
Display the next *N* lines of text (default, one screenful).

*[N]***z**
If *N* is not specified, display the next screenful. Otherwise, display *N* lines and set the default scrolling size to *N* for all subsequent **<space>** and **z** commands.

*[N]***<ctrl-F>**
*[N]***f**
Scroll forward *N* screenfuls (default, one screenful). If *N* is more than the screen size, only the final screenful is displayed.

*[N]***<ctrl-B>**
*[N]***b**
Scroll backward *N* screenfuls (default, one screenful). If *N* is more than the screen size, only the final screenful is displayed.

*[N]***s**
Skip forward *N* lines (default, one line) and display one screenful.

*[N]***<return>**
*[N]***<enter>**
Scroll forward *N* lines (default, one). Display all *N* lines, even if *N* is more than the screen size.

*[N]***<ctrl-D>**
*[N]***d**
Scroll forward *N* lines (default, one half of the screen size). If *N* is specified, it becomes the new default for subsequent **d** and **<ctrl-D>** commands.

<ctrl-L>
Redraw the screen.

' (Apostrophe) Return to the position in the current file where the previous search command started, or to the beginning of the file if no search commands have occurred. This information is lost when a new file is examined.

*[N]***/***pattern*
Search forward for the *N*-th line that contains *pattern* (default, one). *pattern* is a regular expression, as recognized by **ed** or **egrep**. The search starts at the second line displayed.

LEXICON

n Repeat previous search.

:f Display the name of the current file with the current line number.

*[N]***:n**
Examine the *N*-th file after the current file, as given in the command line (default, the next file).

*[N]***:p**
Examine the *N*-th file previous to the current file, as given in the command line (default, the previous file).

! *command*
:! *command*
Pass *command* to the shell specified by environment variable **SHELL** for execution. The default shell is **/bin/sh**.

v Invoke an editor to edit the current file. The editor is set by the environment variables **VISUAL** and **EDITOR**, in that order. If these variables are not set, use **vi**.

= Display the current line number.

q
:q
Q
:Q Quit.

Environment

more uses the following environment variables:

EDITOR Specify default editor.

MORE Set default options for **more**

SHELL Specify the shell being used (normally set at login time).

TERM Specify the type of terminal you are using. **more** uses this variable to read from **/etc/termcap** the terminal characteristics needed to manipulate the screen.

VISUAL Specify default visual editor.

See Also

commands, egrep, scat, vi, zmore

Author

This software is derived from software contributed to Berkeley by Mark Nudleman. **more** is copyright © 1988,1990 by The Regents of the University of California. Copyright © 1988 by Mark Nudleman. All rights reserved.

motd — System Administration

File that holds message of the day
/etc/motd

The file **motd** holds the message of the day. Its contents are displayed by the script **/etc/profile**, which is executed whenever you log in.

Only the superuser can alter the contents of this file.

See Also

Administering COHERENT, login

mount — Command

Mount a file system
/etc/mount [*device directory* **[-ru]]**

The command **mount** mounts a file system from *device* onto *directory*. In effect, it grafts the root directory of file system on *device* onto *directory*.

If you invoke **mount** without any arguments, it displays information about the file systems that are now mounted.

If you use option **-r**, **mount** mounts the specified file system in read-only mode. This is useful if you wish to read a file system without changing it in any such way, such as when you are backing it up. Note, however, that when a file system is mounted in read-only mode, COHERENT does not update file-system information, such the time a file was last accessed.

The option **-u** tells **mount** to write an entry into the mount-table file **/etc/mtab** without actually mounting the file system. When this is done, COHERENT will hereafter mount the file system automatically whenever you boot COHERENT.

Please note that unlike every other COHERENT or UNIX command ever devised, **mount** requires that its options *follow* the file names, rather than precede them. The COHERENT version of **mount** follows this convention in order to conform to this established UNIX practice.

To un-mount a file system, use the command **umount**. (NB, this is not a typographical error — this command's name contains only one 'n'.)

The script **/bin/mount** calls **/etc/mount**, and provides convenient abbreviations for commonly used devices. For example,

```
mount f0
```

executes the command:

```
/etc/mount /dev/fha0 /f0
```

You should edit this script to reflect the devices that you use on your system.

Files

/etc/mtab — Mount table
/etc/mnttab — Mount table
/bin/mount — Shell script that calls **/etc/mount**

See Also

commands, fsck, mkfs, mknod, umount

Diagnostics

Errors can occur if *device* or *directory* does not exist or if you do not have permission to access *device*.

The message

```
/etc/mtab older than /etc/boottime
```

indicates that **/etc/mtab** has probably been invalidated by booting the system.

Attempting to mount a block-special file that does not contain a COHERENT file system (e.g., a tape device) can have disastrous consequences. *Caveat utilitor!* To build a file system on a block-special device, use the command **/etc/mkfs**. For details, see its entry in the Lexicon.

mount.h — Header File

Define the mount table
#include <sys/mount.h>

mount.h defines the structures and constants that comprise the COHERENT system's mount table. It also declares functions that are used internally by routines that manipulate the mount table.

See Also

header files, mount

mount() — System Call (libc)

Mount a file system
#include <sys/mount.h>
#include <sys/filsys.h>
int mount (*device***,** *name***,** *flag***)**
char **device*, **name*; **int** *flag*;

mount() is the COHERENT system call that mounts a file system. *device* names the physical device that through which the file system is accessed. *name* names the root directory of the newly mounted file system. *flag* controls the manner in which the file system is mounted, as set in header file **sys/mount.h**.

See Also

fd, libc, mount, mount.h

mount.all — System Administration

Mount file systems at boot time
/etc/mount.all

The file **/etc/mount.all** holds a set of **mount** commands to mount all COHERENT file systems on hard disk. It is invoked by the script **/etc/rc**, which COHERENT reads and executes at boot-time.

When you add a new COHERENT partition to your system, you should insert an appropriate entry into this file, to ensure that the new partition is mounted whenever you reboot your system. You should also insert an entry into **/etc/checklist**, to ensure that the utility **fsck** examines and corrects the file system on this new partition before the system mounts it.

See Also

Administering COHERENT, checklist, mount, rc

mout() — Multiple-Precision Mathematics (libmp)

Write multiple-precision integer to stdout
#include <mprec.h>
void mout(*a***)**
mint **a***a***;**

mout() writes the multiple-precision integer (or **mint**) pointed to by *a* onto the standard output. The base of the output is set by the value of the external variable **obase**.

See Also

libmp

mprec.h — Header File

Multiple-precision arithmetic
#include <mprec.h>

The header file **mprec.h** declares a set of routines used to perform multiple-precision arithmetic. It also declares the structure **mint**, which holds multiple-precision integers.

See Also

header files, libmp

mrand48() — Random-Number Function (libc)

Return a 48-bit pseudo-random number as a long integer
long mrand48();

Function **mrand48()** generates a 48-bit random number, then returns its high-order 32 bits in the form of a **long**. The value returned is (or should be) uniformly distributed throughout the range of -2^31 through 2^31.

See Also

libc, srand48()

ms — Technical Information

Manuscript macro package
nroff -ms *file ...*

The **nroff** macro package **ms** formats manuscripts. The tutorial on **nroff** describes the **ms** macros in detail.

ms includes the following macros:

.AB Begin the abstract portion of a document's title page.

.AE End the abstract

.AI Indicate author's institution on a document's title page.

.AU Name the author on the title page of a document.

.B Boldface font: set the following argument in boldface. If the argument is longer than one word, it must be enclosed in quotation marks. Anything on the line after the argument is thrown away.

.BD Block-centered display. Take a portion of text; do not adjust it or break it between two lines, but center it as a whole.

.BT Bottom title. This controls the printing of the footer title, should you want one. It uses three strings, all or any of which can be defined by the user: **LF**, for left-hand portion; **CF**, for center portion; and **RF**, for right-hand portion. **CF** has the default definition of printing the page number; the other two strings are undefined.

.CD Centered display. Center individually every line within a display.

.DA Set the date.

.DE Mark the end of a display. Do *not* use after the macros **.LD**, **.CD**, or **.RD**.

.DS Mark the beginning of a display. Do *not* use for displays longer than one page.

.FE Mark the end of a footnote entry.

.FS Mark the beginning of a footnote entry.

.I Italic font. Used like **.B**, above.

.ID Indent a display 1/2 inch before printing.

.IP Indent a paragraph of text before printing. This macro can take two arguments: argument 1 is used as a **tag** that is printed to the left of the first line of the paragraph; argument 2 indicates how far to indent the paragraph, in characters (the default is five characters, or 1/2 inch).

.KE Indicate the end of a *keep*, or a portion of text that must not be broken between two pages.

.KF Start floating keep.

.KS Indicate the beginning of a *keep*.

.LD Set a display flush left; used with displays that are longer than one page.

.NH Set a numbered heading. This macro takes one argument: the *depth* of numbering. For example, a '4' here would yield a number of the format "1.1.1.1". No number higher than five is accepted here. The following line gives the text of the heading.

.PP Begin a new paragraph.

.QE Mark the end of a quoted paragraph.

.QP Quoted paragraph. Used like **.IP**, above.

.QS Mark the beginning of quoted text; text is indented by five characters (1/2 inch).

.R Roman font. Used like **.B**, above.

.RE Mark the end of a relative indentation.

.RS Mark the beginning of a relative indentation. A relative indentation is a block of text that is indented five characters (1/2 inch) more than the text before it.

.SH Subheading. One line of space is inserted, and the following line of text is set boldface and flush left.

.TA Set tabs, in characters.

.TL Title: format the title entry on the cover page of a document.

Files

/usr/lib/tmac.s

See Also

man, nroff, troff, Using COHERENT *Introduction to nroff, Text Processing Language*, tutorial

MS-DOS — Technical Information

That other operating system

MS-DOS is the native operating system of the IBM-AT and compatible computers. As such, it needs no introduction to most users. Many customers have asked, however, how MS-DOS and COHERENT compare in terms of their capabilities; and many have also asked for a chart that maps familiar MS-DOS commands to their COHERENT equivalents. This article attempts to fulfill these requests.

MS-DOS vs. COHERENT

MS-DOS differs significantly from COHERENT in practically every aspect of its design. For example, its file system is incompatible with COHERENT; its shell **command.com** differs significantly from COHERENT's suite of shells; the manner in which it loads and executes a program differs completely from COHERENT's.

The most noticeable difference in design, however, is that MS-DOS is a single-user, single-process operating system, whereas COHERENT is a multi-user, multi-tasking operating system.

Single-user means that only one user can use MS-DOS at any given time: whoever sits at the keyboard "owns" the machine and all its facilities. *Multi-user* means, of course, that more than one user can use COHERENT at any given time, via terminals or modems plugged into the computer's serial ports. The number of users who can use your COHERENT system at once is limited only by your computer's speed, available memory, and by the number of serial ports that can be plugged into your computer.

Single-tasking means that MS-DOS can do only one task at a time: it loads a program into memory, runs it to completion, then awaits your request to execute another program. *Multi-tasking* means that COHERENT can execute more than one program at a time.

To grasp how multi-tasking can simplify some work, consider the task of formatting floppy disks. Under MS-DOS, you pop the floppy disk into the drive, invoke the MS-DOS program **format**, answer its queries, then go get a cup of coffee while the machine grinds away. Formatting a box of high-density floppy disks ties up your machine for the better part of an hour, which is largely wasted time for you. Under COHERENT, however, you can format a floppy disk in the *background* — that is, you can tell COHERENT to execute the disk-format program unsupervised, and let you work with another program. For example, if you wish to low-level format a 5.25-inch, high-density floppy disk in drive 0 (that is, drive A), use the following command:

```
/etc/fdformat -v /dev/fha0 &
```

Try it. You'll notice that the COHERENT prompt returns immediately: while COHERENT is formatting your disk for you, you can edit a file, play a video game, dial out to a remote system, or even format a second disk in your machine's B drive (should you have one).

Multi-tasking also means that you can program COHERENT to execute programs untended, even while you are away from your machine. The UUCP system is a good example of this feature. UUCP lets you exchange mail and files with remote systems via modem; once the system is set up, it runs automatically, without requiring that you sit at the keyboard to run it.

This discussion only gives you a taste of the advantages COHERENT enjoys over an obsolete system like MS-DOS. The following documents contain information that MS-DOS users will find helpful:

- The tutorial *Using the COHERENT System* introduces COHERENT to new users. If you are new to COHERENT and have not yet read this tutorial, you should do so before you continue any farther.
- The Lexicon articles **floppy disks** and **hard disk** discuss the in's and out's of using mass-storage device with COHERENT. The article **floppy disks** in particular discusses in detail all the steps required to format and manipulate MS-DOS-style floppy disks under COHERENT.
- The Lexicon articles **modem**, **printer**, and **terminal** discussion how to connect these devices to COHERENT, and introduce the set of commands with which you can manipulate them under COHERENT.

- The Lexicon article **execution** describes in detail how COHERENT loads and executes a program. This article is aimed at the technically knowledgeable, but neophytes may find parts of it helpful.
- The Lexicon article **commands** summarizes all commands available under the COHERENT system. This article will help you grasp the scope of COHERENT's suite of commands, and will help you explore them systematically.
- The following Lexicon articles describe COHERENT commands for manipulating MS-DOS files and disks:

doscp Copy files to/from an MS-DOS file system.

doscat Concatenate a file on an MS-DOS file system.

doscp Copy a file to/from an MS-DOS file system.

doscpdir Copy directories to/from an MS-DOS file system.

dosdel Delete files from an MS-DOS file system.

dosdir Show the contents of an MS-DOS directory.

dosformat Write an MS-DOS file system onto a floppy disk.

doslabel Label an MS-DOS floppy disk. The MS-DOS file system can reside on a floppy disk or an MS-DOS portion of a hard disk.

dosls List contents of an MS-DOS file system.

dosmkdir Create a directory on an MS-DOS file system.

dosrm Remove a file on an MS-DOS file system.

dosrmdir Remove a directory from an MS-DOS file system.

COHERENT Equivalents to MS-DOS Commands

The following table lists the most commonly used MS-DOS commands, and gives COHERENT equivalents.

Note that often there is no single COHERENT command that equates to a given MS-DOS command. COHERENT often offers several alternatives, and you can select the one that best suits your needs. Every COHERENT command has its own article in the COHERENT Lexicon; look there first for details on how to use the command.

BACKUP

This command copies a directory's files to a formatted floppy disk to back them up. To do so under COHERENT, use the command:

```
find . -print | cpio -ocm > /dev/rfha0
```

Note that **cpio** requires a formatted, defect free floppy disk, however you do not need to create a filesystem on the floppy disk prior to using **cpio**.

Note that if you want COHERENT to prompt you before it backs up a file, use the command:

```
find . -print | cpio -ocmr > /dev/rfha0
```

See the article on the archiving command **cpio** for details on this command — especially important if you expect to retrieve your backed-up files.

Note, too, that the device **/dev/rfha0** corresponds to a 5.25-inch, high-density floppy disk in drive 0 (drive A). See the article **floppy disks** for a list of the devices that correspond to different sizes and configuration of floppy disks.

BREAK

Abort a command. Aborting a command under COHERENT varies, depending upon whether the command is running in the foreground or the background. The keystroke

```
<ctrl-c>
```

aborts most commands that are running in the foreground. To abort a command that is running in the background, you must use the **kill** command. See its Lexicon entry for details on how to use it.

CHDIR or **CD**
Change to another directory. To do so under COHERENT, use the command

```
cd dir
```

where *dir* is the directory to which you wish to go. The directories '.' and '..' are used by both COHERENT and MS-DOS; since MS-DOS "borrowed" its directory structure from UNIX (of which COHERENT is an implementation), the similarity should not be surprising.

Note that MS-DOS requires that before you can change to directory on another physical device or partition, you must first switch to that device by typing its name before you use the **chdir** command. COHERENT has no such restriction.

CHKDSK
Check the integrity of a file system. Under COHERENT, use the command:

```
/etc/fsck [option] [filesystem]
```

Read the Lexicon entry on **fsck** *before you attempt to run it!*

COMP Compare the contents of two files. To do so under COHERENT, use the following command to compare two binary files:

```
cmp [option] file1 file2
```

cmp displays the bytes which differ between the files.

To compare the contents of two text files, use the command:

```
diff [option] file1 file2
```

COPY Copy the contents of one file into another; create the target file if it does not already exist. Under COHERENT, say:

```
cp oldfilename newfilename
```

To copy a set of files into a directory without changing their names, use the following form of the command:

```
cp file1 ... fileN directory
```

DATE Reset the current date and time. Under COHERENT, use the command:

```
date yymmddhhmm.ss
```

Only the superuser can reset the system's date and time. When **date** is used without an argument, it prints the date and time on the standard output.

DIR Type the contents of a directory. Under COHERENT, use the command:

```
ls -l
```

DIR/W List a directory's contents in columnar form. Under COHERENT, use either the command:

```
lc
```

or the command:

```
ls -C
```

DISKCOPY
Copy one floppy disk track-by-track to another floppy disk. COHERENT has no exact equivalent to this command; however, you can copy the contents of one disk to another by using the following set of commands.

First, place a write-protect tab on your source disk; insert the disk into drive 0 (drive A), then type the following command:

```
dd if=/dev/fha0 of=/tmp/filename
```

This copies the contents of the 5.25-inch, high-density floppy disk in drive 0 into file **/tmp/filename**. For a table of devices that correspond to other sizes and configurations of floppy disks, see the Lexicon article **floppy disks**.

Second, insert formatted destination diskette into drive 0, and then type the command:

```
dd if=/tmp/filename of=/dev/fha0
```

This command copies the files in directory **/tmp/filename** onto the target floppy disk. Note that the target disk must be formatted before it can receive files; see the Lexicon article **floppy disks** for information on how to do this.

EDLIN Perform simple-minded editing of text files. Under COHERENT, the **ed** editor performs line editing, but is much more sophisticated than **edlin**. COHERENT also includes the **vi** and MicroEMACS screen editors, which are more useful still.

ERASE or **DEL**
Remove a file or a directory. To erase a file, use the command:

```
rm file1 [... fileN ]
```

To erase a directory, use the command:

```
rmdir directory
```

To erase a directory and all files and directories below it, use the command:

```
rm -r directory
```

FIND Find a pattern within a text file. Under COHERENT, use the command:

```
egrep [option] pattern [file ...]
```

egrep is an extremely useful command; see its Lexicon entry for details on how to use it.

FORMAT
Format a floppy disk. To format a floppy disk for MS-DOS, use the command **dosformat**. To format a floppy disk for COHERENT, use the command **fdformat**. For details, see the respective Lexicon entries for these commands. Under COHERENT, use the command

MEM Find how much space is left free on your hard disk. Under COHERENT, say:

```
df [options]
```

See the Lexicon entry on **df** for details.

MKDIR Create a new directory. Under COHERENT:

```
mkdir directory ...
```

MODE Set parameters for terminals and ports. Under COHERENT, use the command **stty**. This command comes with many options; see its Lexicon entry for details. The default speeds of all ports and terminals reside in file **/etc/ttys**. The superuser can use a text editor to edit this file to change any or all default settings.

MORE Display text a screenful at a time. Under COHERENT, use the commands **more** or **scat**.

PRINT Print files via a serial port. To print a file on a dot-matrix printer, use the command:

```
lpr file1 [ ... fileN ]
```

To print a file on a Hewlett-Packard LaserJet printer, use the command

```
hpr file1 [ ... fileN ]
```

Note that before these commands can be used, the appropriate devices must be linked to your system. See the Lexicon article on **printer** for details.

Note, too, that COHERENT uses a spooling system to manage the printing of files; thus, attempting to print a non-existent file will not hang the system.

PROMPT
Change the **command.com** prompt. The COHERENT shells store the prompt format within the environmental variable **PS1**. This variable is usually defined in each user's **.profile** file; this file holds commands that are executed whenever the user logs in. To change the definition of your prompt, edit **.profile** to define **PS1** to suit your preference, then log in again.

Note that the information that can be embedded within the prompt varies between the Bourne and Korn shells. See the Lexicon articles **sh** and **ksh** for details on those shells and their prompts.

RENAME
Rename a file. Under COHERENT, use the command:

```
mv oldfile newfile
```

mv can also be used to move files from one directory or file system to another.

RESTORE
Restore a file saved with the **BACKUP** command. Under COHERENT, insert the floppy disk upon which the **cpio** utility saved its backup archive; then type the command:

```
cpio -icv < /dev/rfha0
```

Note that this command assumes you are using **/dev/rfha0**, which describes a 5.25-inch, high-density floppy disk in drive 0 (drive A). For a table of devices that correspond to other sizes and configurations of floppy disks, see the Lexicon article **floppy disks**.

TREE List all directories on a file system. Under COHERENT, use the command:

```
find / -type d | more
```

To list all files and directories that are subordinate to the current directory, use the command:

```
find . | more
```

The COHERENT command **ls -lR** also lists a directory tree, in a somewhat different output format.

MS-DOS 6.0 and COHERENT

Release 6.0 of MS-DOS offers a feature of dynamic file compression that creates some difficulties for machines that have both COHERENT and MS-DOS on their systems.

To begin, MS-DOS 6.0 assumes that it is the only operating system on your computer. When you install MS-DOS 6.0, by default it overwrites the COHERENT master boot block. If at all possible, you should install MS-DOS 6.0 onto your system first, then install COHERENT so that its Master Bootstrap is in control of your machine.

Second, MS-DOS 6.0 offers a compression utility called **dblspace**, which compresses MS-DOS file systems on the fly. The COHERENT **dos** commands do not understand compressed MS-DOS file systems created by the MS-DOS 6.0 utility **dblspace** or by such programs as **Stacker**. If you are running MS-DOS 6.0 with file compression, you must copy files to an uncompressed file system (for example, to an uncompressed floppy disk or to the uncompressed host for a compressed file system) to make them accessible to the COHERENT **dos** commands.

See Also

COHERENT, doscat, doscp, doscpdir, dosdel, dosdir, dosformat, doslabel, dosls, dosmkdir, dosrmdir, floppy disks, hard disk, modem, printer, terminal, Using COHERENT

msg — Kernel Module

Kernel module for messages

The kernel module **msg** enables System V-style messages. It is called a *kernel module* because you can link it into your kernel or exclude it, as you wish, just like a device driver; yet it is not a true device devicer because it does not perform I/O with a peripheral device.

See Also

device drivers, kernel, msgctl()

msg — Command

Send a brief message to other users
msg *user*
message

The command **msg** prints the one-line *message* on the screen of *user*.

The message is sent as soon as you type **<return>** on the *message* line. If *user* is not logged in or is not known to the system, **msg** prints an error message on your screen.

See Also

commands

msg.h — Header File

Definitions for message facility
#include <sys/msg.h>

msg.h defines the structures and constants used with the COHERENT message facility.

See Also

header files, msgget()

msgctl() — General Function (libc)

Message control operations
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(*id, command, buffer***)**
int *id***; int** *command***; struct msqid_ds** **buffer***;*

The function **msgctl()** controls the COHERENT's system's messaging facility. This facility permits processes to pass messages from one another.

Each message queue is controlled by a structure of type **msqid_ds**, which is defined in header file **<sys/msg.h>**. This structure points to the first and last messages in the queue, gives the size of the queue and the number of messages in the queue, and names who can manipulate it and how. The messages themselves consist of a linked list of structures of type **msg**, which is also defined in **msg.h**. When the function **msgget()** creates a message queue, it assigns to that queue an identification number and returns that number to the calling process. For details on this process, see the Lexicon entry for **msgget()**.

id identifies the message queue to be manipulated. This value must have been returned by a call to **msgget()**.

command names the operation that you want **msgctl()** to perform. **msgctl()** recognizes the following *command*s:

IPC_STAT Copy the message-queue structure identified by *id* into the structure pointed to by *buffer*. This command lets you gather information about a message queue without actually manipulating the queue.

IPC_SET This command sets permissions for this queue. It does so by copying fields **msg_perm.uid**, **msg_perm.gid**, **msg_perm.mode** (low nine bits only), and **msg_qbytes** from the message-queue structure point to by *buffer* to structure identified by *id*. Only the superuser **root** and the user who owns the process that created structure *id* can execute this *command*. Note that only the superuser can raise the value of field **msg_qbytes**, which gives the size of space occupied by the queue, in bytes.

IPC_RMID Remove the structure identified by *id*, and destroy its queue. Only the superuser **root** and the user who owns the process that created structure *id* do this.

If any of the following conditions occur, **msgctl()** returns -1 and sets **error** to the value in parentheses:

- *id* is not a valid message-queue identifier (**EINVAL**).
- *command* is not a valid command (**EINVAL**).
- *command* equals **IPC_STAT**, but the owner of the calling process lacks permission to execute this command (**EACCES**).
- *command* equals **IPC_RMID** or **IPC_SET**, but the owner of the calling process lacks permission to execute the command (**EPERM**).
- A process owned by someone other than the superuser **root** attempted to increase field **msg_qbytes** (**EPERM**).
- *buffer* points to an illegal address (**EFAULT**).

If all went well, **msgctl()** returns zero.

Example

For an example of this function, see the Lexicon entry for **msgget()**.

Files

/usr/include/sys/ipc.h
/usr/include/sys/msg.h

See Also

libc, msgget(), msgrcv(), msgsnd()

Notes

For information on other methods of interprocess communication, see the Lexicon entries for **semctl()**, **shmctl()**, and **libsocket**.

msgget() — General Function (libc)

Create or get a message queue
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
msgget(*key***,** *flag***)**
key_t *key*; **int** *flag;*

The function **msgget()** gets or creates a message queue. If necessary, it can create a message queue and its control structure, and link them to the identifier *key*.

key is an identifier that your application generates to identify its message queues. To guarantee that each key is unique, you should use the function call **ftok()** to generate keys.

When it creates a message queue, **msgget()** also creates a copy of structure **msqid_ds**, which the header file **<sys/msg.h>** defines as follows:

```
struct msqid_ds {
        struct ipc_perm msg_perm;         /* operation permission struct */
        struct msg *msg_first;            /* ptr to first message on queue */
        struct msg *msg_last;             /* ptr to last message on queue */
        unsigned short msg_cbytes;        /* current # bytes on queue */
        unsigned short msg_qnum;          /* # of messages on queue */
        unsigned short msg_qbytes;        /* max # of bytes on queue */
        unsigned short msg_lspid;         /* pid of last msgsnd() */
        unsigned short msg_lrpid;         /* pid of last msgrcv() */
        time_t msg_stime;                 /* last msgsnd() time */
        time_t msg_rtime;                 /* last msgrcv() time */
        time_t msg_ctime;                 /* last change() time */
};
```

The messages themselves consist of a linked list of structures of type **msg**. Fields **msg_first** and **msg_last** point to, respectively, the first and last messages in the list. Header file **<sys/msg.h>** defines structure **msg** as follows:

```
struct msg {
        struct msg *msg_next;             /* pointer to next message on queue */
        long msg_type;                    /* message type */
        short msg_ts;                     /* message text size */
        short msg_spot;                   /* message text map address */
};
```

Field **msg_perm** is a structure of type **ipc_perm**, which header file **<sys/ipc.h>** defines as follows:

```
struct ipc_perm {
        unsigned short uid;               /* owner's user id */
        unsigned short gid;               /* owner's group id */
        unsigned short cuid;              /* creator's user id */
        unsigned short cgid;              /* creator's group id */
        unsigned short mode;              /* access modes */
        unsigned short seq;               /* slot usage sequence number */
        key_t key;                        /* key */
};
```

msgget() initializes **msqid_ds** as follows:

- It sets the fields **msg_perm.cuid**, **msg_perm.uid**, **msg_perm.cgid**, and **msg_perm.gid** to, respectively, the effective user ID and effective group ID of the calling process.
- It sets the low-order nine bits of **msg_perm.mode** to the low-order nine bits of *flag*. These nine bits define access permissions: the top three bits give the owner's access permissions (read, write, execute), the middle three bits the owning group's access permissions, and the low three bits access permissions for others.
- It sets **msg_ctime** is set to the current time.
- It sets **msg_qbytes** to the value of kernel variable **NMSQB**, which sets the maximum number of bytes available to the message queue.

If any of the following error conditions is true, **msgget()** returns -1 and sets **errno** to the value within parentheses:

- *key* already has a message queue, but the owner of the process that called **msgget()** does not have permission to read it (**EACCES**).
- *key* does not have a message queue associated with it, but *flag* is does not request that one be created (i.e., *flag* **& IPC_CREAT** is false) (**ENOENT**).
- *flag* requests that **msgget()** create a message queue, but the system's maximum number of message queues (as set by the kernel variable **NMSQID**) already exists (**ENOSPC**).
- *key* already has a message queue, but *flag* requests that a queue be created exclusively (i.e., (*flag* **& IPC_CREAT) && (***flag* **& IPC_EXCL)** is true) (**EEXIST**).

If all goes well **msgget()** returns the message-queue identifier, which is always a non-negative integer. Otherwise, it returns -1 and sets **errno** to an appropriate value.

Example

The following program, **samplemsg.c**, gives an example of the COHERENT message facility. One server process accepts user keyboard input, and sends it client 1 if the first character is an upper-case letter, or to client 2 if the first character is not an upper-case letter.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/signal.h>
#include <sys/types.h>
#include <sys/wait.h>

/* Maximum size of messages in this example.
 * The default maximum size is 2048. */
#define     MAX_MSG_SIZE 80

/* template for a message */
struct my_msg {
      long mtype;
      unsigned char mtext[MAX_MSG_SIZE];
};

struct my_msg sndmsg; /* message we will send */
struct my_msg rcvmsg; /* message we will receive */

key_t key;            /* key for getting our message queue */
int id;               /* message queue id returned by msgget() */
long msgtype;         /* type of the message */

main()
{
      /* Generate unigue key */
      if ((key = ftok("./samplemsg", 'A')) == -1)
            fprint (stderr, "samplemsg does not exist.\n");
            exit(EXIT_FAILURE);
      }
```

LEXICON

```
      /* get our message queue, abort on error */
      if( -1 == (id = msgget(key, IPC_CREAT|0660))){
            printf("Error obtaining message queue\n");
            exit(EXIT_FAILURE);
      }

      printf("To end this demonstration, type 'end'.\n"
            "Enter the message -> ");
      fflush(stdout);
      msgtype = 1;      /* 1st client receives messages of type 1 */

      /* fork() to produce our 1st client processes. */
      if (fork()) { /* we are parent process (server) */
            msgtype = 2; /* 2nd client receives messages of type 2 */
            /* fork() again to produce our 2nd client processes. */
            if (fork()) { /* we are parent process (server) */
                  send_messages(); /* server */
            } else
                  receive_messages(); /* second client */
      } else
            receive_messages(); /* 1st client */
      exit (EXIT_SUCCESS);
}

/* Get a message from user and send it to client or child processes */
send_messages()
{
      for (;;) {
            /* get our message to send */
            gets(sndmsg.mtext);

            /* if 'end' was entered, send message to BOTH clients,
             * as this is a flag for them to terminate themselves.
             * Otherwise, just send the message.
             */
            if (!strcmp(sndmsg.mtext,"end")) {
                  sndmsg.mtype = 1;
                  msgsnd(id, &sndmsg, strlen(sndmsg.mtext)+1, 0);
                  sndmsg.mtype = 2;
                  msgsnd(id, &sndmsg, strlen(sndmsg.mtext)+1, 0);
                  printf("Thank you. Bye.\n");
                  break;
            }

            /* Determine the type of message this will be.
             * if the first character is upper case letter,
             * then this is a type-1 message; otherwise,
             * this is a type-2 message.
             */
            if (isupper(sndmsg.mtext[0]))
                  sndmsg.mtype = 1L;
            else
                  sndmsg.mtype = 2L;

            if (msgsnd(id, &sndmsg, strlen(sndmsg.mtext)+1, 0) < 0) {
                  perror("send");
                  break;
            }
      }

      while (wait(NULL) > 0)  /* Wait for the children */
            ;
      msgctl(id, IPC_RMID,0); /* remove message queue */
      return;
}
```

```
/* receive_messages(). */
receive_messages()
{
      char clntbuf[20];

      sprintf(clntbuf, "Client %ld", msgtype);

      for (;;) {
            if (msgrcv(id, &rcvmsg, MAX_MSG_SIZE, msgtype, 0) < 0) {
                  perror(clntbuf);
                  exit(EXIT_FAILURE);
            }

            printf("%s received: '%s'\n", clntbuf, rcvmsg.mtext);
            if (!strcmp(rcvmsg.mtext,"end"))
                  break;
            printf("Enter next message -> ");
            fflush(stdout);
      }
      exit(EXIT_SUCCESS);
}
```

Files

/usr/include/sys/ipc.h
/usr/include/sys/msg.h

See Also

ftok(), ipcrm, ipcs, libc, libsocket, msgctl(), msgrcv(), msgsnd()

Notes

Prior to release 4.2, COHERENT implemented semaphores through the driver **msg**. In release 4.2, and subsequent releases, COHERENT has implemented semaphores as a set of functions that conform in large part to the UNIX System-V standard.

msgrcv() — General Function (libc)

Receive a message
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
msgrcv(*id***,** *buffer***,** *size***,** *type***,** *flag***)**
int *id***,** *size***,** *flag***; long** **buffer***; **long** *type***;**

The function **msgrcv()** reads a message from the queue associated with identifier *id*, and writes it into the user-defined chunk of memory to which *buffer* points. The memory to which *buffer* points has a layout similar to a structure with the following members (if we pretend **mtext[]** is legal C):

```
struct msgbuf {
       long mtype;                    /* message type */
       char mtext[];                  /* message text */
};
```

mtype gives the message's type, as specified by the sending process. **mtext** gives the text of the message.

size gives the size of the message's text, in bytes. **msgrcv()** silently truncates the received message to *size* if it more than *size* bytes long and (*flag* **& MSG_NOERROR**) is true.

type gives the type of message being requested. **msgrcv** obeys the following rules when it reads the message queue:

- If *type* equals **0L**, it reads the first message in the queue.
- If *type* is greater than **0L**, it reads the first message of *type*.
- If *type* is less than **0L**, it reads the first message whose type is less than or equal to the absolute value of *type*.

If the message queue contains no message of the desired type, the behavior of **msgrcv()** is determined by the value of *flag*. If *flag* contains the value **IPC_NOWAIT** (i.e., *flag* **& IPC_NOWAIT** is true) then **msgrcv()** sets **errno** to **ENOMSG** and returns -1. If, however, *flag* does not contain **IPC_NOWAIT**, then **msgrcv()** suspends execution until

one of the following occurs:

1. A message of the desired type appears on the queue.
2. *id* is removed from the system. **msgrcv()** sets **errno** to **EIDRM** and returns -1.
3. The calling process receives a signal. **msgrcv()** sets **errno** to **EINTR** and returns -1. The calling process then resumes execution in the manner by signal received. For information on what given signals mean, see the Lexicon entry for **signal()**.

msgrcv() also fails and returns no message if any of the following is true:

- *id* is not a valid message-queue identifier. **msgrcv** sets **errno** to **EINVAL**.
- The calling process lacks operation permission (**EACCES**).
- *size* is less than zero (**EINVAL**).
- The message's size is greater than *size* bytes long and (*flag* **& MSG_NOERROR**) is false (**E2BIG**).
- *buffer* points to an illegal address (**EFAULT**).

When **msgrcv()** has successfully received its message, it modifies the data structure associated with *id* in the following ways:

- It decrements field **msg_qnum** by one.
- It sets **msg_lrpid** to the identifier of the process that called **msgrcv()**.
- It sets **msg_rtime** to the current time.

When it completes successfully, **msgrcv()** returns the number of bytes written into the field **mtext** of the structure pointed to by *buffer*.

Example

For an example of this function, see the Lexicon entry for **msgget()**.

Files

/usr/include/sys/ipc.h
/usr/include/sys/msg.h

See Also

libc, msgctl(), msgget(), msgsnd()

msgs — Command

Read messages intended for all COHERENT users
msgs [-*q*] **[***number***]**

msgs selects and displays messages that are intended to be read by all COHERENT users. Messages are mailed to the login **msgs**. They should contain information meant to be read once by most users of the system.

The command **msgs** normally is in a user's **.profile**, so that it is executed every time he logs in. When invoked, it prompts the user with the identifier of the user who sent the message and the message's size. **msgs** then asks the user if he wishes to see the rest of the message. The user should reply with one of the following:

y	Display the message.
<return>	Display the message.
n	Skip this message and go to the next one.
-	Redisplay the last message.
q	Quit **msgs**.
number	Display message *number*; then continue.

If environmental variable **PAGER** is defined, **msgs** will "pipe" each message through the command specified in **PAGER**. For example, the **.profile** command line:

```
export PAGER="exec /bin/scat -1"
```

would invoke **/bin/scat** for each message with the command line argument **-1** (the digit one).

msgs writes into the file **$(HOME)/.msgsrc** the number of the next message the user will see when he invokes

msgs. **msgs** keeps all messages in the directory **/usr/msgs**; each message is named with a sequential number, which indicates its message number. The file **/usr/msgs/bounds** contains the low and high numbers of the messages in the directory; **msgs** determines whether a user has not read a message by comparing the information in **$(HOME)/.msgsrc** with that in **/usr/msgs/bounds**. If the contents of **/usr/msgs/bounds** are incorrect, the problem can be fixed by removing that file; **msgs** will create a new **bounds** file the next time it is run.

When the contents of a message are no longer needed, simply remove that message. Avoid removing the **bounds** file and the highest numbered message at the same time.

msgs accepts the following command-line options:

-q Query whether there are messages; print "There are new messages" if there are, and "No new messages" if not. The command **msgs -q** is often used in profile scripts.

number Start at message *number* rather than at the message recorded in **$(HOME)/.msgsrc**. If *number* is greater than zero, then start with that message; if *number* is less than zero, then begin *number* messages before the one recorded in **$(HOME)/.msgsrc**.

Files

/usr/spool/mail/msgs — Mail messages file
/usr/msgs/[1-9]* — Data base
/usr/msgs/bounds — File that contains message number bounds
$(HOME)/.msgsrc — Number of next message to be presented

See Also

commands, mail, PAGER, scat

msgsnd() — General Function (libc)

Send a message
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
msgsnd(*id***,** *buffer***,** *size***,** *flag***)**
int *id***,** *size***,** *flag***; long** **buffer***;*

The function **msgsnd()** inserts a message into the queue associated with identifier *id*.

buffer points to a user-defined buffer that holds a code that defines the type of the message, and the text of the message. *buffer* can be described by a structure something like the following (if we pretend **mtext[]** is legal C):

```
struct msgbuf {
        long mtype;                     /* message type */
        char mtext[];                   /* message text */
};
```

Field **mtype** is a positive long integer that gives the type of message this is. Function **msgrcv()** examines this field to see if this message is of the type that it seeks. The text of the message immediately follows **mtype** in memory, for *size* bytes. *size* can range from zero to a maximum defined in the kernel variable **NMSC**.

If any of the following error conditions occurs, **msgsnd()** does not send the message, sets **errno** to the value given in parentheses, and returns -1:

- *id* is not a valid message queue identifier (**EINVAL**).
- The calling process does not have permission to manipulate this queue (**EACCES**).
- Field **mtype** in the structure pointed to by *buffer* is less than one (**EINVAL**).
- *size* is less than zero or greater than the system-imposed limit (**EINVAL**).
- *buffer* points to an illegal address (**EFAULT**).

Sending a message may exceed a system-defined limit. There are two such limits: one limits the size of a queue, and the other sets the total number of messages available to your system. The maximum size of this queue is given in the field **msg_qbytes** of the structure **msqid_ds** that controls that queue. If issuing a message *size* bytes long would push the total size of the queue's messages past the value of **msg_qbytes**, then an error occurs. Likewise, an error occurs if the system already holds the maximum maximum number of message available to it, as set by the kernel variable **NMSG**.

flag indicates how **msgsnd()** is to react to either of the above conditions. If *flag* is OR'd to include value **IPC_NOWAIT**, then **msgsnd()** reacts as it does with any other error: it does not send the message, it returns -1, and it sets **errno** to an appropriate value (in this case, **EAGAIN**). If, however, *flag* is *not* OR'd to include **IPC_NOWAIT**, then **msgsnd()** waits until any of the following happens:

1. The error condition resolves. In this case, **msgsnd()** sends the message and returns normally.
2. The message queue identified by *id* is removed from the system. In this case, **msgsnd()** does not send the message; it sets **errno** to **EIDRM**; and it returns -1.
3. The process that issued the call to **msgsnd()** receives a signal. In this case, **msgsnd()** does not send the message, sets **errno** to **EINTR**, and returns -1. The calling process then executes the action requested by the signal. For information on the behavior that each signal invokes, see the Lexicon entry for **signal()**.

msgsnd() successfully sends a message, returns zero and modifies the message queue in the following manner:

- It increments by one the value in field **msg_qnum**.
- It sets field **msg_lspid** to the process ID of the calling process.
- It sets **msg_stime** to the current time.

Example

For an example of this function, see the Lexicon entry for **msgget()**.

Files

/usr/include/sys/ipc.h
/usr/include/sys/msg.h

See Also

libc, msgctl(), msgget(), msgrcv()

msig.h — Header File

Machine-dependent signals
#include <signal.h>

The header file **msig.h** defines the machine-dependent signals that the COHERENT system uses to communicate with its processes. The header file **signal.h** declares constants for the machine-independent signals, and includes **msig.h**.

See Also

header files, signal.h

Notes

This header file is obsolete, and will be dropped from a future release of COHERENT. Its use is strongly discouraged.

msqrt() — Multiple-Precision Mathematics (libmp)

Compute square root of multiple-precision integer
#include <mprec.h>
void msqrt(*a, b, r***)**
mint *a, *b, *r**;**

msqrt() sets the multiple-precision integer (or **mint**) pointed to by *b* to the integral portion of the positive square root of the **mint** pointed to by *a*. It sets the **mint** pointed to by *r* to the remainder. The value pointed to by *a* must not be negative. The result of the operation is defined by the condition

$$a = b * b + r.$$

See Also

libmp

msub() — Multiple-Precision Mathematics (libmp)

Subtract multiple-precision integers
#include <mprec.h>
void msub(*a, b, c***)**
mint **a, *b, *c;*

msub() subtracts the multiple-precision integer (or **mint**) pointed to by *a* from the **mint** pointed to by *b*, and writes the result into the **mint** pointed to by *c*.

See Also

libmp

mtab.h — Header File

Currently mounted file systems
#include <mtab.h>

The file **/etc/mtab** contains an entry for each file system mounted by the command **mount**. This does not include the root file system, which is already mounted when the system boots.

Both the commands **mount** and **umount** use the following structure, defined in **mtab.h**. It contains the name of each special file mounted, the directory upon which it is mounted, and any flags passed to **mount** (such as read only).

```
#define     MNAMSIZ     32
struct      mtab {
      char  mt_name[MNAMSIZ];
      char  mt_special[MNAMSIZ];
      int   mt_flag;
};
```

Files

/etc/mtab

See Also

header files, mount, umount

mtioctl.h — Header File

Magnetic-tape I/O control
#include <sys/mtioctl.h>

mtioctl.h defines constants and structures used by routines that control magnetic-tape I/O.

See Also

header files

mtoi() — Multiple-Precision Mathematics (libmp)

Convert multiple-precision integer to integer
#include <mprec.h>
int mtoi(*a***)**
mint **a;*

mtoi() returns an integer equal to the value of the multiple-precision integer (or **mint**) pointed to by *a*. The value pointed to by *a* should be in the range allowable for a signed integer.

See Also

libmp

mtos() — Multiple-Precision Mathematics (libmp)

Convert multiple-precision integer to string
#include <mprec.h>
char *mtos(*a***) mint** **a;*

mtos() converts the multiple-precision integer (or **mint**) pointed to by *a* to a string. It returns a pointer to the string it creates. The string is allocated by **malloc()**, and may be freed by **free()**. The base of the string is set by the value of the external variable **obase**.

See Also

libmp

mtune — System Administration

Define tunable kernel variables
/etc/conf/mtune

File **mtune** defines all of the tunable variables within the kernel. These variables let you configure some aspects of your kernel, without having to modify the kernel's drivers or recompile the kernel.

Command **idmkcoh** reads this file when it builds a new kernel, and uses its contents to help patch the newly build kernel. A **mkdev** script (kept in a subdirectory of **/etc/conf**) also sets appropriate variables within this file, based on your answers to its questions.

Each line within **mtune** defines one tunable parameter. A line consists of four fields, as follows:

1. *Name*
This field names the parameter. It cannot exceed 20 characters.

2. *Minimum Value*
The legal minimum value of this parameter.

3. *Default Value*
The default value for this parameter. This value can be overridden by an entry in file **/etc/conf/stune**.

4. *Maximum Value*
The legal maximum value of this parameter.

Note that under UNIX System V, fields 2 and 3 are reversed. A line that begins with the pound sign '#' is a comment, and is ignored by **idmkcoh** when it builds a new kernel.

For details on the parameters that this file sets, read the comments within this file.

See Also

Administering COHERENT, device drivers, mdevice, sdevice, stune

Notes

mtune contains comments that describe the kernel variables that you can tune. If you wish to tune the kernel, you should read this file and modify it appropriately. The variables are documented in this file rather than in the COHERENT manual to ensure that you have exactly accurate information about the variables that reside in the version of the kernel on your system.

mtype() — General Function (libc)

Return symbolic machine type
#include <mtype.h>
char *mtype(*type***)**
int *type***;**

mtype() takes an integer machine *type* and returns the address of a string that contains the symbolic name of the machine. The header file **mtype.h** defines the possible machine types. For example,

```
mtype(M_PDP11)
```

returns the address of the string **PDP-11**.

Files

<mtype.h>

See Also

libc

Diagnostics

mtype() returns NULL to indicate that it doesn't recognize the type of machine requested.

mtype.h — Header File

List processor code numbers
#include <mtype.h>

The header file **mtype.h** assigns a code number to each of the processors supported by Mark Williams C compilers and operating systems. These include the Intel i8086, i8088, i80186, i80286, and i80386; the Zilog Z8001 and Z8002; the DEC PDP-11 and VAX; the IBM 370, and the Motorola 68000.

See Also

header file

mult() — Multiple-Precision Mathematics (libmp)

Multiply multiple-precision integers
#include <mprec.h>
void mult(*a, b, c***)**
mint **a, *b, *c**;**

mult() multiplies the multiple-precision integers (or **mint**s) pointed to by *a* and *b*, and writes the product into the **mint** pointed to by *c*.

See Also

libmp

mv — Command

Rename files or directories
mv [-f] *oldfile* **[***newfile***]**
mv [-f] *file ... directory*

mv renames files. In the first form above, it changes the name of *oldfile* to *newfile*. If *newfile* already exists, **mv** replaces it with the file being moved; if not, **mv** creates it. If *newfile* is a directory, **mv** places *oldfile* under that directory.

In the second form, **mv** moves each *file* so that it resides under *directory*. If a file with the new name exists but is unwritable, **mv** will not delete it unless the **-f** option is specified.

mv will not copy directories between devices and will not remove directories that occupy the destination of the command.

Normally, **mv** creates a link to the old file with the new file and then removes the old file. If it cannot create the link (e.g., because the new file is on a different file system than the old), **mv** performs a copy and then removes the old file.

See Also

commands, cp, ln, mvdir

Notes

mv tests the validity of directory moves by means of search permission. The superuser always has search permission and thus can use **mv** incorrectly.

mvdir — Command

Rename a directory
/etc/mvdir *olddir newdir*

The COHERENT command **mvdir** renames directory *olddir* to *newdir*. Both can be path names.

For obvious reasons, *olddir* cannot be a subset of *newdir*. Both *olddir* and *newdir* must reside on the same file system.

See Also

commands, mv

Notes

mvdir is a link to **mv**.

mvfree() — Multiple-Precision Mathematics (libmp)

Free multiple-precision integer
#include <mprec.h>
void mvfree(*a***)**
mint ***a*;**

mvfree() frees the space allocated to an automatic multiple-precision integer (or **mint**). You should call **mvfree()** before exiting the function that uses the **mint**, or the storage used by the **val** field of the **mint** structure will never be reclaimed.

See Also

libmp

mwcbbs — Command

Download files from the Mark Williams bulletin board
mwcbbs [-cp] [-d*path***]** *directory*

The command **mwcbbs** lets you select one or more files from **mwcbbs**, the bulletin board maintained by Mark Williams Company. It displays the contents of **Contents** files that you download from the bulletin board, lets you select one or more files interactively, then constructs a **uucp** command and requests the files from Mark Williams. If all goes well, the files will be delivered to directory **/usr/spool/uucppublic** on your system. **mwcbbs**. In this way, you can obtain the latest versions of COHERENT software, sources for public-domain software that has been ported to COHERENT, and exchange mail with MWC developers and support personnel.

Options

mwcbbs recognizes the following options:

-c Force **uucp** to telephone the Mark Williams bulletin board when you exit from **mwcbbs**.

-d*path* Use *path* in place of the default receive path.

-p Print the **Contents** file. You can print all entries in a **Contents** file, or entries newer than a specified date.

mwcbbs looks for the file **.mwcbbs** in the current directory. This file contains the interface variables **DOWNPATH** and **DATAPATH**. The former names names the directory into which **uucp** is to write the requested files; and the latter names the directory where you keep data files. For example:

```
DOWNPATH=/usr/spool/uucppublic/
DATAPATH=/usr/lib/mwcbbs
```

Please note that path names are limited to 45 characters.

When you invoke **mwcbbs**, it displays a menu with the following items:

0. Contents.down
List public-domain software and shareware available for COHERENT release 3.*N* (COHERENT 286).

1. Contents.32bit
List public-domain software and shareware available for COHERENT release 4.*N* (COHERENT 386).

2. Contents.news
List news items and other informative postings from MWC.

3. Contents.UPD
List updates to COHERENT.

4. Maillist
List the mail sites available through **mwcbbs**.

5. Net_Maps
Show available network maps of world-wide UNIX sites.

6. Quit Exit from **mwcbbs**.

Downloading Files

If you select items 0 through 3 from the main menu, **mwcbbs** displays the names of files, 100 at a time. These names are read from a **Contents** file that is stored in a directory you name either with the option **-d** or the variable **DATAPATH**.

You can select one or more of these files for downloading to your system. Note that when you invoke **mwcbbs** for the first time, the only files displayed are those of the **Contents** files themselves; you must download them first, before you can begin to download other files. This is because the **Contents** are continually being updated, and also to test your UUCP with the Mark Williams bulletin board before you attempt to download a large number of source files.

To select a file for downloading, use the arrow keys to move the cursor to that file (or use the **vi** cursor-movement keys **h**, **j**, **k**, and **l**). **mwcbbs** lets you enter any of the following commands:

-s Select highlighted file name for more information or downloading. Pressing (↵) also selects the file.

-n Go to next screen (if there are more than 100 files).

-p Go to the previous screen.

-q Quit **mwcbbs**.

When you select a highlighted file, **mwcbbs** displays the following information about it:

- A summary of the file.
- The date it was added or last updated
- Other files that are required for compilation or use of selected file name.
- Other miscellaneous notes that may be of interest.
- The system commands to be generated to download the selected file from the Mark Williams bulletin board.

If a file is more than 50,000 bytes long, **mwcbbs** downloads it in parts. When a file is to be received in parts you must concatenate the parts into one file, which should be given the name of the file you selected.

Lists and Networks

Item 4 on the main menu (**Maillist**) gives you information about electronic mail sites throughout the United States. When you select this option, **mwcbbs** displays the names of the 50 states. When you select a state, as with the file lists, **mwcbbs** displays information about the mail sites in that state.

Item 5 (**Net_Maps**) gives you information about networks. When you select this option, **mwcbbs** displays a menu with the following items:

0. Net_Maps.WORLD
Network maps of UNIX sites across the world.

1. Net_Maps.USA
Network maps of UNIX sites in the United States, by area code.

2. Net_Maps.CAN
Network maps of UNIX sites in Canada, by area code.

3. Quit Return to main menu.

Options 0 through 2 display maps of available networks. You can select a map interactively, as with the file options.

Printing a Contents File

When you invoke **mwcbbs** with its **-p** option, it lists the four **Contents** files. When you select one, **mwcbbs** asks you to enter a search date. If you enter a search date, **mwcbbs** prints only those entries that are dated later than that date. If you do not enter a date, it prints every entry in the **Contents** file entries.

Entries are printed to the file **mwcbbs**. When the entries have been printed, **mwcbbs** automatically exits to the shell.

See Also

commands, UUCP
COHERENT Tutorial: **UUCP, Remote Communications Utility**

Notes

mwcbbs does not work correctly until you have correctly configured UUCP to contact the Mark Williams bulletin board. For details on how to do so, see the tutorial *UUCP, Remote Communications Utility* in the front of the COHERENT manual.

The charges for downloading a large set of files via a long-distance telephone call can be quite heavy. Much depends upon the speed of your modem and the time you place your call. *Caveat utilitor!*

n.out.h — Header File

Define n.out file structure
#include <n.out.h>

n.out.h defines the **n.out** file structure. It is the same as the standard COHERENT form **l.out**, except that it uses 32-bit addressing. This file structure is used internally in COHERENT, but is not available under the COHERENT C compiler or assembler.

See Also

coff.h header files, l.out.h

name space — C Language

C name-space rules
The term
name space
refers to the "list" where the translator records an identifier.
Each name space holds a different set of identifiers.
If two identifiers are spelled exactly the same and appear within
the same scope but are not in the same name space,
they are *not* considered to be identical.

The five varieties of name space, as follows:

Macro Names
Macro names introduced with **#define** are special. Because macro replacement happens before the program text is scanned for the other classes of names, macro names exist in a global name space that pays no heed to the rules below. See the description of name-space pollution, below, for more on this.

Label Names
The translator treats every identifier followed by a colon ':' or that follows a **goto** statement as a label.

Tags A tag is the name that follows the keywords **struct**, **union**, or **enum**. It names the type of object so declared.

Members
A member names a field within a structure or a **union**. A member can be accessed via the operators '.' or '->'. Each structure or **union** type has a separate name space for its members.

Ordinary identifiers
These name ordinary functions and variables. For example, the expression

```
int example;
```

declares the ordinary identifier **example** to name an object of type **int**.

Name-Space Pollution

The ANSI Standard and the POSIX Standard recognize special problems that relate to the above classes of name space and to the names supplied to the user by the translator or the **#include** mechanism. They provide special rules that govern what names a program and an implementation can define.

Although the above rules are good at resolving conflict, in the context of a large programming project (which the standard C library is, effectively) they are not always sufficient. First, there is the possibility that definitions in library header files may conflict with each other, or with user definitions. Second, an internal definition in the

standard library may conflict with a user definition that happens to have the same name.

The ANSI Standard defines rules that set aside some names for the implementation. The implementation can use only these names, and user applications cannot use them. When implementations and applications both obey these rules, a user program cannot conflict with a definition in a system header file. The rules are as follows:

- Any name that begins with an underscore followed by a capital letter or underscore is reserved for use by the implementation. Applications should not use any symbols of this form except to define feature-test macros (e.g., **_POSIX_SOURCE**, see below).
- Any name that begins with an underscore followed by a lower-case letter is reserved for use by the application if the name is internal (such as a static symbol or a tag- or member-name). Macro names of this form are forbidden, because they do not obey the other name-space rules above: a user-level macro definition could cause a conflict with a private structure-member defined in a system header.
- C++ reserves for the implementation all names that contain two underscores.
- The Standard forbids external identifiers (i.e., non-static functions and variables) that match any of the function or variable defined in the C standard.
- If a program **#include**s a standard library header file, it cannot use a macro definition that matches the name of any function or variable defined in any standard library header.

These rules are supplemented with rules that govern the use of names that are defined in any library header described in the ANSI Standard or the POSIX Standard. The following gives the rules that apply to individual header files:

<errno.h>
The implementation can define extra macros that begin with the letter 'E'.

<signal.h>
The implementation can define extra macros that begin with **SIG_**.

If an application needs to use any function that the POSIX Standard defines, it should contain the following line before any **#include** directives:

```
#define _POSIX_SOURCE 1
```

This sets the **_POSIX_SOURCE** feature-test macro. If this is done, the POSIX Standard reserves symbols for some header files. If an application includes one of the following header files, it must *not* use any of symbols reserved for that header:

<dirent.h>
Reserved prefix: **d_**.

<fcntl.h>
Reserved prefixes: **l_**, **F_**, **O_**, and **S_**. Reserved symbols: **SEEK_CUR**, **SEEK_END**, and **SEEK_SET**.

<grp.h>
Reserved prefix: **gr_**.

<limits.h>
Reserved suffix: **_MAX**.

<pwd.h>
Reserved prefix: **pw_**.

<signal.h>
Reserved prefixes: **sa_**, **SIG_**, and **SA_**.

<sys/stat.h>
Reserved prefixes: **st_** and **S_**.

<sys/times.h>
Reserved prefix: **tms_**.

If an application **#include**s any header described in the POSIX Standard, all symbols with the suffix **_t** are reserved.

Note that the symbols defined above that begin with an upper-case letter may be used by an application after the **#include** directive if the application uses an **#undef** directive to cancel any conflicting definition supplied by the header.

Example

The following program illustrates the concept of name space. It shows how the identifier **foo** can be used numerous times within the same scope yet still be distinguished. This is extremely poor programming style. Please do not write programs like this.

```
#include <stdio.h>
#include <stdlib.h>

/* structure tag */
struct foo {
      /* structure member */
      struct foo *foo;
      int bar;
};

main()
{
      /* ordinary identifier */
      struct foo *foo;
      int i = 0;

      foo = (struct foo *)malloc(sizeof(*foo));
      foo->bar = ++i;
      foo->foo = NULL;

/* label */
foo:  printf("What kind of \"foo\" am I?\n");
      if (foo->foo == NULL) {
            foo->foo = (struct foo *)malloc(sizeof(*foo));
            foo->foo->foo = NULL;
            foo->foo->bar = ++i;
            goto foo;
      }

      printf("The foo loop executed %d times\n", foo->foo->bar);
      return(EXIT_SUCCESS);
}
```

See Also

C language
ANSI Standard, §3.1.2.3

Notes

Pre-ANSI implementations disagree on the name spaces of structure/**union** members. The Standard adopted the "Berkeley" rules, which state that every unique structure/**union** type has its own name space for its members. It rejected the rules of the first edition of *The C Programming Language*, which state that the members of all structures and **union**s reside in a common name space.

named pipe — Definition

A *named pipe* is a special file created with the command **/etc/mknod**. Unlike the block- and character-special files created by **mknod**, a named pipe is not a device.

A named pipe acts like a conventional pipe set up between related processes. It differs in that it has a visible name that can be seen in a file system. It also differs in that it has permissions (since it's a file and has a name) associated with it just like any other file. This allows a named pipe to be accessed by processes that are *not* related to each other, and can even be used for processes that are running on behalf of different users.

The following illustrates how one process can write data into a named pipe and an unrelated process can read from it:

```
/etc/mknod my_pipe p                # create the named pipe
chmod 644 my_pipe
ls -lR / > my_pipe &                # pump data into pipe in background
mail fred < my_pipe                 # read from the pipe and process
```

This script creates a named pipe called **my_pipe** and makes sure that it is readable; it then pumps a mass of data into the pipe (in the background), and finally has a process read data from the named pipe and perform some action on them (in this case, mail the data to user **fred**). In this example, the **mail** process could be running from

another login and could either be in the foreground or background.

See Also

libsocket, mkfifo(), mknod, pipe, Using COHERENT
POSIX Standard, §5.4.2

nap() — System Call (libc)

Sleep briefly
long nap(*interval***)**
long *interval***;**

nap() sleeps for *interval* milliseconds, or until its process receives a signal, whichever occurs first.

If it receives no signal, **nap()** returns the number of milliseconds it slept. If it received a signal, it returns -1 and sets **errno** to **EINTR**.

See Also

libc, sleep()

Notes

nap() is governed by the granularity of the system clock. Under COHERENT, the system clock ticks every ten milliseconds; thus, the call **nap(1);** and the call **nap(9);** have the same effect. Note that **nap()** is guaranteed to sleep for at least *interval* milliseconds; thus, the call **nap(11);** sleeps for two clock ticks, or 20 milliseconds.

ncheck — Command

Print file names corresponding to i-node
ncheck [-i *number* **...] [-as]** *filesystem ...*

An *i-number* identifies an i-node. **ncheck** generates a list of file names by i-number for each *filesystem*, which should be the name of a device special file that contains a proper COHERENT file system. Using the raw device generally decreases the time **ncheck** requires to do its work.

The output is in the unsorted traversal order of the file system hierarchy. **ncheck** distinguishes directories from files by suffixing '/.' to directory names.

Under the **-i** option, **ncheck** prints the file name corresponding to each i-number *number* in the given list. Under the **-a** option, **ncheck** prints only the names of special files and set user-ID mode files; this option allows the system administrator to ascertain quickly whether these files represent possible security breaches.

See Also

commands, i-node

Diagnostics

ncheck appends '??' to the generated file name if it cannot find the proper parent structure while retrieving the file-name information. It represents any loops detected in the file name by the characters '...'. Extremely addled file systems may generate other reasonably self-explanatory diagnostics.

ndbm.h — Header File

Header file for NDBM routines
#include <ndbm.h>

Header file **<ndbm.h>** declares the functions used to manipulate NDBM data bases:

dbm_clearerr() . . . . . Clear an error condition on an NDBM data base
dbm_close() . . . . . Close an NDBM data base
dbm_delete() . . . . . Delete records from an NDBM data base
dbm_dirfno() . . . . . Return the file descriptor for an NDBM .dir file
dbm_error() . . . . . Check a NDBM data base for an error
dbm_fetch() . . . . . Fetch a record from an NDBM data base
dbm_firstkey() . . . . . Retrieve the first key from an NDBM data base
dbm_nextkey() . . . . . Retrieve the next key from an NDBM data base
dbm_open() . . . . . Open an NDBM data base
dbm_pagfno() . . . . . Return the file descriptor for an NDBM .pag file
dbm_rdonly() . . . . . Set an NDBM data base into read-only mode

dbm_store() . . . . . Store a record into an NDBM data base

Routines **dbm_error()** and **dbm_clearerr()** are macros that, in fact, do nothing.

This header file also defines two structures that the NDBM routines use. The first, **datum**, defines the structure of a data element, either a key or its associated data set:

```
typedef struct {
      char *dptr;
      int dsize;
} datum;
```

This structure lets you have a key and a data element of unlimited length.

The other structure, **DBM**, holds the information that the NDBM routines use to access a NDBM data base:

```
typedef struct {int dummy[10];} DBM;
```

See Also

Notes

For a statement of copyright and permissions on this header file, see the Lexicon entry for **libgdbm**.

netdb.h — Header File

Define structures used to describe networks
#include <netdb.h>

Header file **<netdb.h>** defines structures into various **sockets** functions write information about the local network. It also defines manifest constants and macros used by various **sockets** routines.

See Also

endnetent(), endprotoent(), endservent(), getnetbyaddr(), getnetbyname(), getnetent(), getprotobyname(), getprotobynumb(), getprotoent(), getservbyname(), getservbyport(), getservent(), header files, libsocket, setnetent(), setprotoent(), setservent()

networks — System Administration

Name remote networks
/etc/networks

The file **/etc/networks** names remote networks with which you can communicate, and gives information with which your system can pass datagrams to those networks.

If you wish to communicate on the Internet, you must create this file by obtaining the official network data base maintained by the Network Information Control Center (nic.ddn.mil). To this, add information about other networks not listed by NIC, with which you may wish to communicate.

If you are not going to use the Internet, you can create your own version of **/etc/networks**. Each line within **networks** describes one remote network, and consists of the following fields:

- The network's name. A network name can contain any printable character other than white space, a newline character, or the comment character '#'.
- The network's Internet-protocol (IP) address, in standard dot notation.
- Aliases, if any, for the network's name.

For example:

```
mysubnet    127.0.1          an_alias    # a comment
```

If you create your own version of **/etc/networks**, be sure to set its permissions correctly. It should be owned by the superuser **root**, and be executable.

See Also

Administering COHERENT, hosts, hosts.equiv, inetd.conf, protocols, services

newaliases — Command

Build the smail aliases data base from an ASCII source file
/usr/lib/mail/newaliases

Command **newaliases** reads the ASCII source file for an aliases data base, and builds the aliases data base according to the configuration information in **/usr/lib/mail/config**. Run this program whenever changes have been made to the ASCII source file. If this program is not used, **smail** may not notice the changes that have been made.

The aliases data base can be in a DBM data base, a sorted text file, or a plain text file. (For information about what a DBM data base is, see the Lexicon entry for **libgdbm**.) In the latter case, which is the default under COHERENT, the ASCII source file doubles as the data-base file.

To process an file, first use the command **mkline** to remove comments and regularize it. If you wish to build a sorted data base, then use the command **mksort** with its command-line option **-f** to create the sorted data base. If you wish, however, to build a DBM data base, use command **mkdbm**, again with option **-f**, to create the data base. In either case, be careful that **smail** never uses a truncated or partially built data base.

For plain text data bases, **newaliases** displays a summary of its contents, but no changes are actually made.

Files

/usr/lib/mail/aliases
The text file that defines aliases.
/usr/lib/mail/aliases.dir
/usr/lib/mail/aliases.pag
The DBM data base that is built from the text file **aliases**.
/usr/lib/mail/config
The file that gives the default configuration for **smail**.

See Also

commands, libgdbm, mail [overview], mkdbm, mkline, mksort, smail

Notes

The name **newaliases** is retained for compatibility with BSD **sendmail**. Under **smail** release 3.1, this command usually is named **mkaliases**.

newgrp — Command

Change to a new group
newgrp *group*

newgrp changes the user's group identification to the specified *group*, if access is permitted. The file **/etc/group** determines group access. Group access may be unrestricted, or open to all users with specific exceptions, or restricted to certain users via a password.

The shell executes **newgrp** directly.

Files

/etc/group — Give group access

See Also

commands, group, ksh, sh

Diagnostics

If **newgrp** succeeds, no diagnostic is printed.

Notes

Interruption of **newgrp** can result in the user being logged off.

Under the Korn shell, **newgrp** is an alias for **exec newgrp**.

newusr — Command

Add new user to COHERENT system
/etc/newusr *login "User Name" parentdir* **[** *shell* **]**

newusr adds a new user to the system. It automatically adds an entry to the file **/etc/passwd**, creates a home directory for the user, installs the user in the mail system, and otherwise performs the myriad tasks required to add a new user to your COHERENT system.

login is the login idenifier of the new user. This is a single word in lower case, by which that user is identified. Note that each user must have a unique login identifier. Identifiers are usually the user's first name, initials, or a nickname. *parentdir* is the directory or (more usually) the file system in which **newusr** will create the new user's home directory. *User Name* is the name of the human for whom *login* is being created. *shell* names the shell to be used; the default is the Bourne shell **/bin/sh**.

For example, the command

```
/etc/newusr batman "Bruce Wayne" /v /usr/bin/ksh
```

creates new user Bruce Wayne, with login **batman**, home directory **/v/batman**, and default shell **/usr/bin/ksh**.

Files

/etc/group — User groups
/etc/passwd — User passwords
/parentdir/user — User home directory
/usr/spool/mail/*user* — User mailbox

See Also

commands, passwd, welcome

Diagnostics

newusr complains if an entry for *user* already exists in the password file.

Notes

Only the superuser can add new users to the system with **newusr**.

nextkey() — DBM Function (libgdbm)

Retrieve the next record from a DBM data base
#include <dbm.h>
datum nextkey ()

Function **nextkey()** retrieves the next record from the currently open DBM data base. The data base must first have been opened by a call to **dbminit()**, and the first record within the data base must have been retrieved by a call to **firstkey()**.

nextkey() returns a pointer to the retrieved record. If no record is available (i.e., every record has already been retrieved), or if an error occurred, field **dptr** within the returned record is initialized to NULL.

You can use this function with function **firstkey()** to walk through the entire contents of a DBM data base. For example:

```
for(key=firstkey(); key.dptr!=NULL; key=nextkey(key))
```

Please note that the hashing algorithm used the DBM functions dictates which record is "next" within the data base. A loop that uses this function plus the function **firstkey()** will retrieve every record from the data base; however, the records probably will not be in the order you expect.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

nm — Command

Print a program's symbol table
nm [-**adgnopru**] *file* ...

The command **nm** prints the symbol table of each *file*. It can read binary files produced by the compiler, assembler, or linker.

When a C source file is compiled with the **-c** switch to the **cc** command, or when a file of assembly language is assembled, the result is an object module, which is signified by the suffix **.o**.

The linker **ld** links multiple object modules to form an executable program. Frequently used object modules often are grouped by the archiver **ar** into a *library*, which is signified by the suffix **.a**. **nm** can read all three kinds of files: .o, .a, and fully linked executables.

Options

nm recognizes the following options:

-a (COHERENT 286 only)
Print all symbols. Normally, **nm** prints names that are in C-style format and ignores symbols with names inaccessible from C programs.

-d Print only defined symbol.

-g Print only global symbols.

-n Sort numerically rather than alphabetically. **nm** uses unsigned compares when sorting symbols with this option.

-o Append the file name to the beginning of each output line.

-p Print symbols in the order in which they appear within the symbol table.

-r Sort in reverse-alphabetical order.

-u Print only undefined symbols.

Output Format

The output of **nm** is a series of lines of the form:

segment address symbol

segment gives the segment in which the symbol appears, or **UNDEF** for undefined symbols. *address* is either the address in hexadecimal, or the length of a common variable. *symbol* names the symbol.

For example, if **foo.o** is a relocatable object module, the output of the command **nm -o foo.o** would appears as follows:

```
#nm foo.o
UNDEF    00000000 _canl
UNDEF    00000000 _stderr
.text    0000077C acomp
.text    00000034 acomp_old
UNDEF    00000000 alloc
.text    00000F28 archive
.comm    00000004 asw
.text    000003CC csymbol
.comm    00000004 dsw
```

See Also

cc, commands, ld, size, strip

nohup — Command

Run a command immune to hangups and quits
nohup *command* **[***arguments***]**

The command **nohup** tells the COHERENT shell to execute *command* while ignoring all hangup and quit signals.

If you do not redirect the output of *command*, **nohup** redirects both the standard output and the standard error

into the file **nohup.out**. If **nohup.out** cannot be created in the current directory, **nohup** redirects all output into the file **$HOME/nohup.out**.

nohup is often used to execute scripts or pipelines that would normally abort if you logged out during the middle of execution.

Examples

If **file** is a shell script, then the command

```
nohup sh file
```

executes the contents of **file** in the foreground while ignoring all quit or hangup signals. The command

```
nohup sh file &
```

executes **file** in the background; you can log out safely and all the contents of **file** will still be executed.

See Also

commands, kill, ksh, sh, signal()

nologin — System Administration

Lock out logins
/etc/nologin

login looks for file **/etc/nologin** before it permits a user to login in. If this file exists, **login** forbids the user to log in, and instead displays on the terminal the contents of this file — which, presumably, explain why logging in is now forbidden.

You should create this file when you wish to "lock out" users during a critical time, such as when backups are being run or when the system is about to be shut down. When the critical time has passed, be sure to remove it.

login cannot lock out the superuser **root**, even if **nologin** exists. Nor will it lock out the users named in the file **/etc/trustme**, should it exist.

See Also

Administering COHERENT, login, trustme

Notes

The script **/etc/rc** removes **/etc/nologin** by default, on the assumption that after you reboot, you once again want users to be able to log in. If this is not a sound assumption, edit **/etc/rc** to change this behavior.

notmem() — General Function (libc)

Check whether memory is allocated
int notmem(*ptr***);**
char **ptr***;*

notmem() checks if a memory block has been allocated by **calloc()**, **malloc()**, or **realloc()**. *ptr* points to the block to be checked.

notmem() searches the arena for *ptr*. It returns one if *ptr* is not a memory block obtained from **malloc()**, **calloc()**, or **realloc()**, and zero if it is.

See Also

arena, calloc(), free(), libc, malloc(), memok(), realloc(), setbuf()

Notes

The only valid use for **notmem()** is in debugging code, such as in the bodies of calls to the macro **assert()**. We furthermore recommend that portable code should conditionalize use of **notmem()** so that the code may continue to compile on systems that lack such a facility.

nptx — Command

Generate permutations of users' full names
/usr/bin/nptx

The command **nptx** reads an address/name pair (that is, an address and a user's full name), and prints on the standard output as many permutations of the user's name as it can devise, each linked to the given address. A set of such permutations helps to relieve a user of the need to know the exact form of another user's name when she wishes to send mail to that user. When a set of users' names is filtered through **nptx**, the mail program **smail** can use the output as a "full-name data base".

The format of an input line is:

```
name<tab>address
```

name gives the user's first name, last name, optional middle initial, and optional nickname in parentheses; all are separated by space characters. *address* can contain any e-mail address. *name* and *address* are separated by one **<tab>** character.

nptx prints all permutations of the first names and initials, with the last name appearing in each permutation. Permutations are not necesarily unique.

Example

Given the name/address pair

```
LaMonte Cranston(Shadow)<tab>shadow@goodguy.com
```

nptx produces the following set of permutations:

```
Cranston                shadow@goodguy.com
L.Cranston              shadow@goodguy.com
LaMonte.Cranston        shadow@goodguy.com
S.Cranston              shadow@goodguy.com
Shadow.Cranston         shadow@goodguy.com
```

See Also

commands, mail, mkfnames, paths, smail

Notes

nptx normally is invoked via the script **mkfnames**, which reads a file of names (or the file **/etc/passwd** and generates a data base of names and addresses that can be used by the mail system.

nptx assumes European-style names, i.e., that the family name comes last (unlike Asian or Hungarian names, in which the family name comes first).

nrand48() — Random-Number Function (libc)

Return a 48-bit pseudo-random number as a non-negative long integer
long nrand48(*xsubi***)**
unsigned short *xsubi***[3];**

Function **nrand48()** generates a 48-bit random number, then returns its high 31 bits in the form of a non-negative **long**. The value returned is (or should be) uniformly distributed throughout the range of zero through 2^31. *xsubi* is an array of three unsigned short integers from which the pseudo-random number is built.

See Also

libc, srand48()

nroff — Command

Text-formatting language
nroff [*option ...***] [***file ...***]**

nroff is the COHERENT text-formatter and text-formatting language. By embedding commands within files of text, you can instruct **nroff** to format text, create paragraphs, subheadings, headers, footers, and in general perform all tasks required to format text for the printed page or for screen display.

nroff is designed to be used with character-display terminals or monospace printers. The related program **troff** performs typeset-quality formatting, suitable for printing on the Hewlett-Packard LaserJet printer or any printer for

which the PostScript language has been implemented. **troff**'s formatting language is a superset of that used by **nroff**. Text that you have encoded for formatting by **nroff** will work with **troff**, but the reverse is not always true. See the Lexicon entry on **troff** for information that applies to **troff** alone.

nroff Input

nroff processes each *file*, or the standard input if none is specified, and prints the formatted result on the standard output. The input must contain formatting instructions as well as the text to be processed.

Basic **nroff** commands provide for such things as setting line length, page length, and page offset, generating vertical and horizontal motions, indentation, filling and adjusting output lines, and centering. The great flexibility of **nroff** lies in its acceptance of user-defined macros to control almost all formatting. For example, the formation of paragraphs, header and footer areas, and footnotes must all be implemented by the user via macros.

The following summarizes the commands and options that can be used with **nroff**. Four types of commands and options are described: (1) command line options; (2) **nroff**'s basic commands (also called *primitives*); (3) escape sequences that can be used with **nroff**; and (4) **nroff**'s dedicated number registers, and what information each one keeps.

Command-line Options

Command-line options may be listed in any order on the command line. They are as follows:

-d Debug: print each request before execution. This options is extremely useful when you are writing new macros.

-f *name*
Write the temporary file in file *name*.

-k Keep: do not erase the temporary file.

-i Read from the standard input after reading the given *files*.

-m*name*
Include the macro file **/usr/lib/tmac.***name* in the input stream.

-n*N* Number the first page of output *N*.

-r*aN* Set number register *a* to the value *N*.

-r*abN* Set number register *ab* to value *N*. For obvious reasons, *ab* cannot contain a digit.

-v Return the number of your version.

-x Do not eject to the bottom of the last page when text ends. Use this option when you wish to use **nroff** interactively. It, too, is useful when debugging macros.

nroff appends the contents of the environmental variable **NROFF** to the beginning of the list of command-line arguments. This let you set commonly used options once in the environment, rather than retype them for each invocation of **nroff**.

Primitives

The following gives the basic commands, or *primitives*, that are built into **nroff**. These primitives can be assembled into macros, or can be written directly into the text of your document. Commands may begin either with a period '.' or with an apostrophe; the former causes a break (see **.br**, below), the latter does not.

.ab *msg*
Abort: print *msg* on the standard error and abort processing.

.ad [bclr]
Enter adjust mode: that is, insert white space between words to create right-justified output. **b** adjusts for both margins; this is the default. **c** adjusts and centers on the line. **l** adjusts, flush with the left margin. **r** adjusts, flush with the right margin.

.af *R X* Assign format *X* to number register *R*. The assigned format may be one of the following:

LEXICON

1	Arabic numerals (default)
i	Lower-case Roman numerals
I	Upper-case Roman numerals
a	Lower-case alphabetic characters
A	Upper-case alphabetic characters

.am *XX* Append the following to macro *XX*. Used like **.de**, below.

.as *XX* Append the following to string *XX*. Used like **.ds**, below.

.bp Begin a new page.

.br Break; print any fraction of a line of text that is in the input buffer before reading new text.

.c2 *c* Set the no-break control character to *c*. With no argument, reset it to the default character, which is the apostrophe.

.cc *c* Set the normal control character to *c*. With no argument, reset it to the default character, which is the period.

.ce *N* Center *N* lines of text (default, one).

.ch *XX N*
Change the location of the trap for macro *XX* to vertical position *N* on the page. Used like command **.wh**, below.

.co *endmark*
Copy input directly to the output until *endmark* is seen. If no *endmark* is given, copy until another **.co** is seen.

.cu *N* Underline the next *N* lines. When used without an argument, one line is underlined. The instruction

```
.cu 0
```

turns off underlining. Note that unlike the UNIX version of **nroff**, **.cu** does not perform continuous underlining — it underlines words, but not spaces.

.da *X* Divert and append the following text into macro *X*. A diversion is ended by a **.da** command that has no argument.

.de *X* Define macro *X*. The macro definition is ended by a line that contains only two periods "..".

.di *X* Divert the following text into macro *X*. Diversion is ended by a **.di** command that has no argument.

.ds *X value*
Define string *X* to have the given *value*.

.ec *c* Set the escape characer to *c*. With no argument, reset it to the default backslash character '\'.

.el *action*
Execute *action* when the test in an **.ie** command fails. This command must be used with an **.ie** command.

.em *XX* Execute macro *XX* when processing is completed.

.eo Escape off: turn off special handling of all escape sequences.

.ev *N* Change the environment. When followed by 0, 1, 2, the command *pushes* that environment; when used without an argument, the command *pops* the present environment and returns to the previous environment.

.ex Exit from **nroff** without further ado.

.fi Enter fill mode.

.fl Flush; same as **.br**.

.ft *X* Change the current font to *X*. **nroff** recognizes **R**, **B**, and **I**, for Roman, bold, and italic, respectively.

.ie *condition action*
This command tests to see if *condition* is true; if true, it then executes *action*; otherwise, it performs the action introduced by an **.el** primitive. This command must be used with the **.el** command.

.if *condition action*
This command tests to see if *condition* is true; if so, then *action* is executed; otherwise, *action* is ignored. The command **.if o** applies if the page number is odd, and the command **.if e** applies if the page number is even. The command **.if n** applies if the text is processed by **nroff**, and the command **.if t** applies if the text is processed by **troff**. The command **.if l** applies in landscape mode. The command **.if p** applies to **troff** PostScript mode. Note that the last two conditions are unique to the COHERENT implementation of **nroff**, and may not be portable to other implementations.

.ig *X* Ignore all input until macro *.X* is called; if no argument is given, ignore input until two periods "..".

.in *NX* Change the normal indentation to *N* units of *X* scale. *X* can be **u** or **i**, for *machine units* or *inches*, respectively. If *N* is used without *X*, **nroff** assumes the indentation to be given in number of character-widths (in picas, or tenths of an inch). Default indentation is zero.

.it *N XX*
Set an input trap to execute macro *XX* after *N* input lines (not counting request lines).

.lc *c* Set the leader dot character to *c*. When **nroff** sees the escape sequence **\a**, it fills space to the next tab stop with the leader dot character. **lc** with no argument tells **nroff** to use spaces to fill leaders.

.ll *NX* Set the line length. Used like the **.in** command, above.

.ls *X* Leave spaces; insert *X* vertical spaces after each line of text. Default is zero.

.lt *NX* Length of title. Used like the **.in** command, above.

.na Enter no-adjust mode. Line lengths are not changed.

.ne *NX* Confirm that at least *N* portions of *X* units of measure of vertical space are needed before the next trap. If this amount of space is not available, then move the text to the top of the next page. *X* can be **i** or **v**, for inches or vertical spaces, respectively. This command is used in display macros and in paragraph macros to help prevent widows and orphans.

.nf Enter no-fill mode; no right justification is performed, although line lengths are changed to approximate uniform line length.

.nh Turn off hyphenation. **nroff** hyphenates according to built-in algorithms that are correct most of the time, but not always.

.nr *X N1 N2*
Set number register *X* to value *N1*; set its default increment/decrement to *N2*. For example, **.nr X 2 3** sets number register **X** to 2, and sets its default increment to 3.

The basic unit of measurement for **nroff** 1/120th of an inch; this is also called the *machine unit*. It is indicated by the suffix **u** to a measurement. Unless otherwise stated, all number registers that information about a page holds that information in **nroff** machine units.

Other units of measure convert into **nroff** units as follows:

inch:	1**i** = 120**u**
vertical line space:	1**v** = 20**u**
centimeter:	1**c** = 47**u**
em:	1**m** = 12**u**
en:	1**n** = 12**u**
pica:	1**P** = 20**u**
point:	1**p** = 1**u**

.ns No-space mode.

.nx *file* Terminate processing of the current input file and begin processing *file* instead.

.pl *NX* Set the page length to *N*. The unit of measure *X* can be **V** or **i**, for vertical spaces (sixths of an inch) or inches, respectively. The default unit of measure is vertical spaces.

.pn *N* Set the page number to *N*.

.po *NX* Set the default page offset to *N*. The unit of measure *X* can be set to **i**, for inches. The default unit of measure is number of characters.

LEXICON

.rb *file* Read binary: read the given *file* and copy it directly to the output without processing.

.rd *prompt*
Read an insertion from the standard input after issuing the given *prompt*.

.rf *XX YY*
Rename font *XX* as *YY*. For example, to have calls to font K remapped to Roman font, use the call:

```
.rf K R
```

.rm *XX* Remove macro or string *XX*.

.rn *XX YY*
Change the name of a macro or string from *XX* to *YY*.

.rr *X* Remove register *X*.

.rs Restore normal space mode.

.so *file* Open *file*, read its contents, and process them. When the end of *file* is reached, resume processing the contents of the present file.

.sp [|]*NX*
Space down *N*. The unit of measure *X* can be **i**, for inches, with the default unit of measure being vertical spaces, or sixths of an inch. The optional vertical bar '|' indicates that *N* is an absolute value; for example, **.sp |1.5i** means to move to 1.5 inches below the top of the page, whereas **.sp 1.5i** means to move to 1.5 inches below the present position.

.sy *command*
Execute *command* under the shell. Please note that this primitive is non-standard. Macros that use it cannot be formatted under standard AT&T **nroff**.

.ta *NX ...*
Set the tab to *N*. The unit of measure *X* can be set to **i**, for inches; the default unit of measure is number of characters, or tenths of an inch. A tab setting, of course, is for an absolute, not a relative, value. If more than one tab setting is defined, the first defines the first tabulation character on a text line, the second defines the second tabulation character, etc. Any undefined tabulations are thrown away.

.tc *X N* Fill any unused space within a tabulation field with the character *X*. If the optional *N* is present, it specifies a width for the character; for example, **.tc . .1i** fills tabs with dots spaced one-tenth of an inch apart.

.ti *NX* Temporary indent; indent only the next line. Used like the **.in** command, above.

.tl *'left'center'right'*
Set a three-part title, with *left* being set flush left, *center* being centered on the line, and *right* being set flush right. Note the use of the apostrophes to separate the fields; the apostrophes for an undefined field must still be present, or a syntax error will be generated.

.tm *message*
Print *message* on the standard error device. This is often used with **.if** or **.ie** commands to indicate an error condition.

.tr *xy* Translate character *x* to *y* on output.

.ul *N* This behaves the same as **.cu**.

.vs *Np* Reset the normal vertical spacing to *N* points **p**. One point equals 1/72 of an inch. The default setting one pica, which equals is 12 points or 1/6 of an inch.

.wh *NX action*
Set a trap to perform *action* when point *N* is reached on every formatted page. If *N* is negative, it is measured up from the bottom of the page. The unit of measure *X* may be **i** or **v**, for inches or number of vertical lines, respectively; the default unit of measure is **v**.

Escape Sequences

The following lists **nroff**'s escape sequences, or commands that suspend or work around the normal operation of **nroff**. Each escape sequences is introduced by the *escape character*, normally the backslash character '\':

\(*xx* Print special character *xx*, as defined by a **.dc** request. **nroff** reads default special character definitions from file **/usr/lib/roff/nroff/specials.r**. For example, the escape sequence **\(<=** prints the less-than-or-equal-to symbol ≤.

\. Print a literal period.

\' Print a literal apostrophe. This should be used in text that will be manipulated by the **\w** escape sequence or the **.tl** primitive.

**** Delay interpretation of a backslash character. This normally is used to defer the interpretation of a macro or string from the time it is processed to the time that it is called.

\- Print a minus sign.

\& Ignore what is normally a command string.

\$*N* Call macro argument *N*.

\'' Introduce a comment within your text. All text to the right of this escape sequence will be ignored by **nroff**. This sequence must read **.\''** when used at the beginning of a line.

*****S* Call string *S*.

(ST* Call string *ST*.

\a Fill the space to the next tab stop with leader dots (normally '.').

\d Move *down* by one-half em (**troff**) or one-half line (**nroff**). Normally used to do crude subscripting, or to undo the effect of the **\u** escape sequence.

\e Print the escape character in the output text — normally, a backslash.

\f*X* Set font to *X*; this can be either **R**, **I**, **B**, or **P**, for Roman, *italic*, **bold**, or previous font, respectively.

\h'[|]*NX***'**
Move horizontally by *N* units of *X*. If *N* is positive, move to the right; if negative, move to the left. The unit of measure *X* may be **i**, for inches; the default unit of measure is ems. (One em equals one pica, which is one-sixth of an inch). When the optional vertical bar '|' is used, move to an absolute position on the line. For example **\h'|1.5i'** moves to 1.5 inches to the right of the left margin, whereas **\h'1.5i'** moves 1.5 inches to the right of the current position.

\k*x* Record the current vertical position into register *x*.

\l'*NX***'** Draw a horizontal line *N* units of *X* long. The unit of measure *X* may be **i**, for inches; the default unit of measure is character-widths.

\L'*NX***'** Draw a vertical line; used like **\l**, above.

\n*X* Read the value of number register *X*.

\n(*XY* Read the value of number register *XY*.

\o'*chars***'**
Overstrike the given *chars*, centered on the widest.

\s*N* Change the current size of the type to *N* points.

\s+*N* Increment the current point size by *N* points.

\s-*N* Decrement the current point size by *N* points.

\t Print a tab.

\u Move *up* by one-half em (**troff**) or one-half line (**nroff**). Normally used to do crude superscripting, or to reverse the effect of the **\d** escape sequence.

\v'*NX***'** Vertical motion; move *N* units of *X* vertically. If *N* is positive, move down; if negative, move up. The unit of measure *X* may be **i** or **v**, for inches or vertical spaces (sixths of an inch), respectively. The default unit of measure is **v**.

LEXICON

\w'*argument***'**
Measure the width of *argument*. For example

```
\w'stuff and nonsense'
```

measures the width of the phrase **stuff and nonsense**; or

```
\w'\$1'
```

measures the width of the first argument passed to a macro, whatever that argument might happen to be. Therefore, the command **.in \w'\$1'** will indent a line by the width of argument 1.

\X*dd* Output the character with hexadecimal value *dd*, where *dd* are two hexadecimal digits. Users can use this option to encode characters that are not part of the English-language character set. The hexadecimal values to which characters map depend upon the character set that you (or your printer) use. Please note **nroff** reserves the following values for its internal use:

<Ctrl-SP>	X00	Ignored
<Ctrl-A>	X01	Leader dots, same as "\a"
<Ctrl-I>	X09	Tab, same as "\t"
<Ctrl-J>	X10	Newline

This escape sequence is unique to the COHERENT implementation of **nroff** and **troff**. Code that uses it will behave differently when ported to other implementations.

\z*c* Print character *c* with zero width.

\<newline>
Ignore this **<newline>** character.

\{ Begin conditional commands; used after an **.if**, an **.ie**, or an **.el** command.

\{ Begin conditional commands, and ignore the following carriage return.

\} End conditional commands.

Dedicated Number Registers

The following lists the number registers that are predefined in **nroff**. You can read or reset these registers to suit the need of any special formats that you wish to devise.

$$ Process identifier of the current **nroff** process. This usually is used with the primitive **.sy** to name temporary files.

.$ Number of arguments passed to a macro.

% Present page number.

.c Number of lines read from the current input file. This can be used to help set an input-line trap.

.d Current vertical position in the current diversion. If no diversion is opened, this register's contents equal those of the **nl** register, described below.

dl Maximum width of last completed diversion.

dn Height of last completed diversion.

dw Day of the week (one through seven; one indicates Sunday).

dy Day of the month, as set by COHERENT.

.F Name of input file being read. This is very useful for printing error messages. This register applies only the COHERENT implementation of **nroff**. Code that uses it is not portable to other implementations.

.h Vertical position of the current line's base-line. This number register gives you the best idea of your current vertical position on the page.

hp Horizontal position on current input line.

.i Present amount of indentation.

.j Current type and mode of text adjustment.

.l Present line length.

ln Current line number in the output.

mo Month, as set by COHERENT.

.n Width of the text portion of the previously printed line. Useful for underlining, shading, or otherwise modifying the previous line of text. For example

```
\l'\n(.nu'
```

draws a line under the previously printed line of text.

nl Vertical position of the base-line of the last printed line of text.

.o Present page offset.

.p Page length.

.s Size of the type currently being printed, in points.

sb Depth to which a string hangs below its base line. This is generated by the width function.

st Height to which a string extends above its base line. This is generated by the width function.

.t Distance to the next trap. Check this register to see if the object you wish to print on a page will fit.

.v Size of a line, in points. This is set by the **vs** primitive.

yr Last two digits of the year, as set by COHERENT.

.z Name of the current diversion.

Printer Configuration

nroff reads several files in directory **/usr/lib/roff/nroff** to find printer-specific information. It reads special character definitions from file **specials.r.** If file **fonts.r** exists, **nroff** reads font information from it; **nroff** understands only Roman, bold and italic fonts, but **.rf** requests may define alternative font names. If file **.pre** exists, **nroff** copies it at the beginning of the output. If file **.post** exists, **nroff** copies it at the end of the output. In landscape mode, **nroff** looks for files **.pre_land** and **.post_land** instead. You can change these files as desired to include printer-specific commands in **nroff** output.

Miscellaneous

The **-ms** macro package is kept in file **/usr/lib/tmac.s.** The macros in this package are more than sufficient for most ordinary text processing. Beginners should work through this macro package rather than trying to deal at once with the basic program.

The tutorial to **nroff**, which is included with this manual, provides a detailed introduction to **nroff**. Error messages for **nroff** appear in the appendix to this manual.

Files

/tmp/rof* — Temporary files
/usr/lib/tmac.* — Standard macro packages
/usr/lib/roff/nroff/ — Support files directory
/usr/lib/roff/nroff/.pre — Output prefix
/usr/lib/roff/nroff/.pre_land — Output prefix, landscape mode
/usr/lib/roff/nroff/.post — Output suffix
/usr/lib/roff/nroff/.post_land — Output suffix, landscape mode
/usr/lib/roff/nroff/fonts.r — Alternative font name definitions
/usr/lib/roff/nroff/specials.r — Special character definitions

See Also

col, commands, deroff, man, ms, printer, troff
nroff, The Text-Formatting Language, tutorial

Diagnostics

nroff returns the following error messages. Most are self-explanatory.

LEXICON

-f option requires file argument *(fatal)*
.bd not implemented yet
.co: unexpected EOF before *string (error)*
.dt not implemented yet
.el without .ie *(error)*
.fc not implemented yet
.hc not implemented yet
.hw not implemented yet
.hy not implemented yet

.ie nested more than *N* levels *(error)*
The **.ie/.el** combination can be nested only 15 levels deep.

.ie without matching .el *(error)*
Every **.ie** must be followed by an **.el**.

.lf: *string*, file "*string*" *(error)*
troff could not load a font-width table from file *string*.

.lf: "*string*" is not a PCL font width table *(error)*
troff expects a PCL font-width table, but file *string* is not in the PCL font-width format.

.lf: "*string*" is not a PostScript font width table *(error)*
troff expects a PostScript font-width table, but file *string* is not in the PostScript font-width format.

.lf: cannot load more than *N* fonts *(error)*
troff has a static limit on the number of font-width tables that can be loaded at one time.

.lf: cannot open file "*string*" *(error)*
.lf: requires fontname and filename *(error)*
.nm not implemented yet
.nn not implemented yet
.pi not implemented yet
.rb: cannot open file *string (error)*
.rb: no file specified *(error)*
.rf: requires name and new name *(error)*

\} without matching \{ *(error)*
Every **\}** must be preceded by a **\{**.

arguments too long *(error)*
attempted zero divide *(error)*
attempted zero modulus *(error)*
bad adjustment type *(error)*
bad argument reference *(error)*
bad directive *N (fatal)*
bad font *N (fatal)*
bad font *N* at dev_font, nfonts=*N (fatal)*
bad font *N*, nfonts=*N (fatal)*
bad pattern *(fatal)*
bad tab stop *(error)*
bad tab stop *(error)*

botch: fontname(*N*) *(fatal)*
nroff cannot handle font *N* and must abort processing.

botch: swdmul=*N* psz=*N* swddiv=*N (fatal)*
An undefined error has occurred within **nroff**. The printed numbers give the value of **nroff**'s internal registers. If such an error occurs regularly when you process a given piece of text, please send the text in question and a copy of the error message to Mark Williams technical support.

bracket building not implemented yet
cannot create temp file *(fatal)*
cannot dehyphenate *(fatal)*

cannot end diversion *(error)*
: You attempted to close a diversion without first opening one.

cannot find current file *(error)*

cannot find font *XX (error)*
: Font *XX* has not been opened; therefore **[nt]roff** cannot use it. To open a font, use the load-font primitive **.lf**.

cannot find font *N (error)*

cannot find register *string (error)*
: You attempted to read a number register without first loading a value into it.

cannot open *string (error)*
cannot open file "*string*" *(error)*

cannot pop environment *(error)*
: You popped an environment without first pushing one.

cannot read environment *(fatal)*
cannot remove *string (error)*
cannot reopen temp file *(fatal)*
cannot write environment *(fatal)*
delimiter argument too large *(error)*
diversion buffer odd alignment *(fatal)*
environment does not exist *(error)*
environments stacked too deeply *(error)*
field with too large *(error)*
file "*string*" not found *(error)*
flushd -- current diversion null *(fatal)*
font position out of range *(error)*

fonts.r not found *(fatal)*
: **nroff** and **troff** read the list of fonts to use from a file named **fonts.r**. If you do not have such a file in your current directory, **nroff** and **troff** read the one out of their home directories: **/usr/lib/roff/nroff**, **/usr/lib/roff/troff_pcl**, or **/usr/lib/roff/troff_ps**, depending which variety of output you have requested. This error message means that your current directory does not hold a file named **fonts.r**, and that **[nt]roff** cannot open the **fonts.r** file in its appropriate home directory.

illegal hex digit *(error)*
: The escape sequence **\X***NN* prints a character by its literal hexadecimal value. This should be used when processing characters that are not normally printable on the terminal screen. Digit *N* can be the numerals '0' through '9', the letters 'a' through 'f', or the letters 'A' through 'F'. All other characters will trigger this error.

illegal option: *string (fatal)*

incomplete macro in trap *(fatal)*
: A trap has jumped to a macro, but that macro does not terminate, for whatever reason. Usually this indicates that you have opened a diversion but failed to close it.

line buffer overflow *(fatal)*
no room for new font name *XX (error)*
out of space - memory *string (fatal)*
request '*string*' not found *(error)*
section *N* of title too large *(error)*
special character *XX* not found *(error)*

syntax error *(error)*
: This message any number of errors with your **nroff** source. Check the line number given in the message.

temporary file write error *(fatal)*
: **nroff** cannot write a temporary file, for whatever reason. This usually indicates that you lack permission to write into the directory into which **nroff** is attempt to write its temporary files.

LEXICON

too many tab stops *(error)*
: **nroff** allows a maximum of nine tab stops in one line. It ignores all tab stops that exceed that limit.

unexpected end of file *(fatal)*
: This error indicates that **nroff** is in the middle of processing a macro when the file ends. This error usually occurs when you open a diversion and fail to close it.

unknown macro/register type *N (fatal)*
vertical line drawing not implemented yet *(error)*
word buffer overflow *(fatal)*

NUL — Definition

NUL is the ASCII null character '\0' — i.e., the character with the value zero. Do not confuse it with the null pointer NULL or with the empty string "". A C-language string is always terminated with a NUL. The empty string "" is an array of **char**s with only one element, namely a NUL.

See Also

ASCII, NULL, Programming COHERENT

NULL — Manifest Constant

The manifest constant **NULL** is defined in the header file **stddef.h**. It is the null pointer **(char *)0**, which is a pointer initialized to zero. Numerous routines return this value to indicate failure; it is useful as a return value because it points nowhere, and so removes the possibility of accidentally destroying a section of memory after failure.

See Also

manifest constant, NUL, pointer, stdio.h
ANSI Standard, §7.1.6

null — Device Driver

The 'bit bucket'

All data written to the special file **/dev/null** are thrown away (sent to the "bit bucket"). This is useful, for example, when you wish to test a program's side effects while ignoring its output.

A read from file **/dev/null** returns end of file (zero bytes of data). The shell **sh** uses **/dev/null** as input to background processes.

Files

/dev/null

See Also

device drivers, idle, ksh, sh

nybble — Definition

A **nybble** is four bits, or half of an eight-bit byte. The term is generally used to refer to the low four bits or the high four bits of a byte. Thus, a byte may be said to have a "low nybble" and a "high nybble". One nybble encodes one hexadecimal digit.

See Also

bit, byte, Programming COHERENT

object format — Definition

An *object format* describes the form of compiled program that still contains relocation information. The linker **ld** reads file in object format to create executable files.

COHERENT creates object modules that are in the format **l.out**.

See Also

l.out, ld, Programming COHERENT

od — Command

Print an octal dump of a file
od [-bcdox] [*file***] [[+]** *offset***[.][b]]**

od prints the specified *file* as a sequence of octal numbers, or machine words. If no *file* is specified, **od** dumps the standard input.

The following options set the format of **od**'s output:

- **-b** Bytes in default base
- **-c** Bytes in ASCII characters
- **-d** Words in decimal
- **-o** Words in octal
- **-x** Words in hexadecimal

The default base is octal on the PDP-11 and hexadecimal on the i80286, Z-8001, and M68000 families of microprocessors.

Dumping can start at position *offset* into the file. The specified *offset* is octal unless the '.' suffix is present to signify decimal. *offset* is in bytes unless the **b** suffix is present to signify 512-byte blocks.

See Also

ASCII, commands, conv, db, strings

offsetof() — General Macro (stddef.h)

Offset of a field within a structure
#include <stddef.h>
size_t offsetof(*structname***,** *fieldname***);**

offsetof() is a macro that is defined in the header **<stddef.h>**. It returns the number of bytes that the field *fieldname* is offset from the beginning of the the structure *structname*.

offsetof() may return an offset for *fieldname* that is larger than the sum of the sizes of all the members that precede it. This will be due to the fact that some implementations insert padding into a structure to ensure that they are properly aligned.

Example

The following example displays the offset of some fields within a structure:

```
#include <stddef.h>
```

```
struct foo {
      char a[13];
      long b;
      char c[7];
      short d;
      char e[3];
};

main ()
{
      int A, B, C, D, E;

      A = offsetof(struct foo, a[0]);
      B = offsetof(struct foo, b);
      C = offsetof(struct foo, c[0]);
      D = offsetof(struct foo, d);
      E = offsetof(struct foo, e[0]);

      printf ("%d %d %d %d %d\n", A, B, C, D, E);
}
```

When run, this program prints:

```
0 16 20 28 30
```

Note that even though field 'a' of structure **foo** is only 13 bytes long, field 'b' is aligned at byte 16. This is done to conform to the requirements of COFF. For details, see the section on "COFF Linking" in the Lexicon entry for the linker **ld**.

See Also

alignment, C language, ld, libc, stddef.h, struct
ANSI Standard, §7.1.6

open() — System Call (libc)

Open a file
#include <fcntl.h>
int open(*file***,** *type***[,** *mode***])**
char **file***; **int** *type*; **[int** *mode*;]*

open() opens a *file* to receive data, or to have its data read. When it opens *file*, **open()** returns a file descriptor, which is a small, positive integer that identifies the open *file* for subsequent calls to **read()**, **write()**, **close()**, **dup()**, **dup2()**, or **lseek()**. After *file* is opened, reading or writing begins at byte 0.

The second argument, *type*, determines how the file is opened. It is a bitwise OR of flag bits taken from the following list (as defined in the header file **<fcntl.h>**):

O_RDONLY	Read only
O_WRONLY	Write only
O_RDWR	Read and write

One, and only one, of the above three bit values must be set in *flag*. The following bit values can be used to describe further how the file can be opened:

O_NDELAY	Non-blocking I/O
O_APPEND	Append (writes guaranteed at the file's end)
O_SYNC	Sync on every write
O_TRACE	For file system debugging (*non-standard*)
O_NONBLOCK	Non-blocking I/O
O_CREAT	Open with file create (third argument)
O_TRUNC	Open with truncation
O_EXCL	Exclusive open
O_NOCTTY	Do not assign a controlling tty

The remaining bit values are used to how you wish to manipulate *file*:

O_APPEND
Precede every write with an automatic seek to end of *file*.

O_CREAT
If *file* does not exist, create it. If this flag is set the third argument, *mode*, sets the mode on the file. Note that this mode will be masked by **umask()**. See the Lexicon article on the command **chmod** for details on what the various values of *mode* mean.

O_EXCL
Exclusive open: this flag is meaningful only if **O_CREAT** is also used. In that case, **open()** fails with error value **EEXIST** if *file* already exists.

O_NDELAY
No delay in writing to disk. Please note the following caveats when using this flag:

If set: Opening a FIFO with **O_RDONLY** returns without delay. Opening a FIFO with **O_WRONLY** returns an error if no process has the file open for reading. Opening a file associated with a communication line returns without waiting for a carrier signal.

If not set:
Opening a FIFO with **O_RDONLY** blocks until a process opens the file for writing. Opening a FIFO with **O_WRONLY** blocks until a process opens the file for reading. Opening a file associated with a communication line blocks until a carrier signal is present.

O_NOCTTY
If *file* names a terminal device, do not set it to be the controlling terminal for the process.

O_SYNC
All writes to *file* will be synchronous to disk. This means that **write()** will not return until the data have been physically written to disk.

O_TRUNC
If *file* exists, truncate it to zero length. You must have write permissions on *file* to use this flag.

The third argument, *mode*, is significant only if **O_CREAT** is specified in the second argument and if *file* did not exist before the call to **open()**. In that case, *mode* specifies the access permissions of the new file, in exactly the manner that the system call **creat()** uses its *mode* argument to set permissions. The value of *mode* is typically given as either an octal constant or bitwise OR of permission-bit values as defined in header file **<sys/stat.h>**.

Example

This example copies the file named in **argv[1]** to the one named in **argv[2]** by using system calls. It demonstrates **open()** plus the system calls **close()**, **read()**, **write()**, and **creat()**.

```
#include <stdio.h>
#include <fcntl.h>
#define BUFSIZE (20*512)
char buf[BUFSIZE];

void fatal(s)
char *s;
{
      fprintf(stderr, "copy: %s\n", s);
      exit(1);
}

main(argc, argv)
int argc; char *argv[];
{
      register int ifd, ofd;
      register unsigned int n;
```

```
      if (argc != 3)
            fatal("Usage: copy source destination");
      if ((ifd = open(argv[1], O_RDONLY)) == -1)
            fatal("cannot open input file");
      if ((ofd = open(argv[2], O_CREAT | O_RDWR | O_TRUNC, 0666)) == -1)
            fatal("cannot open output file");
      /* For COHERENT 286, use creat() instead of open():
       * if ((ofd = creat(argv[2], 0666)) == -1)
       */

      while ((n = read(ifd, buf, BUFSIZE)) != 0) {
            if (n == -1)
                  fatal("read error");
            if (write(ofd, buf, n) != n)
                  fatal("write error");
      }

      if (close(ifd) == -1 || close(ofd) == -1)
            fatal("cannot close");
      exit(0);
}
```

See Also

fopen(), file descriptor, close(), libc
ANSI Standard, §4.9.3
POSIX Standard, §5.3.1

Diagnostics

open() returns -1 if the file does not exist, if the caller lacks permission, or if a system resource is exhausted.

Notes

open() is a low-level call that passes data directly to COHERENT. It should not be mixed with high-level calls, such as **fread()**, **fwrite()**, or **fopen()**.

Code that uses the third argument to **open()** cannot be ported to COHERENT 286.

COHERENT release 4.2.10 changes some of the behaviors triggered by flags **O_EXCL** and **O_NDELAY**. In previous release of COHERENT, flag **O_EXCL** COHERENT would handle blocking subsequent **open()**s. This is no longer the case — the device driver must handle it. In previous release of COHERENT, when flag **O_NDELAY** was used to open a character driver, the I/O flag **IONDLY** would be set. Now, the I/O flag **IONONBLOCK** is set instead.

opendir() — General Function (libc)

Open a directory stream
#include <sys/types.h>
#include <dirent.h>
DIR ***opendir (***dirname**)**
char **dirname**;**

The COHERENT function **opendir()** is one of a set of COHERENT routines that manipulate directories in a device-independent manner. It opens a directory stream and connects the directory *dirname* with it.

opendir() returns a pointer to the directory stream it has created. It returns NULL if it cannot access *dirname*, if *dirname* is not a directory, or if it cannot create the directory stream (perhaps due to insufficient memory).

If an error occurs, **opendir()** exits and sets **errno** to an appropriate value.

Example

The following example searches the current working directory for entry **FOO**:

```
#include <stddef.h>
#include <sys/types.h>
#include <dirent.h>
```

```
main()
{
      DIR *dirp
      struct dirent *dp;

      dirp = opendir( "." );

      while ((dp = readdir( dirp )) != NULL ) {
            if ( strcmp( dp->d_name, "FOO" ) == 0 ) {
                  printf("Found FOO\n");
                  (void) closedir(dirp);
                  return FOUND;
            }
      }

      (void) closedir( dirp );
      printf("FOO not found\n");
      return NOT_FOUND;
}
```

See Also

closedir(), dirent.h, getdents(), libc, readdir(), rewinddir(), seekdir(), telldir()
POSIX Standard, §5.1.2

Notes

The **dirent** routines buffer directories; and because directory entries can appear and disappear as other users manipulate the directory, your application should continually rescan a directory to keep an accurate picture of its active entries.

The COHERENT implementation of the **dirent** routines was written by D. Gwynn.

operator — Definition

An **operator** is a function that is built into the C language. It usually relates one operand to another. For example, the statement

```
1+2
```

relates the operands **1** and **2** through the operation of addition; on the other hand, the statement

```
A>B
```

relates the operands **A** and **B** logically, by asserting that the former is greater than the latter; whereas

```
A=B
```

relates the operands **A** and **B** by assigning the value of the latter to the former. The following is a table of the C operators:

*	Multiplication
/	Division
%	Remainder
+	Addition
-	Subtraction
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
&&	Logical AND
!=	Inequality
!	Logical negation
| |	logical OR
&	Bitwise AND
^	Bitwise exclusive OR
~	Bitwise complement
|	Bitwise inclusive OR

<<	Bitwise shift left
>>	Bitwise shift right
=	Assign
+=	Increment and assign
-=	Decrement and assign
*=	Multiply and assign
/=	Divide and assign
%=	Modulus and assign
++	Increment
--	Decrement
==	Equivalence
&=	Bitwise AND and assign
^=	Bitwise exclusive OR and assign
|=	Bitwise inclusive OR and assign
<<=	Bitwise shift left and assign
>>=	Bitwise shift right and assign
*	Indirection
&	Render an address
()	Function indicator
[]	Array indicator
->	Structure pointer
.	Structure member
? :	Conditional expression
sizeof	size of an object

Precedence

Precedence refers to the order in which C executes operators. The C languages assigns a level of precedence to each operator. Operators are executed in the order of their precedence level, from highest to lowest.

The following table summarizes the precedence of C operators. The are listed in *descending* order of precedence: those listed higher in the table are executed before those lower in the table. Operators listed on the same line have the same level of precedence, and the implementation determines the order in which they are executed. If you use two or more such operators in the same expression, you would be wise to use parentheses to indicate exactly the order in which you want the operators executed.

Operator	*Associativity*
`() [] -> .`	Left to right
`! ~ ++ -- - (`*type*`) * & sizeof`	Right to left
`* / %`	Left to right
`+ -`	Left to right
`<< >>`	Left to right
`< <= > >=`	Left to right
`== !=`	Left to right
`&`	Left to right
`^`	Left to right
`|`	Left to right
`&&`	Left to right
`||`	Left to right
`?:`	Right to left
`= += -= *= /= %=`	Right to left
`,`	Left to right

You can always determine precedence in an expression by enclosing sub-expressions within parentheses: the expression enclosed within the innermost parentheses is always executed first.

See Also

Programming COHERENT, sizeof
ANSI Standard, §6.1, §6.3

PAGER — Environmental Variable

Specify Output Filter
PAGER=''*command options*''

The environmental variable **PAGER** directs programs such as **msgs**, **mail** and others to "pipe" their output into the *command* specified as the value of **PAGER**. For example, the following sets up **/bin/scat** as the desired output filter and passes a command line option to it to specify that the output screen has 20 lines.

```
export PAGER="exec /bin/scat -l20"
```

See Also

scat, environmental variables, mail, more, msgs

param.h — Header File

Define machine-specific parameters
#include <sys/param.h>

param.h defines machine-specific parameters. These parameters set limits on the operation of the COHERENT system; e.g., the number of files that can be open at any one time.

See Also

header files

Notes

This header file is obsolete, and will be dropped from a future release of COHERENT. Its use is strongly discouraged.

passwd — Command

Set/change login password
passwd [*user***]**

passwd sets or changes the password for the specified *user*. If *user* is not specified, **passwd** changes the password of the caller.

passwd requests that the old password (if any) be typed, to ensure the caller is who he claims to be. Next it requests a new password, and then requests it again in case of typing errors. **passwd** requests a longer password if the one given is too brief or too simple.

Files

/etc/shadow — Encrypted passwords

See Also

commands, crypt(), login

Notes

One good way to construct a password is to concatenate two common words plus a punctuation mark. For example, "dog~collar" or "hamlet&horatio" are passwords that are both easy to remember and difficult to guess.

passwd — System Administration

Define system users

The file **/etc/passwd** holds information about each user who has permission to use the COHERENT system. This information is read by the commands **login** and **passwd** whenever a user attempts to log in, to ensure that that user is really himself and not an impostor.

/etc/passwd holds one record for each user; each record, in turn, consists of seven colon-separated fields, as follows:

```
name:password:user_id:group_id:comments:home_dir:shell
```

name is the user's login name.

password is his encrypted password. If this field holds a single asterisk '*', then the program **login** reads his password out of the file **/etc/shadow**.

user_id is a unique number that is also used to identify the user. *group_id* identifies the group to which the user belongs, if any.

comments holds miscellaneous data, such as names, telephone numbers, or office numbers.

home_dir gives the user's home directory.

Finally, *shell* gives the program that is first executed when the user logs on; in most instances, this is an interactive shell (default, **/bin/sh**).

If you wish, you can set additional passwords to control users who attempt to log into your system remotely (that is, via a modem). You can set a different remote-access password for each group of users, based on the program invoked when they log in; for example, you can set one password for the users who log in and invoke **uucico**, and another for the users who log in and invoke the interactive shells **ksh** or **sh**. For details on how to do this, see the Lexicon entries **d_passwd** and **dialups**.

When a user creates a file, that file is "owned" by him. For example, whenever user **joe** create a file, that file is "owned" by **joe**; and **joe** has user-level permissions on that file. The superuser **root** can use the command **chown** to change the ownership of a file from one user to another. For details on this command, see its entry in the Lexicon.

See Also

Administering COHERENT, chown, passwd [command]

Notes

/etc/passwd can be read by anyone: if access to it were refused to a user, he could not log on. Thus, the passwords encrypted within it can be read and copied by anyone, and so may be vulnerable to brute-force decryption. For this reason, close attention should be paid to passwords: they should not be common words or names, preferably mix cases or use unique spellings, and be at least six characters long.

paste — Command

Merge lines of files
paste [-s] [-d *list***]** *file ...*

paste merges corresponding lines from multiple input files. By default, **paste** uses the **<tab>** character to delineate texts from different files. **paste** writes the merged text to standard output; thus, **paste** can be used at the head of a shell pipeline.

If **paste** reads EOF from any of the input files while other files still contain data, it substitutes blank lines as input from the file that has ended.

Options

paste recognizes the following command-line options:

-d *list* Use the characters in *list* to separate the output fields. The characters in *list* are taken in sequence and used circularly, i.e., taken in order until the end of *list* is reached, then returning to the first character in *list*. By default, **paste** uses the **<tab>** character to delineate the output fields. The following character sequences have special meaning when encountered in *list*:

**** Output a single backslash character
\t Output a **<tab>** character
\n Output a **<newline>** character
\0 Output a null string (i.e., no separator between output fields)

-s Output successive lines from each input *file* across the page, with each input line separated from the next by a **<tab>** character. After all input lines from a given file have been concatenated, terminate the output line with a **<newline>** character and repeat the process on the next input file.

Example

The following two files will be used for subsequent examples. **File1** contains:

```
File1_Line1
File1_Line2
File1_Line3
File1_Line4
```

File2 contains:

```
File2_Line1
File2_Line2
File2_Line3
File2_Line4
```

The command

```
paste File1 File2
```

generates the following output:

```
File1_Line1        File2_Line1
File1_Line2        File2_Line2
File1_Line3        File2_Line3
File1_Line4        File2_Line4
```

Adding the option **-s** yields the output:

```
File1_Line1        File1_Line2        File1_Line3        File1_Line4
File2_Line1        File2_Line2        File2_Line3        File2_Line4
```

See Also

awk, commands, cut, sed

Notes

patch — Command

Patch a variable or flag within the kernel
/conf/patch [-k] *image symbol=value ...*

The command **patch** alters the value of datum *symbol* to *value* in executable *image*. In general, you should use **patch** to alter configuration data (constants) in programs, in device drivers, and in the COHERENT kernel. For **patch** to work with a symbolic constant, *image* must have a symbol table that includes information about *symbol*. Therefore, executables that have been processed by the command **strip** cannot be **patch**ed.

Options

patch recognizes the following command-line options:

-k Patch *image*, and patch the kernel memory of the running COHERENT system via device **/dev/kmem**. Only the superuser **root** can use **patch** to access kernel memory.

-K Patch **/dev/kmem** only. Refer to *image* for its symbol table, but do not change it.

-p "Peek" — just display current values; change nothing.

-v Verbose — display values before and after patching.

Variable Names

symbol and *value* can be either a numeric constant or a symbol from the symbol table of *image*. *symbol* and *value* expressions can include a numeric offset. In addition, *value* can be composed of the construct **makedev(***major***,***minor***)**, where *major* and *minor* are the "major" and "minor" device numbers, respectively, resulting in a **dev_t**-sized device type. No spaces can appear around the equal sign in the **symbol=***constant* construct.

Numeric constants default to decimal, but may be specifyed with a leading **0** prefix to specify an octal number or a **0x** prefix to specify a hexadecimal number.

The size of the altered *symbol* field is, by default, **sizeof(int)**. **patch** recognizes the following explicit size overrides:

:c The size of the altered field is **sizeof(char)**.

:i The size of the altered field is **sizeof(int)**.

:l The size of the altered field is **sizeof(long)**.

:s The size of the altered field is **sizeof(short)**.

Example

The following example gives technique that allows kernel display — that is, the output of the routines **cmn_err()** and the kernel's version of **printf()** — to go to a serial port. With this, you can save the panic messages and register dumps on a terminal screen or printer page while you reboot and try to track down what went wrong. To do so, plug a terminal into a serial port, and then do the following.

1. Find the major and minor numbers of a working serial port. Do not configure the port for modem control or flow control; use something simple like **com2l**. Make sure you can send data out the port; for example see that the command

   ```
   date > /dev/com2l
   ```

   sends data to the terminal's screen. The baud rate for the port will be whatever is specified for the default in file **/etc/default/async** — 9600 unless you have changed it.

2. Make sure the port is *not* enabled.

3. Create a test kernel around that you can modify. Call it something easily remembered, such as **/testcoh**.

4. Patch the kernel with the command

   ```
   /conf/patch -v /testcoh condev=makedev(major,minor):s
   ```

   where *major* is the major number for the serial port, and *minor* is its minor number.

5. Boot the patched kernel.

With this change, you will not be able to control kernel output with XON and XOFF, nor will you seen kernel output from very early startup (before the page tables are working) appear on the serial device.

Example

The following example patches the kernel to redirect error messages to a terminal device on a serial port, instead of displaying them on the console:

```
/conf/patch -v /Ikernel_name "condev=makedev(maj, min):s"
```

where *kernel_name* names the kernel you wish to patch, and *maj* and *min* are, respectively, the major and minor device numbers of the serial port to which you wish to redirect messages.

Note that **condev** is a short integer, so the ":s" is essential. The patch is made to the file on disk. You must reboot before it can work — chaos results if you try to switch console devices in a running kernel.

See Also

commands, device drivers, kernel

Notes

It is extremely dangerous to patch the COHERENT kernel. Almost all changes that you may wish to make the kernel can be accomplished more safely by using the commands **idtune** and **idmkcoh**. For details on how to use the commands, see their entries in the Lexicon. Therefore, do not use **/conf/patch** to patch the kernel unless you know *exactly* what you are doing. *Caveat utilitor!*

LEXICON

Beginning with release 4.2 of COHERENT, the symbol table has been removed from the kernel, and is kept in its own file. The symbol-table file is named after its corresponding kernel; for example, the symbol table for a kernel named **/coherent** is kept in file **/coherent.sym**. This complicates using **patch** to hot-patch a kernel. As noted above, you are well advised to use commands **idtune** and **idenable** to modify your kernel configuration, than using **patch** to hot-patch an existing kernel.

PATH — Environmental Variable

Directories that hold executable files

PATH names a default set of directories that are searched by COHERENT when it seeks an executable file. You can set **PATH** with the command **PATH**. For example, typing

```
PATH=/bin:/usr/bin
```

tells COHERENT to search for executable files first in **/bin**, and then in **/usr/bin**. Note the use of the colon ':' to separate directory names.

See Also

environmental variables, path.h

path() — General Function (libc)

Path name for a file
#include <path.h>
#include <stdio.h>
char ***path(***_path_**,** _filename_**,** _mode_**);**
char **_path_**,** ***_filename_**;*
int _mode_**;**

The function **path()** builds a path name for a file.

path points to the list of directories to be searched for the file. You can use the function **getenv()** to obtain the current definition of the environmental variable **PATH**, or use the default setting of **PATH** found in the header file **path.h**, or, you can define *path* by hand.

filename is the name of the file for which **path** is to search. *mode* is the mode in which you wish to access the file, as follows:

X_OK	Execute the file
W_OK	Write to the file
R_OK	Read the file

path() calls the function **access()** to check the access status of *filename*. If **path()** finds the file you requested and the file is available in the mode that you requested, it returns a pointer to a static area in which it has built the appropriate path name. It returns NULL if either *path* or *filename* are NULL, if the search failed, or if the requested file is not available in the correct mode.

Example

This example accepts a file name and a search mode. It then tries to find the file in one of the directories named in the **PATH** environmental variable.

```
#include <path.h>
#include <stdio.h>
#include <stdlib.h>

void
fatal(message)
char *message;
{
      fprintf(stderr, "%s\n", message);
      exit(1);
}
```

```
main(argc, argv)
int argc; char *argv[];
{
      char *env, *pathname;
      int mode;

      if (argc != 3)
            fatal("Usage: findpath filename mode");

      if(((mode=atoi(argv[2]))>4) || (mode==3) || (mode<1))
            fatal("modes: 1=execute, 2=write, 4=read");

      env = getenv("PATH");
      if ((pathname = path(env, argv[1], mode)) != NULL) {
            printf("PATH = %s\n", env);
            printf("pathname = %s\n", pathname);
            return;
      } else
            fatal("search failed");
}
```

See Also

access(), libc, PATH, path.h

path.h — Header File

Define/declare constants and functions used with PATH
#include <path.h>

path.h declares constants used to handle the environmental variable **PATH**. These include, among others, the default path, the path separator, and the list separator. **path.h** also declares the function **path()**.

See Also

header files, path(), PATH

pathalias — Command

Generate a set of paths among computers
/usr/lib/mail/pathalias [-ivcDf] [-d *link***] [-l** *host***] [-t** *link***] [** *datafile ...* **]**

The command **pathalias** computes the shortest path and corresponding route from a host to every other known, reachable host. It reads host-to-host connectivity information from the standard input or *datafile*, then writes a list of host-route pairs onto the standard output. This command normally is used only by administrators of busy systems, to maintain the path information used by **smail**.

pathalias recognizes the following command-line options:

-c Print costs: print the path cost before each host-route pair.

-D Terminal domains: see **domains** section, below.

-d *arg* *arg* is a dead link, host, or network. If *arg* is of the form

*host-1***!***host-2*

pathalias treats the link from *host-1* to *host-2* as an extremely high-cost (i.e., dead) link. If *arg* is a single host name, **pathalias** treats that host as dead and uses it on any path only as the relay host of last resort. If *arg* names a network, the network requires a gateway.

-f First-hop cost: the printed cost is the cost to the first relay in a path, instead of the cost of the entire path. This option implies (and overrides) option **-c**.

-i Ignore case: map all host names to lower case. By default, case is significant.

-l *host* Set the name of the local host to *host*. By default, **pathalias** reads file **/etc/uucpname** to discover the name of your system.

-t *arg* Output trace information for *arg* onto the standard error.

-v Verbose: report some statistics on the standard error output.

Input Format

A line that begins with white space continues the preceding line. **pathalias** ignores anything following a '#'.

A list of a host-to-host connection consists of a **from** host in column one, followed by white space, followed by a comma-separated list of **to** hosts, called *link*s. A link may be preceded or followed by a network character to use in the route. Valid network characters are '!' (default), '@', ':', and '%'. A link (and network character, if present) can be followed by a "cost" enclosed between parentheses.

The *cost* is an arithmetic expression that includes numbers, parentheses, and the operators '+', '-', '*', and '/'. It cannot be negative. **pathalias** recognizes the following symbolic costs:

LOCAL A local-area-network connection. Set cost to 25.

DEDICATED A high-speed dedicated link. Set cost to 95.

DIRECT A toll-free telephone call. Set cost to 200.

DEMAND A long-distance telephone call. Set cost to 300.

HOURLY An hourly poll. Set cost to 500.

EVENING A time-restricted telephone call. Set cost to 1,800.

DAILY A daily poll (also called **POLLED**). Set cost to 5,000.

WEEKLY An irregular poll. Set cost to 30,000.

In addition, the symbolic cost **DEAD** is a very large number (effectively, infinite); **HIGH** and **LOW** are -5 and +5, respectively, for baud-rate or quality bonuses/penalties; and **FAST** is -80, for adjusting costs of links that use high-speed modems (9600 baud or faster). These symbolic costs represent an imperfect measure of bandwidth, monetary cost, and frequency of connections. For most mail traffic, it is important to minimize the number of hosts in a route; for this reason, **HOURLY** times 24 is much larger than **DAILY**. If no cost is given, **pathalias** uses a default cost of 4,000.

For the most part, an arithmetic expression that mixes symbolic constants other than **HIGH**, **LOW**, and **FAST** makes no sense. For example, if a host calls a local neighbor whenever there is work, and in addition polls every evening, the cost is **DIRECT**, not **DIRECT+EVENING**.

Examples

Consider the following input:

```
down      princeton!(DEDICATED), tilt,
          %thrash(LOCAL)
princeton topaz!(DEMAND+LOW)
topaz     @rutgers(LOCAL+1)
```

If a link is encountered more than once, the least-cost occurrence dictates the cost and network character. **pathalias** treats links as bidirectional but asymmetric: for each link declared in the input, **pathalias** assumes a **DEAD** reverse link.

If the "to" host in a link is enclosed by angle brackets, **pathalias** regards the link as being terminal, and heavily penalizes all links beyond it. For example, when given the input

```
seismo    <research>(10), research(100), ihnp4(10)
research  allegra(10)
ihnp4     allegra(50)
```

pathalias generates a direct path from site **seismo** to site **research**; however, the path from **seismo** to **allegra** uses **ihnp4** as a relay, not **research**.

The set of names by which a host is known to its neighbors is called its *aliases*. Aliases are declared as follows:

name **=** *alias*, *alias ...*

name is the name by which the host is known to its predecessor in the route.

Fully connected networks, such as the Internet or a local-area network, are declared as follows:

net **=** {*host*, *host*, **...**}

The list of hosts may be preceded or followed by a routing character (by default, '!'), and may be followed by a cost (default 4,000). The network name is optional; if not given, **pathalias** makes one up. Consider the following input:

```
etherhosts = {rahway, milan, joliet}!(LOCAL)
ringhosts = @{gimli, alida, almo}(DEDICATED)
= {etherhosts, ringhosts}(0)
```

The routing character used in a route to a network member is the one encountered when "entering" the network. For details, see the sections on gateways and domains, below.

If you wish to give connection data, but also wish to hide the host names, use a declaration of the form:

private {*host*, *host*, **...**}

pathalias will not generate a route *to* a private host, but it may produce routes *through* it. The scope of a **private** declaration extends from the declaration either to the end of the input file in which it appears, or to a **private** declaration with an empty host list, whichever comes first. The latter scope rule lets you retain the semantics of a **private** declarations when you pass data to **pathalias** via the standard input.

Dead hosts, links, or networks may be presented in the input stream by declaring

dead {*arg*, **...**}

where *arg* has the same form as the argument to the command-line option **-d**.

To force a specific cost for a link, use

delete {*host-1***!***host-2*}

to delete all prior declarations, then re-declare the link as desired. To delete a host and all its links, use the instruction:

delete {*host*}

Diagnostic messages name the file in which **pathalias** found the error. To change the file's name, use the instruction:

file {*filename*}

You can fine-tune an entry by adjusting the weights of all links from a given host. For example:

adjust {*host-1*, *host-2***(LOW),** *host-3***(-1)**}

If no cost is given, **pathalias** uses a default of 4,000.

The following script pipes into **pathalias** input from compressed (and uncompressed) files:

```
for i in $*; do
      case $i in
            *.Z)  echo "file {‘expr $i : '.Z'‘}
                  zcat $i ;;
            *)    echo "file {$i}"
                  cat $i ;;
      esac
      echo "private {}"
done
```

Output Format

pathalias writes to the standard output a list of host-route pairs, where the route is a string appropriate for use with **printf()**, e.g.:

```
rutgers  princeton!topaz!%s@rutgers
```

%s in the route string is replaced by the name of the user to whom the message is being sent. This task normally is performed by a mailer, e.g., **mail** or **elm**.

Except for domains, the name of a network is never used in routes. Thus, in the earlier example, the path from down to up would be **up!%s**, not **princeton-ethernet!up!%s**.

LEXICON

Gateways

pathalias represents a network by a pseudo-host and a set of network members. Links from the members to the network have the weight given in the input, whereas the cost from the network to its members is zero. If a network is declared dead, the member-to-network links are marked dead, which effectively prohibits access to the network from its members.

If, however, the input also shows an explicit link from any host to the network, then that host can be used as a gateway. In particular, the gateway need not be a network member. For example, if CSNET is declared dead and the input contains

```
CSNET = {...}
csnet-relay CSNET
```

then routes to CSNET hosts will use **csnet-relay** as a gateway.

Domains

A network whose name begins with '.' is called a *domain*. Domains are assumed to require gateways, i.e., they are **DEAD**. The route given by a path through a domain is similar to that for a network, but here the domain name is tacked onto the end of the next host. Subdomains are permitted. For example, the definition

```
harvard  .EDU   # harvard is gateway to .EDU domain
.EDU   = {.BERKELEY, .UMICH}
.BERKELEY = {ernie}
```

yields:

```
ernie   ...!harvard!ernie.BERKELEY.EDU!%s
```

Output is given for the nearest gateway to a domain. For example, the example above yields:

```
.EDU   ...!harvard!%s
```

Output is given for a subdomain if it has a different route than its parent domain, or if all its ancestor domains are private.

If the you use its command-line option **-D**, **pathalias** treats a link from a domain to a host member of that domain as terminal. This property extends to host members of subdomains, etc., and discourages routes that use any domain member as a relay.

Files

/usr/local/lib/palias.dir — Default output
/usr/local/lib/palias.pag — Default output
comp.mail.maps — Likely location of some input files

See Also

commands, mail [overview], pathmerge, smail
Honeyman P., Bellovin, S.M.: PATHALIAS, or the care and feeding of relative addresses. Atlanta, *Proceedings of the Summer USENIX Conference*, 1986.

Notes

This command is not used by the implementation of **smail** that COHERENT uses. It is included, however, for compatibility with other implementations.

The order of arguments is significant. In particular, options **-i** and **-t** should appear early.

pathconf() — System Call (libc)

Get a file variable by path name
#include <unistd.h>
long pathconf(*path***,** *name***)**
const char ******path***; int** *name***;**

pathconf() returns the value of a limit or option associated with the file *path*. *name* is a symbolic constant (defined in **<unistd.h>**) that represents the limit or option to be returned. The value that **pathconf()** returns depends upon the type of file that *path* names.

pathconf() can return information about the following constants:

_PC_LINK_MAX
The maximum value of a file's link count. If *path* names a directory, the value returned applies to the directory itself.

_PC_MAX_CANON
The number of bytes in a terminal's canonical input queue. Behavior is undefined if *path* does not name a terminal file.

_PC_MAX_INPUT
The number of bytes for which space will be available in a terminal's input queue. Behavior is undefined if *path* does not name a terminal file.

_PC_NAME_MAX
The number of bytes in a file name. The behavior is refined if *path* does not name a directory. The value returned applies to the file names within the directory.

_PC_PATH_MAX
The number of bytes in a path name. Behavior is undefined if *path* does not refer to a directory. If *path* names the current working directory, **pathconf()** returns the maximum length of a relative path name.

_PC_PIPE_BUF
The number of bytes that can be written atomically when writing to a pipe. If *path* names a pipe or FIFO, the value returned applies to the FIFO itself. If *path* names a directory, the value returned applies to any FIFOs that exist or can be created within that directory. If *path* names any other type of file, behavior is undefined.

_PC_CHOWN_RESTRICTED
chown() can be used only by a process with appropriate privileges, and only to change the group ID of a file to either that process's effective group ID or one of its supplementary group IDs. If *path* names a directory, the value returned applies to any file, other than a directory, that exists or can be created within the directory.

_PC_NO_TRUNC
Path-name components longer than **NAME_MAX** generate an error. The behavior isundefined if *path* does not refer to a directory. The value returned applies to the file names within the directory.

_PC_VDISABLE
If this value is defined, terminal-special characters can be disabled. Behavior is undefined if *path* does not name a terminal file.

The value of the system limit or option that *name* specifies does not change during the lifetime of the calling process.

pathconf() fails and returns -1 if *name* is not set to a recognized constant. It fails, returns -1, and sets **errno** to an appropriate value if any of the following is true:

- The process that calls **pathconf()** lacks permission to search a directory named in *path*. **pathconf()** sets **errno** to **EACCES**.
- *path* is needed for the command specified and it either points to an empty string or names a file that does not exist. **pathconf()** sets **errno** to **ENOENT**.
- A component of *path*'s prefix is not a directory. **pathconf()** sets **errno** to **ENOTDIR**.
- *name* is an invalid value. **pathconf()** sets **errno** to **EINVAL**.

See Also

fpathconf(), libc
POSIX Standard, §5.7.1

pathmerge — Command

Merge sorted paths files
/usr/lib/mail/pathmerge *file ...*

pathmerge reads the sorted path *file*s, each of which was generated by command **pathalias**, merges the path information they contain, and writes the result onto the standard output. It normally is used only by administrators of busy systems, to maintain the path information used by **smail**.

In its output, **pathalias** writes one path given for each host name. It gives precedence in paths to the *file*s that appear earlier in the argument list. The file name '-' represents the standard input; this lets you mingle input from files with input from the standard input.

As an example of the use of **pathmerge**, consider two files, **forces** and **paths**, whose contents, respectively, are

```
ihnp4    cbosgd!ihnp4!%s
muts12   muts12!%s
sun      sun!%s
```

and:

```
cbosgd   cbosgd!%s
ihnp4    ihnp4!%s
sun      ames!sun!%s
uunet    uunet!%s
```

The command

```
pathmerge forces paths
```

writes the following onto the standard output:

```
cbosgd   cbosgd!%s
ihnp4    cbosgd!ihnp4!%s
muts12   muts12!%s
sun      sun!%s
uunet    uunet!%s
```

For the purposes of **pathmerge**, a host name is terminated a by a space, a tab, a colon, or a newline. The number of *file*s that you can pass to **pathmerge** is limited by the number of available file descriptors, as all of the files are opened and read simultaneously.

See Also

commands, mail [overview], mkline, mkpath, mksort, mkdbm, pathalias, smail

Notes

This command is not used by the implementation of **smail** that COHERENT uses. It is included, however, for compatibility with other implementations.

paths — System Administration

Routing data base for mail
/usr/lib/mail/paths

File **/usr/lib/mail/paths** holds the data base that the command **smail** uses to route mail.

Each line gives routing information to a host, and has the following format:

```
host  route [cost]
```

host names a remote host. The *route* field details the route by which mail can travel from your system to *host*. Note that it includes the **printf()**-style format string "%s". This field uses the bang-path format for describing a route. For example, if you access site **foo** via site **bar**, then route field for site **bar** reads:

```
bar   foo!bar!%s
```

smail uses the optional field *cost* to decide whether to queue mail that is spooled for other systems, or to invoke the command **uucico** to deliver the mail immediately. If the *cost* is at or below **smail**'s "queueing threshold", then **smail** attempts to deliver it immediately. This speeds mail delivery between hosts that enjoy an inexpensive UUCP link, such as a serial line; and batches mail that must be sent over expensive media, such as long-distance telephone. If the *cost* field is absent, **smail** gives this host a cost value above that of its queueing threshold.

Note that the value in the *costs* field does not override the connection times set in the UUCP file **sys**. Thus, this field is useful only for systems that you can call any time, or that you call frequently.

Example

The following gives a sample **paths** file for a COHERENT system named **lepanto**:

```
friend            friend!%s          300
hubsys            hubsys!%s          95
lepanto           %s                 0
lepanto.ampr.org  %s                 0
widget            hubsys!widget!%s   95
```

As this file shows, **lepanto** is linked to systems **hubsys** and **friend**. The cost of 95 associated with **hubsys** is low, and is appropriate to a low-cost link, such as a hard-wired link. On the other hand, the cost of 300 associated with **friend** is high, which indicates that the connection with **friend** is expensive, such as a long-distance telephone connection. If cost is 100 or greater, mail will be queued for later delivery. A cost below 100 tells **smail** to attempt immediate delivery.

In this example, machine **lepanto** is registered in the domain **ampr.org**. "ampr" is an abbreviation for "Amateur Packet Radio," which indicates that **lepanto** is a packet-radio node. Note that machine name **lepanto** appears in both conventional form ("lepanto") and domain form ("lepanto.ampr.org"); this is done to make it easier for **smail** to resolve addresses.

lepanto can use **hubsys** to forward mail to **widget**. Thus, when **smail** receives mail for system **widget**, it will transmit it to **hubsys** for forwarding. Note that **hubsys**'s administrator must have given **lepanto** permission to use it as a mail relay, or this will not work.

See Also

Administering COHERENT, mail [overview], smail

Notes

Please note that the mail-routing program **smail** does not actually read **/usr/lib/mail/paths** when it processes mail; rather, it reads a DBM-style data base that is built from this file. The DBM data base can be read much faster than an ordinary text file, thus improving the speed with which **smail** handles mail. Thus, when you edit **paths**, you must invoke the command **mkpaths** to "cook" its contents into DBM format, so **smail** see the changes you have made. For information on DBM-style data bases, see the Lexicon entry for **libgdbm**.

pattern — Definition

A **pattern** is any combination of text and wildcard characters that can be interpreted by a command. Patterns are also called "regular expressions".

The function **pnmatch()** compares two patterns and indicates whether they match.

For a fuller explanation of how to use patterns within applications, see the section on *Expert Editing* in the tutorial for the line editor **ed**.

See Also

egrep, pnmatch(), Programming COHERENT, wildcards

pause() — System Call (libc)

Wait for signal
#include <unistd.h>
int pause()

pause() suspends execution until the process receives a signal. The awaited signal could come from **kill()**, **alarm()**, or the controlling terminal.

See Also

alarm(), kill(), libc, signal(), sleep(), unistd.h
POSIX Standard, §3.4.2

pclfont — Command

Prepare a PCL font for downloading via MLP
pclfont [-f *n***]** *font* **[...** *font***]**

The command **pclfont** prepares each *font* for downloading via the MLP spooler to a printer that runs the Hewlett-Packard Page Control Language (PCL). *font* must give the full path name of a PCL bitmapped "soft font". **pclfont** brackets each *font* with the PCL commands that tell the printer to load the font into a given "slot" in its memory,

and to let the font reside permanently in memory, then writes the altered *font* to the standard output.

The option **-f** names the slot into which you want to load *font*. If the command line names more than *font*, **pclfont** sequentially assigns slots beginning with slot *n*. If you do not use the option **-f**, **pclfont** assigns slots beginning with slot 1.

The processed fonts can either be piped to the command **lp** or redirected into a file for later downloading. When downloaded via **lp**, you must use the MLP device **hpfont**. For example, to download fonts **tr240bpn.usp** and **op240bpn.usp** into slots 16 and 17 on your printer, use the command:

```
pclfont -f 16 tr240bpn.usp op240bpn.usp | lp -dhpfont
```

See Also

commands, lp, printer, troff

pclose() — STDIO Function (libc)

Close a pipe
#include <stdio.h>
int pclose(*fp***)**
FILE ******fp***;**

pclose() closes the pipe pointed to by *fp*, which must have been opened by the function **popen()**.

pclose() awaits the completion of the child process and performs other cleanup. It returns the value from a **WAIT** done on the child process. This value includes information in addition to the "simple" exit value of the child process.

Example

For an example of this function, see the Lexicon entry for **popen**.

Files

<stdio.h>

See Also

fclose(), fopen(), libc, pipe(), popen(), sh, system(), wait()

Diagnostics

pclose() returns -1 if *fp* had not been created by a call to **popen()**. Otherwise, **pclose()** returns the exit status of the *command*, in the format described in the entry for **wait()**: exit status in the high byte, signal information in the low byte.

perror() — General Function (libc)

System call error messages
#include <errno.h>
perror(*string***)**
char ******string***; extern int** *sys_nerr***; extern char** ******sys_errlist***[];**

perror() prints an error message on the standard error device. The message consists of the argument *string*, followed by a brief description of the last system call that failed. The external variable **errno** contains the last error number. Normally, *string* is the perror of the command that failed or a file perror.

The external array **sys_errlist** gives the list of messages used by **perror()**. The external **sys_nerr** gives the number of messages in the list.

See Also

errno, errno.h, libc
ANSI Standard, §7.9.10.4
POSIX Standard, §8.1

phone — Command

Print numbers and addresses from phone directory
phone *person ...*

The command **phone** searches a number of telephone directory files for each *person* argument that is given. Any lines that matches any of the *person* arguments is printed. Typically, such lines contain the telephone number, name, and address of a person or organization. Lower-case letters in *person* can be matched by both the same letter and the corresponding upper-case letter in the phone directory.

The user may supply his own phone directory by setting the (exported) shell variable **PHONEBOOK**, to the name of that file. If given, this file is searched first. Then, the system-wide phone book is always searched.

Files

$PHONEBOOK — User-supplied phonebook (searched first)
/usr/pub/phonebook — System-wide phone directory

See Also

commands

Diagnostics

phone exits with non-zero status if a call fails. A diagnostic message is written to **stderr** if no matching entries are found.

The standard phonebook shipped with COHERENT includes telephone numbers and descriptions of third-party vendors who sell software for COHERENT. If you're looking for software to run under COHERENT, check there first.

pipe — Definition

A *pipe* directs the output stream of one program into the input stream of another program, thus coupling the programs together. With pipes, two or more programs (or *filters*) can be coupled together to perform complex transforms on streams of data. For example, in the following command

```
cat DATAFILE1 DATAFILE2 | sort | uniq -d
```

the filter **cat** opens two files and prints their contents. Its output is piped to the filter **sort**, which sorts it. The output of **sort** is piped, in turn, to the filter **uniq**, which (with the **-d** option) prints a single copy of each line that is duplicated within the file. Thus, with this simple set of commands and pipes, a user can quickly print a list of all lines that appear in both files.

See Also

filter, mkfifo(), named pipe, pipe(), Using COHERENT

pipe() — System Call (libc)

Open a pipe
#include <unistd.h>
int pipe(*fd***)**
int *fd***[2];**

A *pipe* is an interprocess communication mechanism. **pipe()** creates a pipe, typically to construct pipelines in the shell **sh**.

pipe() fills in *fd***[0]** and *fd***[1]** with *read* and *write* file descriptors, respectively. The file descriptors allow the transfer of data from one or more writers to one or more readers. Pipes are buffered to 5,120 bytes. If more than 5,120 bytes are written into the pipe, the **write()** call will not return until the reader has removed sufficient data for the **write()** to complete. If a **read()** occurs on an empty pipe, its completion awaits the writing of data.

When all writing processe close their write file descriptors, the reader receives an end of file indication. A write on a pipe with no remaining readers generates a **SIGPIPE** signal to the caller.

pipe() is generally called just before **fork()**. Once the parent and child processes are created, the unused file descriptors should be closed in each process.

Example

The following example prints the word **Waiting** until a line of data is entered. It illustrates how to use **pipe()**, **fstat()**, and **fork()**.

```
#include <stdio.h>
#include <sys/stat.h>         /* for stat */
#include <sgtty.h>            /* for stty/gtty functions */
#include <unistd.h>

static int fd[2];       /* pipe array */

main()
{
      printf("This prints 'Waiting' every second until a 'q' is hit.\n");

      /*
       * Pipe may also be constructed by /etc/mknod
       * If it is desired to have tasks communicate where
       * they are not parent and child. In this case make
       * sure the constructed pipe has the correct owner and
       * permissions. Such pipe may be used exactly like this
       * but open()ed on each side.
       */

      if (-1 == pipe(fd)) {
            fprintf(stderr, "Cannot open pipe\n");
            exit(EXIT_FAILURE);
      }

      if (fork())
            parentProcess();
      else
            childProcess();
      exit(EXIT_SUCCESS);
}

parentProcess()
{
      struct stat s;
      char buff;

      for (buff = ' '; 'q' != buff;) {
            fstat(fd[0], &s); /* get status of pipe */
            if (s.st_size) {        /* char in the pipe */
                  read(fd[0], &buff, sizeof(buff));
                  printf("Got a '%c'\n", buff);
                  continue;
            }

            /*
             * This can be any process, it can use system()
             * or exec()
             */
            printf("Waiting\n");
            sleep(1);
      }
}

childProcess()
{
      struct      sgttyb      os, ns;
      char buff;

      gtty(fileno(stdin), &os);     /* save old state */
      ns = os;                /* get base of new state */
      ns.sg_flags |= RAW;           /* process each character as entered */
      ns.sg_flags &= ~(ECHO|CRMOD); /* no echo for now... */
      stty(fileno(stdin), &ns);     /* set mode */

      do {
            buff = getchar(); /* wait for the keyboard */
            write(fd[1], &buff, sizeof(buff));
      } while ('q' != buff);
```

```
      stty(fileno(stdin), &os);     /* reset mode */
}
```

See Also

close(), libc, libsocket, mkfifo(), mknod(), read(), sh, signal(), unistd.h, write()
POSIX Standard, §6.1.1

Diagnostics

pipe() returns zero on successful calls, or -1 if it could not create the pipe.

If it is necessary to create a pipe between tasks that are not parent and child, use **/etc/mknod** to create a named pipe. These named pipes can be opened and used by different programs for communication. Remember to give them the correct owner and permissions.

If you attempt to open a pipe write only, **O_NDELAY** is set, and there are currently no readers on this pipe, **open()** returns immediately and sets **errno** to **ENXIO**.

pnmatch() — String Function (libc)

Match string pattern
int pnmatch(*string***,** *pattern***,** *flag***)**
char **string***, ***pattern***; int** *flag***;**

pnmatch() matches *string* with *pattern*, which is a regular expression. The shell **sh** uses patterns for file name expansion and **case** statement expressions.

pnmatch() returns one if *pattern* matches *string*, and zero if it does not. Each character in *pattern* must exactly match a character in *string*; however, the wildcards '*', '?', '[' and ']', and '[!' and ']' can be used in *pattern* to expand the range of matching.

flag must be either zero or one: zero means that *pattern* must match *string* exactly, whereas one means that *pattern* can match any part of *string*. In the latter case, the wildcards '' and '$' can also be used in *pattern*.

Example

For an example of this function, see the entry for **fgets()**.

See Also

egrep, grep, libc, sh, string.h, wildcards

Notes

flag must be zero or one for **pnmatch()** to yield predictable results.

pnmatch() is a more powerful version of the ANSI functions **strstr()** and **strcmp()**.

For an **egrep**-style version of **pnmatch()**, see the function **regexp()**. It is described in the Lexicon article **libmisc**.

pointer — C Language

A *pointer* is an object whose value is the address of another object. The name "pointer" derives from the fact that its contents "point to" another object. A pointer may point to any type, complete or incomplete, including another pointer. It may also point to a function, or to nowhere.

The term *pointer type* refers to the object of a pointer. The object to which a pointer points is called the *referenced type*. For example, an **int *** ("pointer to **int**") is a pointer type; the referenced type is **int**. Constructing a pointer type from a referenced type is called *pointer type derivation*.

The Null Pointer

A pointer that points to nowhere is a *null pointer*. The macro **NULL**, which is defined in the header **stdio.h**, defines the null pointer. The null pointer is an integer constant with the value zero. It compares unequal to a pointer to any object or function.

Declaring a Pointer

To declare a pointer, use the indirection operator '*'. For example, the declaration

```
int *pointer;
```

declares that the variable **pointer** holds the address of an **int**-length object. Likewise, the declaration

```
int **pointer;
```

declares that **pointer** holds the address of a pointer whose contents, in turn, point to an **int**-length object.

Failure to declare a function that returns a pointer will result in that function being implicitly declared as an **int**. This does not cause an error on microprocessors in which an **int** and a pointer have the same size; however, if you transport this code to a microprocessor in which an **int** consists of 16 bits and a pointer consists of 32 bits, the pointer will be truncated truncated to 16 bits and the program probably will fail.

C allows pointers and integers to be compared or converted to each other without restriction. The COHERENT C compiler flags such conversions with the strict message

```
integer pointer pun
```

and comparisons with the strict message

```
integer pointer comparison
```

These problems should be corrected if you want your code to be portable to other computing environments.

See **C language** for more information.

Wild Pointers

Pointers are omnipresent in C. C also allows you to use a pointer to read or write the object to which the pointer points; this is called *pointer dereferencing*. Because a pointer can point to any place within memory, it is possible to write C code that generates unpredictable results, corrupts itself, or even obliterates the operating system if running in unprotected mode. A pointer that aims where it ought not is called a *wild pointer*.

When a program declares a pointer, space is set aside in memory for it. However, this space has not yet been filled with the address of an object. To fill a pointer with the address of the object you wish to access is called *initializing* it. A wild pointer, as often as not, is one that is not properly initialized.

Normally, to initialize a pointer means to fill it with a meaningful address. For example, the following initializes a pointer:

```
int number;
int *pointer;
   . . .
pointer = &number;
```

The address operator '&' specifies that you want the address of an object rather than its contents. Thus, **pointer** is filled with the address of **number**, and it can now be used to access the contents of **number**.

The initialization of a string is somewhat different than the initialization of a pointer to an integer object. For example,

```
char *string = "This is a string."
```

declares that **string** is a pointer to a **char**. It then stores the string literal **This is a string** in memory and fills **string** with the address of its first character. **string** can then be passed to functions to access the string, or you can step through the string by incrementing **string** until its contents point to the null character at the end of the string.

Another way to initialize a pointer is to fill it with a value returned by a function that returns a pointer. For example, the code

```
extern char *malloc(size_t variable);
char *example;
   . . .
example = malloc(50);
```

uses the function **malloc** to allocate 50 bytes of dynamic memory and then initializes **example** to the address that **malloc** returns.

Reading What a Pointer Points To

The indirection operator '*' can be used to read the object to which a pointer points. For example,

```
int number;
int *pointer;
   . . .
pointer = &number;
   . . .
printf("%d\n", *pointer);
```

uses **pointer** to access the contents of **number**.

When a pointer points to a structure, the elements within the structure can be read by using the structure offset operator '->'. See the entry for **operators** for more information.

Pointers to Functions

A pointer can also contain the address of a function. For example,

```
char *(*example)();
```

declares **example** to be a pointer to a function that returns a pointer to a **char**.

This declaration is quite different from:

```
char **different();
```

The latter declares that **different** is a function that returns a pointer to a pointer to a **char**.

The following demonstrates how to call a function via a pointer:

```
(*example)(arg1, arg2);
```

Here, the '*' takes the contents of the pointer, which in this case is the address of the function, and uses that address to pass to a function its list of arguments.

A pointer to a function can be passed as an argument to another function. The functions **bsearch** and **qsort** each take a function pointer as an argument. A program may also use arrays of pointers to functions.

void *

void * is the generic pointer; it replaces **char** * in that role. A pointer may be cast to **void** * and then back to its original type without any change in its value. **void** * is also aligned for any type in the execution environment. Please note that COHERENT's C compiler does not yet recognize the type **void** *.

In Kernighan and Ritchie C, character pointers are equivalent to **void** *. To convert a program to use **void** *, rewrite the sources so that instances of

```
char *foo(bar);
```

is replaced by:

```
VOID_T *foo(bar);
```

Be sure that you do not replace legitimate **char** *s — that is, pointers that actually point to character strings. Then put the code

```
#if defined(__ANSI__) || defined(__GNUC__)
typedef void VOID_T
#else
typedef char VOID_T
#endif
```

into an application-owned header file that is included by every source file.

Pointer Conversion

One type of pointer may be converted, or *cast*, to another. For example, a pointer to a **char** may be cast to a pointer to an **int**, and vice versa.

The ANSI Standard states that any pointer can be cast to type **void** * and back again without its value being affected in any way. (Once again, please note that COHERENT's C compiler does not yet recognize the type **void** *.) Likewise, any pointer of a scalar type may be cast to its corresponding **const** or **volatile** version. The qualified pointers are equivalent to their unqualified originals.

Pointers to different data types are compatible in expressions, but only if they are cast appropriately. Using them without casting produces a *pointer-type mismatch*. The translator should produce a diagnostic message when it

detects this condition.

Pointer Arithmetic

Arithmetic may be performed on all pointers to scalar types, i.e., pointers to **char**s or **int**. Pointer arithmetic is quite limited and consists of the following:

1. One pointer may be subtracted from another.
2. An **int** or a **long**, either variable or constant, may be added to a pointer or subtracted from it.
3. The operators **++** or **--** may be used to increment or decrement a pointer.

No other pointer arithmetic is permitted. No arithmetic can be performed on pointers to non-scalar objects, e.g., pointers to functions.

When an **int** or **long** is added to a pointer, it is first multiplied by the length of what the pointer is declared as pointing to. Thus, if a pointer to an **int** is incremented by two, it points down two more **int**s, not two more characters. The following program demonstrates this feature:

```
char *pc = "Welcome";
int array[5] = { 1, 2, 3, 4, 5 };
int  *pi = array;

main()
{
        pc += 2;        /* pc points to  'l' */
        pi += 2;        /* pi points to 3 */
}
```

See Also

C language data formats **operators, portability, Programming COHERENT**
ANSI Standard, §6.1.2.5, §6.2.2.1, §6.2.2.3, §6.3.2.2-3, §6.5.4.1

poll() — System Call (libc)

Query several I/O devices
#include <poll.h>
int poll(*fds***,** *nfds***,** *timeout***)**
struct pollfd *fds***[]; unsigned long** *nfds***; int** *timeout***;**

The COHERENT system call **poll()** polls one or more file streams for one or more polling conditions. *fds* gives the address of an array of **struct**s of type **pollfd**, which has the following structure:

```
struct pollfd {
      int   fd;   /* file descriptor */
      short events;     /* requested events */
      short revents;    /* returned events */
};
```

Field **fd** gives the file descriptor for a file stream, as returned by a call to **open()**, or **creat()**. Fields **events** and **revents** give, respectively, the polling conditions that interest you, and those that have occurred. The legal conditions, as defined in header file **poll.h**, are as follows:

POLLIN Input, or a non-priority or file-descriptor passing message, is available for reading. In **revents**, this bit is mutually exclusive with **POLLPRI**.

POLLPRI A priority message is available for reading. In **revents**, this bit is mutually exclusive with **POLLIN**.

POLLOUT Output may be performed; the output queue is not full.

POLLERR An error message has arrived. This field is used only in **revents**, and is ignored in **events**.

POLLHUP A hangup has occurred. This field is used only in **revents**, and is ignored in **events**.

POLLNVAL The specified **fd** value does not belong to an open I/O stream. This field is used only in **revents**, and is ignored in **events**.

nfds gives the number of entries in *fds*.

For each array element *fds***[***i***]**, **poll()** examines the file descriptor *fds***[***i***].fd** for the events specified by bits set in *fds***[***i***].events**, and places the resulting status into *fds***[***i***].revents**. If the **fd** value is less than zero, **revents** for that

entry is set to zero. Event flags **POLLIN**, **POLLPRI**, and **POLLOUT** are set in **revents** only if the same bits are set in **events** *and* the corresponding condition holds. Event flags **POLLHUP**, **POLLERR**, and **POLLNVAL** are always set in **revents** if the corresponding condition holds, regardless of the contents of **events**.

If none of the defined events for any of the file descriptors has occurred, **poll()** waits for *timeout* milliseconds. Because the system clock runs at 100 hertz, the value used for *timeout* is the next higher multiple of ten milliseconds. If *timeout* is zero, **poll()** returns immediately. If *timeout* is -1, **poll()** blocks until a requested event occurs or a signal interrupts the call.

poll() returns the number of file descriptors for which **revents** is nonzero. It returns zero if it timed out with no matching events. If the call failed, it returns -1 and sets **errno** to an appropriate value.

Example

For an example of using **poll()** to read a serial port, see the Lexicon entry for **ioctl().** The following example uses **poll()** to sleep for a fraction of a second.

```
#include <poll.h>
#include <sys/v_types.h>
#include <sys/times.h>

main()
{
      struct pollfd fds;
      int timeout;
      struct tms tmp;
      int before; /* time in millisec before poll() */
      int after; /* time in millisec after poll() */

      timeout = 270; /* sleep time is timeout * 10 millisec */

      fds.fd = -1; /* no file needed for sleeping */

      before = times(&tmp); /* Get time before poll */

      /* sleep not less than 0.270 sec */
      poll(&fds, 1, timeout);

      after = times(&tmp); /* Get time after poll */

      printf("%d\n", (after - before) * 1000 / CLK_TCK);
}
```

See Also

libc, poll.h

poll.h — Header File

Define structures/constants used with polling devices
#include <poll.h>

poll.h defines structures and constants used by routines that poll devices.

See Also

header files

popd — Command

Pop an item from the directory stack
popd [*item ...* **]**

The COHERENT shell **sh** maintains an internal "directory stack", which is a stack of names of directories. You can manipulate this stack should you, for any reason, wish to traverse a number of directories quickly and efficiently.

The command **popd** pops an item from the directory stack. If called without an argument, it pops the last item. Otherwise, it pops the given stack *item*s in the order requested, where each *item* is a positive integer and zero is the top of the stack.

See Also

commands, dirs, pushd, sh

popen() — STDIO Function (libc)

Open a pipe
#include <stdio.h>
FILE *popen(*command, how***)**
char **command, *how**;*

popen() opens a pipe. It resembles the function **fopen()**, except that the opened object is a command line to the shell **sh** rather than a file.

The caller can read the standard output of *command* when *how* is **r**, or write to the standard input of *command* when *how* is **w**. **popen()** returns a pointer to a **FILE** structure that may be read or written.

Example

This example is equivalent to the command

```
ls -l | mail me
```

where **me** is your login identifier.

```
#include <stdio.h>
main()
{
      FILE *ifp, *ofp;
      int c;

      if ((NULL == (ofp = popen("lmail me", "w"))) ||
          (NULL == (ifp = popen("ls -l",     "r")))) {
                fprintf(stderr, "cannot popen\n");
            exit(1);
      }

      while (EOF != (c = fgetc(ifp)))
            fputc(c, ofp);

      pclose(ifp);
      pclose(ofp);
}
```

Files

<stdio.h>

See Also

fclose(), fopen(), libc, pclose(), pipe(), sh, system(), wait()

Diagnostics

popen() returns NULL if the link to *command* could not be established.

port — System Administration

File that describes ports for UUCP
/usr/lib/uucp/port

File **/usr/lib/uucp/port** names and describes the serial ports that **uucico** and **cu** use to connect to remote systems.

port consists of a set of entries, one for each port. Entries should be separated from each other by one blank line. Each entry consists of one or more of the following commands:

port *port_name*
: Name the port being described. This command must appear first in every port's entry.

type *string*
: This command gives the type of port. It must appear immediately after the **port** command. *string* must be one of the following:

  direct The port directly accesses another, usually via a serial port.

modem
The port accesses a modem. This is the default.

pipe The connection is a pipe that runs through another program

stdin The connection runs through the standard input and standard output. Use this option when **uucico** is run as a login shell

tcp The port is a TCP port.

protocol *string*
List the protocols that can be used with this port. If **/usr/lib/uucp/sys** contains a list of protocols, that list takes precedence over the one set in **port**. We recommend that protocols be specified in the file **sys** intsead of here. For information on the available protocols, see the Lexicon article **sys**.

protocol-parameter *protocol parameter*
Set a *parameter* for the *protocol*. This command recognizes exactly the same arguments as its namesake in the system-configuration file **sys**. For information on how to use this command, see the Lexicon entry for

seven-bit true | false
If **true**, then this port (or the modem plugged into it) supports only seven-bit transfers; if false, then it supports both seven-bit and eight-bit protocols. **uucico** uses this command only during protocol negotiation, to force the selection of a protocol that works across a seven-bit link. It will not prevent eight-bit characters from being transmitted. The default is **false**.

Note that some devices use only seven bits to define a character, and reserve the eighth bit as a parity bit. It is not possible it is not possible to send eight-bit characters across such devices.

reliable true | false
This command is used only when your system negotiates with the remote system over what protocol to use. If set to **false**, it forces your system to accept only a protocol that works over a seven-bit (or unreliable) connection. If **true**, then an eight-bit protocol is acceptable. The default is false.

half-duplex true | false
If **true**, then this port supports only half-duplex communications, which forces **uucico** not to use a bidirectional protocol with this port. If it is **false**, then the port supports both half-duplex and full-duplex communications. The default is **false**. **sys**.

device *string*
This command names the device associated with the port. For example, the command

```
device /dev/com2l
```

names port **com2l** as the device used by this port. This command is used only with ports of types **modem** or **direct**.

baud *number*
speed *number*
Set the baud rate for this port. If an entry in file **/usr/lib/uucp/sys** specifies a speed but no port entry, **uucico** tries every entry in **port** that has a matching baud rate, in the order in which they appear, until it finds one that is unlocked. These commands are used only with ports of type **modem** or **direct**.

baud-range *low high*
speed-range *low high*
Set the range of speeds at which this port can be run. *low* gives the minimum speed, *high* the maximum. This command applies only to ports of type **modem**.

carrier true | false
If **true**, the port supports carrier; if **false**, the port does not. If a port does not support carrier, the carrier-detect signal will never be required on this port, regardless of what the modem chat script says. If a direct port supports carrier, the port will be set always to expect carrier.

This command applies only to ports of type **direct** or **modem**. The default for a **modem** port is true; but for a **direct** port is false.

hardflow true | false
If **true**, turn on hardware flow control for this port; otherwise, do not. The default is **true**. This command applies only to ports of type **direct** or **modem**.

LEXICON

dial-device *device*
Send dialing instructions to *device*, instead of the the normal port device. This applies only to ports of type **modem**.

dialer *string*
Names the dialer to use. Information about the dialer is read from file **/usr/lib/uucp/dial**. This applies only to ports of type **modem**.

dialer *string ...*
Execute a simple dialing script. This command can be used in situations where the dialing script is so simple that it would be cumbersome to embed it within a separate file. If the command **dialer** is used with only one argument (to name a dialing script), this command is ignored. This applies only to ports of type **modem**.

dialer-sequence *dialer phone_number ...*
Name pairs of dialers and telephone numbers. The telephone number is substituted for the escape sequences **\D** or **\T** in the *dialer* entry. In effect, this lets you name a sequence of chat scripts to use. At present, this command is the only way to use a chat script with a TCP port.

This command applies only to ports of type **modem** or **tcp**.

lockname *name*
Use *name* when locking this port. This applies only to ports of type **modem** or **direct**.

service *service_name*
Name the TCP port to use. If this names a service, then **uucico** looks the port for that service in file **/etc/services**. If it is a number, then **uucico** binds itself to that TCP port. If this command is not used, then **uucico** by default uses the well-known port 540. This command applies only to ports of type **tcp**.

command *command* **[** *arguments* **]**
If the port is of type **pipe**, name the command and its arguments with which **uucico** will be exchanging data. For example, if your system is on a network, then *command* could a form of the command **rlogin**, which would permit **uucico** to log into the remote system via the network.

Example

The following gives a sample entry for a port:

```
port MWCBBS
type modem
device /dev/com2l
baud 9600
dialer tbfast
```

The following describes each command in detail:

port This names the port being described in this entry, in this case **MWCBBS**.

type The type of port — in this case, a modem.

device The device used by this port. The device name usually matches the port name, but it does not have to.

baud The speed of the port, in this case 9600.

dialer The type of dialing device (i.e., modem) plugged into this port — in this case, the dialer named **tbfast**. This dialer is described in the file **/usr/lib/uucp/dial**. For information on how a dialer is described in its file, see the Lexicon entry for **dial**.

See Also

Administering COHERENT, dial, sys, UUCP

Notes

Only the superuser **root** can edit **/usr/lib/uucp/port**.

The file **port** supports many commands in addition to the ones described here. This article describes only those commands that might be used in typical UUCP connections. For more information, see the original Taylor UUCP documentation, which is in the archive **/usr/src/alien/uudoc.tar.Z**.

portability — Definition

Portability means that code can be recompiled and run under different computing environments without modification. Although true portability is an ideal that is difficult to realize, you can take a number of practical steps to ensure that your code is portable:

- Do not assume that an integer and a pointer have the same size. Remember that undeclared functions are assumed to return an **int**. If a function returns a pointer, declare it so.
- Do not write routines that depend on a particular order of code evaluation, particular byte ordering, or particular length of data types.
- Do not write routines that play tricks with a machine's "magic characters"; for example, writing a routine that depends on a file's ending with **<ctrl-Z>** instead of **EOF** ensures that that code can run only under operating systems that recognize this magic character.
- Always use manifest constants, such as **EOF**, and make full use of **#define** statements.
- Use header files to hold all machine-dependent declarations and definitions.
- Declare everything explicitly. In particular, be sure to declare functions as **void** if they do not return a value; this avoids unforeseen problems with undefined return values.
- Do not assume that integers and pointers have the same size or even the same kind of structure. Do not assume that pointers are all the same or can point anywhere. On the i8086, in SMALL model a pointer to a function addresses relative to the code segment, whereas a pointer to data addresses relative to the data segment. On some machines, character pointers are of a different size or structure than word pointers.
- The constant NULL is defined as being different from any valid pointer. Use it and nothing else for that purpose.
- Keep test scripts, preferably at the function level. That is, follow each function with an

  ```
  #ifdef TEST
  ```

  section that will exercise that function. Running these can rapidly isolate portability problems.
- Place plenty of

  ```
  #assert
  ```

  statements in your programs. These can often pick up portability problems.

See Also

header files, pointer, Programming COHERENT, void

POSIX Standard — Definition

The term "POSIX Standard" refers to the Portable Operating System Interface (POSIX) standard, published by the International Standards Organization (ISO) in 1990 as its standard 9945-1. It is based on standard 1003.1 published in 1988 by the Institute for Electrical and Electronics Engineers (IEEE).

The POSIX Standard is built upon the documentation for the UNIX Operating System published originally by AT&T Bell Laboratories (now by Unix Systems Laboratories). It defines a common set of guidelines to which UNIX and UNIX-like operating systems like COHERENT should adhere in order to ensure common functionality, and to maximize the portability of code from one operating system to another. The publication of the POSIX Standard is a long step towards maintaining the openness of the UNIX family of operating systems.

ANSI Standard, Programming COHERENT

pow() — Multiple-Precision Mathematics (libmp)

Raise multiple-precision integer to power
#include <mprec.h>
void pow(*a, b, m, c***)**
mint **a, *b, *m, *c;*

pow() sets the multiple-precision integer (or **mint**) pointed to by *c* to the value pointed to by *a* raised to the power of the value pointed to by *b*, reduced modulo of the value pointed to by *m*.

See Also

libmp

pow() — Mathematics Function (libm)

Compute a power of a number
#include <math.h>
double pow(*z***,** *x***)**
double *z***,** *x***;**

pow() returns *z* raised to the power of *x*, or *z*^x. If an overflow error occurs (that is, you attempt to compute a number that is too large to fit into a double-precision floating-point number), **pow()** returns a huge value and sets **errno** to **ERANGE**.

Example

For an example of this function, see the entry for **log10()**.

See Also

libm
ANSI Standard, §4.5.5.1
POSIX Standard, §8.1

pr — Command

Paginate and print files
pr [*options* **] [** *file …***]**

pr paginates each *file* and writes it onto the standard output. At the top of each page, **pr** writes a header that that gives today's date, the file's name, the number of the page, and the number of the line in the input file at which printing begins.

The file name '-' tells **pr** to read the standard input; this lets you mingle text from one or more files with text you type from the keyboard or pipe in from another program. **pr** also reads the standard input by default if its command line does not name a file.

pr recognizes the following command-line options:

+ *skip* Skip the first *skip* pages of each input file.

-*N* Print the text in *N* columns. This is used to print out material that was typed in one or more columns.

-h *header*
Use *header* in place of the text name in the title. If *header* is more than one word long, it must be enclosed in quotation marks.

-l*N* Set the page length to *N* lines (default, 66).

-m Print the texts simultaneously, in separate columns. Each text will be assigned an equal amount of width on the page, and any lines longer than that width will be truncated. You can use this to print several similar texts or listings simultaneously.

-n Number each line.

-s*c* Separate each column by the character *c*. You can separate columns with a letter of the alphabet, a period, or an asterisk. Normally, each column is left-justified in a fixed-width field.

-t Suppress the printing of the header on each page, and the header and footer space.

-w*N* Set the page width to *N* columns (default, 80). Text lines are truncated to fit the column width. The maximum width is 254 columns.

See Also

cat, commands, nroff, prps

Notes

pr generates normal ASCII text, suitable for displaying on your screen or printing with a dot-matrix printer. The command **prps** also paginates text, but its output is in the PostScript language, suitable for printing on a PostScript printer.

prep — Command

Produce a word list
prep [-dfp] [-i *ifile* **] [-o** *ofile* **] [** *file ...* **]**

The command **prep** prepares a word list that is useful for statistical processing from the textual data found in each input *file*. If no *file* is given, **prep** reads the standard input for text.

For the purposes of **prep**, a word consists of a string of alphabetic letters and apostrophes. Words are written, one per line, to the standard output. Hyphenated words are treated as two words. However, any word hyphenated between two lines is rejoined as one word.

prep recognizes the following options:

-d Print a sequence number (of words in the input text) before each output word.

-f Fold upper-case letters into lower case. This is sometimes useful for producing unique lists of words.

-i *ifile* Ignore words found in *ifile*. *ifile* has words one per line that are matched against each input word, independent of case.

-o *ofile* Print only words found in *ofile*. Only one of **-i** or **-o** may be specified.

-p In addition to printing words, also print each punctuation character (printable, non-numeric characters that separate words), one per line. These lines are not counted for **-d.**

See Also

commands, deroff, ksh, sh, sort, spell, typo, wc

Notes

What constitutes a *word* is different in **deroff**, **prep**, and **wc**.

print — Command

Echo text onto the standard output
print [-enru*n***] [***argument ...***]**

The command **print** is built into the Korn shell **ksh**. It echoes each *argument* onto the standard output. Arguments are separated from each other by whitespace, and the list of arguments is terminated by a newline character.

print recognizes and substitutes for the following C-style escape sequences:

\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Tab
\v	Vertical tab
\0*nnn*	*nnn* is the octal value of the desired character

print recognizes the following options:

-e Re-enable expansion of C escape sequences.

-n Suppress printing of a newline at the end of the list of arguments.

-r Suppress expansion of C escape sequences.

-u*n* Redirect output from the standard output to shell file descriptor *n*.

See Also

commands, echo, ksh

printer — Technical Information

How to attach and run a printer

A *printer* is the device that transfers text to paper. The COHERENT system includes a system for spooling a file to one or more printers. *Spooling* means that the file is copied into a special area and printed by a daemon. With a spooler, more than one user can send files to the same printer at the same time, yet the files will not collide.

COHERENT also includes commands to prepare text for printing a variety of printers. These include line printers (that is, dot-matrix printers), Epson-compatible printers, laser printers that use the PCL page-description language, and printers that use PostScript. With COHERENT, you can run prepare text into a variety of formats, and print the output on any number of printers plugged into either parallel or serial ports.

COHERENT has implemented spooling in two ways. Versions of COHERENT prior to release 4.2 control printing through a version of the Berkeley command **lpr**. COHERENT release 4.2 and subsequent releases also control printing through the MLP print spooler, which implements a version of the System-V command **lp** and related tools. These systems differ greatly; each set is discussed in its own section below.

Before we begin to describe printing, please note that one major source of confusion for users is the fact that the same names occur over and over again. For example, please do not confuse the parallel-port's device driver **lp** with the print-spooler command **lp** or with the device **/dev/lp**. COHERENT inherits much of this confusion from the UNIX operating system; but we will do our best to make these terms clear to you. *Caveat lector.*

Device Drivers

Both the **lpr** and **lp** spoolers work through COHERENT's device drivers for the serial and parallel ports. The following gives an overview of these drivers.

The driver **lp** manages parallel ports. The architecture of the PC permits your computer to have up to three parallel ports. Devices **/dev/lpt1**, **/dev/lpt2**, and **/dev/lpt3** control, respectively, parallel ports 1, 2, and 3 in cooked mode. For more information, see the Lexicon entry for the driver **lp**.

COHERENT uses the driver **asy** to manage all serial ports, whether COM ports or multi-port cards. For details, see its entry in the Lexicon.

Finding the Port

Both spooler systems require that you be able to identify a port when you plug a printer into it. This can be more difficult than it seems, largely because the labels on your system's port may not be reliable: those labels reflect what MS-DOS thinks the ports are, and that may not be accurate.

The following describes how to identify the port into which you have just plugged a printer. Note that these directions assume that you are printing to a parallel port; however, you can adapt them to serial ports as well, depending on the configuration of serial devices on your system.

1. Plug the printer into an unused port. Load paper into the printer and turn it on.
2. Log in as the superuser **root**.
3. **cd** to directory **/dev**.
4. Send some output to each parallel port. The output must be something that your printer can print. If your new printer is a line printer, type:

   ```
   cat /etc/uucpname | pr > lpt1
   ```

   If the printer is a laser printer that uses PCL, type:

   ```
   cat /etc/uucpname | hp > lpt1
   ```

   Or, if the printer is a PostScript printer, type:

   ```
   cat /etc/uucpname | prps > lpt1
   ```

   If text appears on your printer, then you have discovered the correct port. Jot down its name on a piece of paper, e.g., "lpt1". If nothing happens, try the command again for **lpt2** and **lpt3**, until you have found the correct port and noted its name.
5. Exit from superuser status.

The lpr Printing System

Versions of COHERENT prior to release 4.2 use a version of the Berkeley command **lpr** to control printing. Although this command can print text onto printers plugged into either serial or parallel ports, they are almost always used through parallel ports; therefore, the descriptions in this section assume that all printers are plugged into parallel ports.

To begin, **lpr** is actually a family of commands, as follows:

hpd	Daemon that prints files on the laser printer
hpr	Spool a file for printing on the laser printer
hpskip	Abort/restart printing a file on the laser printer
lpd	Daemon that prints files on the line printer
lpr	Spool a file for printing on the line printer
lpskip	Abort/restart printing a file on the line printer

Each command has its own entry in the Lexicon, which describes it in detail.

The commands **lpr** and **hpr** dispatch text to printers: **lpr** to the printer plugged into device **/dev/lp**, and **hpr** to the printer plugged into device **/dev/hp**. Each of these devices is actually a link to the correct parallel port — that is, to devices **/dev/lpt1**, **/dev/lpt2**, or **/dev/lpt3**, as described above. (For information on what a *link* is, see the Lexicon entry for the command **ln**). The fact that each command uses a "generic" device for its output makes it easy for you to dispatch files to the right device; however, it also means that you can have only one line printer and one laser printer plugged into your computer.

When you installed COHERENT, the installation program tried to link **/dev/lp** and **/dev/hp** for you automatically; however, you may need to set them yourself (say, because you have purchased a new printer).

To set these links correctly, first follow the directions given above to identify the port into which you have plugged the printer. Then, link that port to the device by which you will access the printer. If you are installing a line printer that you will access via the command **lpr**, then you must use the command **ln** to link the port to device **/dev/lp**; if, however, the printer is a laser printer that you will access via the command **hpr**, then you must link the port to device **/dev/hp**. For example, if you have plugged a line printer into port **lpt1**, then use the following commands:

```
ln -f lpt1 lp
ln -f rlpt1 rlp
```

(Please note that the last character in "lpt1" and "rlpt1" is the numeral one — not a lower-case el.) If, however, you have plugged a laser printer into port **lpt3**, then use the following commands:

```
ln -f lpt3 hp
ln -f rlpt3 rhp
```

After you have made the links, use the command **lpr** or **hpr** (whichever is applicable) to test whether you have set up the links correctly. If you have not, go through the above procedure again.

The following describes how to use the **lpr** family of commands to print to a variety of printers.

Dumb Printers

To print on a line printer, simply use the command **lpr**. This command performs some formatting on the file, and invokes the line-printer daemon **lpd** to spool the file for printing. For example, to print the name of your system, use the command:

```
cat /etc/uucpname | pr | lpr -B
```

The option **-B** suppresses the printing of a banner page.

You can also print the output of the text-formatting command **nroff** on a line printer, assuming that your line printer understands how to backspace. For example, the manual pages included with COHERENT were formatted with **nroff**. To print the text of this Lexicon entry on your line printer, type:

```
man printer | lpr -B
```

Epson-Compatible Printers

The command **epson** massages text into a form that uses some of the text-formatting features of the Epson MX-80 printer and clones thereof. It is especially to be used with text that has been formatted with **nroff**: it turns **nroff**'s character-backspace-character sequence into the Epson escape sequences for emphasized text and italics. **epson** writes its formatted output to the standard output, from which you can pipe it to a spooler or other program.

LEXICON

For example, to print this manual page on an Epson-compatible printer, type:

```
man printer | epson | lpr -B
```

Laser Printers with PCL

The Hewlett-Packard LaserJet, and its clones, use the Hewlett-Packard Control Language (HPCL) to control their behavior. Note that some laser printers, such as the Apple LaserWriter, use PostScript instead of HPCL; these printers are described below.

The command **hp** prepares files to be printed on a HPCL printer. (Please do not confuse this with the device **/dev/hp**.) You should use it to prepare simple text, such as program listings, for printing on your laser printer.

Like the command **epson**, **hp** massages the output of **nroff** into escape sequences used by a printer — in this case, escape sequences used by a printer that's running the Hewlett-Packard Page Control Language (PCL). For example, to print this manual page on your PCL printer, type:

```
man printer | hp | hpr -B
```

The command **hpr** spools files to be printed on a laser printer. It works like the command **lpr**, except that it includes a number of special features; for example, you can use it to download LaserJet "soft fonts" into your printer.

PostScript Printers

Some laser printers use PostScript instead of HPCL to control their behavior. These printers expect their input to a program written in the PostScript language; if you send them ordinary text, they simply hang. To print ordinary text on a PostScript printer use the command **prps**, which is a PostScript version of the COHERENT command **pr**. It paginates text, draws a box around the page, and prints a simple header at the top of each page. For example, to print this manual page on a PostScript printer, use the command:

```
man printer | prps | hpr -B
```

Note that to print on a PostScript printer, you must use the **-B** option to the command **hpr**. If you do not, **hpr** will attempt to print a banner page in ordinary text on your printer, and your printer will hang.

The lp Printing System

Versions of COHERENT beginnning with release 4.2 also include the MLP spooler, which is an implementation of the System-V **lp** family of printing commands (hereafter called **lp**).

lp is considerably more sophisticated than the **lpr** commands. It permits you to have multiple printers of the same type (instead of just one laser printer and one line printer, as under **lpr**), which can be plugged into serial or parallel ports. It supports prioritization of printing jobs (that is, you can give some users or some types of jobs higher priority than others), lets each user set a default printer for his jobs, allows users to reprint their jobs easily, and allows applications to customize their output to take advantage of special printer features. It even supports local printing — that is, it will format and print output onto a printer that is plugged into a terminal's auxiliary port.

lp's commands resemble those used by UNIX System V to control printing, so this system can work more easily with third-party applications. Note, however, that the MLP implementation of **lp** does differ in some important respects from the System-V original; therefore, users who have used **lp** under UNIX should pay close attention to the following descriptions.

lp consists of the following commands:

cancel	Cancel the printing of a job
chreq	Change priority, lifetime, or printer for a job
lp	Spool one or more files for printing
lpadmin	Administer the print-spooler system
lpsched	Print jobs spooled with command lp; turn on printer daemon
lpshut	Stop the printer daemon
lpstat	Give the status of printer or print request
pclfont	Prepare a PCL font for downloading via MLP
reprint	Reprint a spooled print job
route	Let a user change his default printer

Each of these commands is described in its own Lexicon entry.

lp uses the following directories:

/usr/spool/mlp/backend	This directory holds the programs and scripts used to manage printers.
/usr/spool/mlp/queue	This directory holds all print requests.
/usr/spool/mlp/route	This directory holds files that name each user's default printer.

lp's behavior is set by the contents of the following files:

/usr/spool/mlp/controls	This file holds **lp**'s configuration data base. This data base links a printer by name to the device through which it is accessed, and to the configuration script (if any) with which its input is massaged. For information on how to modify it, see the Lexicon entry for **controls**.
/usr/spool/mlp/log	This file logs **lp**'s activity.
/usr/spool/mlp/status	This file gives the status of each defined printer.

To use **lp**, you must first use the command **lpadmin** to build a description file for each class of printer that you have plugged into your system. The description file names the class of printer (e.g., "epson" or "laserjet") and gives the information **lp** needs to manipulate input to the printer. For example, a script may include a **stty** command to set the port into a special mode, and one or more commands for filtering the input so it will print properly. A backend script can invoke commands like **prps** or **epson** to process text for printing. **lp** can perform sophisticated filtration; for example, it can correctly handle PostScript code that prints images or bar codes. See the Lexicon entry for **lpadmin** for more details on these scripts.

You must then use **lpadmin** to link a given printer, by name, to the device through which it is accessed. You must have first identified the port into which each printer is plugged, as described above. These links are stored in file **/usr/spool/mlp/controls**. If you have prepared a configuration script for this printer's type, then you must link it to the given printer as well. For example, if you have prepared a configuration script for all PostScript printers and named it **postscript**, then you must link that script to every PostScript printer whose input you want to be massaged in this manner. Unlike the **lpr** printing system, **lp** lets you attach to your computer more than one printer of each type.

One last point: each "printer" should identify a given physical device plus a given means of accessing it. Thus, one physical printer can have more than one name if you plan to access it in more than one manner. See the Lexicon entry for **lpadmin** for more information on this topic.

Note that if a printer is a "local printer" — that is, a printer plugged into the auxiliary port of the terminal that the user is using, the **termcap** description for that terminal must define the variables **PS** (print start) and **PN** (print end). Each printer's description file is stored in directory **/usr/spool/mlp/backend**.

You can use the command **route** to assign a default printer to each user. If the user has set a default printer for himself and if he does not name a printer on the **lp** command line, the output goes to that default printer. If the user has *not* set a default printer for himself and does not name a printer on his **lp** command line, the output goes to the system's default printer. This feature is an extension to the version of **lp** that is implemented by UNIX System V.

To spool a job for printing, use the command **lp**. A *job* consists either of one or more files, or of text read from the standard input. **lp** prefaces the job with a header that describes where and how the job is to be printed, then copies it into directory **/usr/spool/mlp/queue**. The name that **lp** gives the spooled job reflects its status, that is, the order in which it should be printed relative to other jobs that user has spooled. This allows each user to give a priority to the jobs that he has spooled.

Each job resides in the spooling directory until the printer daemon **lpsched** reads it and prints it. **lpsched** selects jobs for printing based on their relative priority, as shown in their names. It finds where the job is to be printed by reading its header; then it opens the description file for that printer and follows its directions for printing the job. To turn on the daemon, use the command **lpsched** by itself; to turn it off, use the command **lpshut**. If the spooler is shut down, jobs remain in **/usr/spool/mlp/queue** until you reawaken the daemon by issuing the command **lpsched**.

To see what files are being printed where, use the command **lpstat**. To cancel a printing request, use the command **cancel**.

A job remains "alive" in **/usr/spool/mlp/queue** until its "life" has expired; the life is set in its header. There are three types of "lifetime": *temporary*, in which a job survives two hours from the time of spooling; *short-term*, in which a job survives 48 hours; and *long-term*, in which a job survives 72 hours. The default is short-term. When a

job's life expires, **lpsched** removes it. A user can use the command **chreq** to change a job's lifetime or priority; or redirect it from one printer to another. While a job lives in the spool directory, a user can use the command **reprint** to reprint it. He can also use the command **route** to change his default printer.

Note that you should be *very* careful that jobs that include sensitive information — e.g., the payroll checks or your resume — do not linger in spool directory, where other users can reprint them. For information on resetting a job's lifetime, see the Lexicon entries for **chreq** and **MLP_LIFE**. You can change the default definitions of temporary, short-term, and long-term by editing **controls**. See its entry in the Lexicon for more information. *Caveat utilitor!*

The following environmental variables affect **lp**'s default behavior:

MLP_COPIES	The number of copies to print.
MLP_FORMLEN	The number of lines on the page to be printed.
MLP_LIFE	The "lifespan" of a spooled file.
MLP_PRIORITY	The default priority to give each spooled file.
MLP_SPOOL	Set a number of user-specific variable, such as title of document, type of document, and data base.

These variables can be set either by a user, or embedded in a script. Each is detailed in its own Lexicon entry.

See Also

Administering COHERENT, hpr, lp, lp [device driver], lpr, lpsched

Notes

When you link **/dev/lp** or **/dev/hp** to a device, it normally is linked to a "cooked" device, e.g., **/dev/hp**. This works correctly for character-based output, such as text (or PostScript files); however, if you are downloading binary data to the printer, such as graphics or fonts, be sure to use the "raw" device, e.g., **/dev/rhp**. Passing binary information through a "cooked" device will garble the data and distort the resulting image.

Some COHERENT 4.2 customers have experienced printing problems, including no printing, slow printing, or printing stops after a line or two. To fix this, one needs to do the following steps in exact order;

1. Edit file **/etc/conf/install_conf/keeplist**.
2. Change the last line so that it reads as follows:

   ```
   echo '-I SHMMNI:SEMMNI:NMSQID:LPWAIT:LPTIME:LPTEST'
   ```

3. Type the following command to build a new COHERENT kernel:

   ```
   /etc/conf/bin/idmkcoh -o /testcoh
   ```

4. Shutdown and reboot with the new kernel.
5. Log in as the superuser **root**.
6. Set the kernel variables that control discipline of the printer. The driver uses a hybrid busy-wait/timeout discipline, to efficiently support in a multi-tasking environment a variety of printers whose buffers come in a multiplicity of sizes.

   The variable **LPWAIT** sets the time for which the processor waits for the printer to accept the next character. If the printer is not ready within the **LPWAIT** period, the processor then resumes normal processing for the number of ticks set by the kernel variable **LPTIME**. Thus, setting **LPWAIT** to an extremely large number (e.g., 1,000) and **LPTIME** to a very small number (e.g., one) results in a fast printer, but leaves very few cpu cycles available for anything else. Conversely, setting **LPWAIT** to a small number (e.g., 50) and **LPTIME** to a large number (e.g., five) results in efficient multi-tasking but also results in a slow printer unless the printer itself contains a buffer (as is normal with all but the least expensive printers). By default, **LPWAIT** is set to 400 and **LPTIME** to four. We recommend that you set **LPWAIT** to no less than 50 and no more than 1,000 and **LPTIME** to no less than one.

   The variable **LPTEST** determines wether the device driver checks to see if the printer is in an "on-line" condition before it uses the device. If your printer does not support this signal, you must set **LPTEST** to zero.

   To reset the values of **LPWAIT**, **LPTIME**, and **LPTEST**, edit file **/etc/conf/mtune** and set the parameters **LPWAIT_SPEC**, **LPTIME_SPEC**, and **LPTEST_SPEC** to the values that you want. Then use the command **/etc/conf/bin/idmkcoh** to build a new kernel. For details on this command, see its entry in the Lexicon. One word of caution to the wary: be sure to name your new kernel something innocuous, such as **cohtest**, to ensure that you do not clobber your current working kernel.

7. Reboot the new kernel and try printing again.

8. If your printer still exhibits problems, try increasing or decreasing the values of **LPTIME** and **LPWAIT**. Remember, each time you build a new kernel kernel, you must reboot in order for the new variables to take effect.

The MLP printer spooler is distributed under license from Magnetic Data Operations, 9400B Two Notch Road, Columbia, SC 29223.

The message

```
cannot open device /dev/lp
```

from **lpr** means either that the printer is not turned on, or that the device **/dev/lp** is not linked to the correct parallel-port device. Use the directions given above to find and link the correct device. The same applies when you receive this message from **hpr**.

printf() — STDIO Function (libc)

Print formatted text
#include <stdio.h>
int printf(*format* **[,***arg1, …. argN***])**
char **format***; **[data type]** arg1, … argN;*

printf() prints formatted text. It uses the *format* string to specify an output format for each *arg*, which it then writes on the standard output.

printf() reads characters from *format* one at a time; any character other than a percent sign '%' or a string that is introduced with a percent sign is copied directly to the output. A '%' tells **printf()** that what follows specifies how the corresponding *arg* is to be formatted; the characters that follow '%' can set the output width and the type of conversion desired. The following modifiers, in this order, may precede the conversion type:

1. A minus sign '-' left-justifies the output field, instead of the default right justify.

2. A string of digits gives the *width* of the output field. Normally, **printf()** pads the field with spaces to the field width; it is padded on the left unless left justification is specified with a '**-**'.

   If the field width begins with '0', the field is padded with '0' characters instead of spaces; the '0' does not cause the field width to be taken as an octal number. Note that this applies only to numeric string descriptors. If the field descriptor describes a character or string (i.e., **%c** or **%s**), **printf()** ignores a leading '0' and always pads the field with spaces.

   If the width specification is an asterisk '*', the routine uses the next *arg* as an integer that gives the width of the field.

3. A period '**.**' followed by one or more digits gives the *precision*. For floating point (**e**, **f**, and **g**) conversions, the precision sets the number of digits printed after the decimal point. For string (**s**) conversions, the precision sets the maximum number of characters that can be used from the string. If the precision specification is given as an asterisk '*', the routine uses the next *arg* as an integer that gives the precision.

4. The letter '**l**' before any integer conversion (**d**, **o**, **x**, or **u**) indicates that the argument is a **long** rather than an **int**. Capitalizing the conversion type has the same effect; note, however, that capitalized conversion types are *not* compatible with all C compiler libraries, or with the ANSI standard. This feature will not be supported in future editions of COHERENT.

The following format conversions are recognized:

% Print a '%' character. No arguments are processed.

c Print the **int** argument as a character.

d Print the **int** argument as signed decimal numerals.

e Print the **float** or **double** argument in exponential form. The format is *d.ddddddesdd*, where there is always one digit before the decimal point and as many as the *precision* digits after it (default, six). The exponent sign *s* may be either '+' or '-'.

f Print the **float** or **double** argument as a string with an optional leading minus sign '-', at least one decimal digit, a decimal point ('**.**'), and optional decimal digits after the decimal point. The number of digits after the decimal point is the *precision* (default, six).

g Print the **float** or **double** argument as whichever of the formats **d**, **e**, or **f** loses no significant precision and takes the least space.

ld Print the **long** argument as signed decimal numerals.

lo Print the **long** argument in unsigned octal numerals.

lu Print the **long** argument in unsigned decimal numerals.

lx Print the **long** argument in unsigned hexadecimal numerals.

o Print the **int** argument in unsigned octal numerals.

p The ANSI standard states that the behavior of the **%p** descriptor is implementation-specific. Under COHERENT, **%p** prints in format **%#.8X** the literal value of a pointer. Its corresponding variable must be of type **char** *.

r The next argument points to an array of new arguments that may be used recursively. The first argument of the list is a **char** * that contains a new format string. When the list is exhausted, the routine continues from where it left off in the original format string.

This descriptor is not part of the ANSI Standard. Its use is deprecated. Code that uses it may not be portable to other systems.

s Print the string to which the **char** * argument points. Reaching either the end of the string, indicated by a null character, or the specified *precision*, will terminate output. If no *precision* is given, only the end of the string will terminate.

u Print the **int** argument in unsigned decimal numerals.

x Print the **int** argument in unsigned hexadecimal numerals. The digits are prefaced by the string **0x**.

X Like **%x**, except that the digits are prefaced by the string **0X**. Note COHERENT release 4.2 has changed the means of **%X** to conform to the ANSI C standard. In versions prior to release 4.2, this format conversion printed a **long** argument in unsigned hexadecimal numerals. Programs that depend upon the obsolete use of **%X** will no long work the same under the current release of COHERENT.

If it wrote the formatted string correctly, **printf()** returns the number of characters written. Otherwise, it returns a negative number.

Example

This example implements a mini-interpreter for **printf()** statements. It is a convenient tool for seeing exactly how some of the **printf()** options work. To use it, type a **printf()** conversion specification at the prompt. The formatted string will then appear. To reuse a format identifier, simply type **<return>**:

```
#include <math.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* the replies go here */
static char reply[80];

/* ask for a string and echo it in reply. */
char *askstr(msg)
char *msg;
{
      printf("Enter %s ", msg);
      fflush(stdout);

      if (gets(reply) == NULL)
            exit(EXIT_SUCCESS);
      return (reply);
}

main()
{
      char fid[80], c;
```

```
      /* initialize to an invalid format identifier */
      strcpy(fid, "%Z");

      for (;;) {
            askstr("format identifier");
            /* null reply uses previous FID */
            if (reply[0])
                  /* leave the '%' */
                  strcpy(fid + 1, reply);

            switch(c = fid[strlen(fid) - 1]) {
            case 'd':
            case 'i':
                  askstr("signed number");
                  if(strchr(fid, 'l') != NULL)
                        printf(fid, atol(reply));
                  else
                        printf(fid, atoi(reply));
                  break;

            case 'o':
            case 'u':
            case 'x':
            case 'X':
                  askstr("unsigned number");
                  if(strchr(fid, 'l') != NULL)
                        printf(fid, atol(reply));
                  else
                        printf(fid, (unsigned)atol(reply));
                  break;

            case 'f':
            case 'e':
            case 'E':
            case 'g':
            case 'G':
                  printf(fid, atof(askstr("real number")));
                  break;

            case 's':
                  printf(fid, askstr("string"));
                  break;

            case 'c':
                  printf(fid, *askstr("single character"));
                  break;

            case '%':
                  printf(fid);
                  break;

            case 'p':
                  /* print pointer to format id */
                  printf(fid, fid);
                  break;

            case 'n':
                  printf("n not implemented");
                  break;

            default:
                  printf("%c not valid", c);
            }

            printf("\n");
      }
}
```

See Also

ecvt(), fcvt(), fprintf(), gcvt(), libc, putc(), puts(), scanf(), sprintf(), vprintf()
ANSI Standard, §7.9.6.3

POSIX Standard, §8.1

Notes

Because C does not perform type checking, it is essential that each argument match its counterpart in the format string.

Versions of COHERENT prior to release 4.2 recognized the conversion formats **%D**, **%O**, and **%U**. The ANSI standard does not recognize these conversion characters, and beginning with release 4.2 the COHERENT implementation of **printf()** no longer recognizes them. You should instead use, respectively, the conversion characters **%ld**, **%lo**, and **%lu**.

proc.h — Header File

Define structures/constants used with processes
#include <sys/proc.h>

proc.h defines structures and constants used by routines that manipulate processes.

See Also

header files

process — Definition

A **process** is a program in the state of execution.

See Also

daemon, file, Using COHERENT

prof — Command

Print execution profile of a C program
prof [-abcs][*progfile* **[** *monfile* **]]**

prof interprets the profile file produced by an execution of a C program and reports the execution frequencies of each routine. It also reports the percentage of execution time spent in each routine.

prof normally reports times and frequencies spent for regions of programs between externally defined names. *progfile* is the executable program; if omitted, **a.out** is assumed. *monfile* is the monitor file produced during execution of the program; if omitted, **mon.out** is assumed.

To produce **mon.out**, a program must be compiled with the **-VPROF** option to **cc**. To profile all modules, each module must be compiled with this option.

The following options are available.

-a Profile all symbols, not just externals.

-b Print all bin information.

-c Print all call information.

-s Report stack usage high-water mark.

Files

a.out — Program file (with name list intact)
mon.out — Raw execution profile

See Also

cc, commands, ld, nm

profile — System Administration

Set default environment at login
/etc/profile

The shell executes the script **/etc/profile** whenever any user logs in. This script sets up the default environment for a user. Note that the actions of this script can be altered or supplemented by each user's **.profile** script.

If **/etc/passwd** specifies a program in the login-shell slot, then **/etc/profile** is read by **/bin/sh**. Those lines that begin with the command **export** are recognized as global environments, and the remainder of the line is inserted

into the environment.

Please note that if **/bin/sh** or **/bin/ksh** is not the shell, any constructions other than

```
export foo=value
```

are not likely to work.

See Also

Administering COHERENT, ksh, .kshrc, .profile, sh

.profile — System Administration

Execute commands at login
$HOME/.profile

The shell reads file **$HOME/.profile** whenever a user logs in. The user can edit its contents to set up her environment however she prefers, and to execute programs routinely upon login.

The following gives one user's **.profile**:

```
MAIL=/usr/spool/mail/sally
PATH=/usr/bin:/bin:/v/sally/bin:.
EDITOR=me
PS1="Sally(!) "
PS2="MORE(!)> "
PAGER=scat
set -h
set -o emacs
echo "CALENDAR:"
calendar
echo ""
/usr/games/fortune
```

The first six entries set environmental variables; note that these are in addition to the variables set in **/etc/profile**.

The next two entries

```
set -h
set -o emacs
```

set two features of the Korn shell, which is used by the person. The first turns on its hashing feature, and the second turns on MicroEMACS-style editing of the command line.

The last four entries

```
echo "CALENDAR:"
calendar
echo ""
/usr/games/fortune
```

execute two programs upon login. The two **echo** commands print, respectively, the word **CALENDAR** and a blank line on the screen. The command **calendar** reads the user's personal calendar and prints all entries the relate to today (or to the weekend, should today be a Friday). The command **fortune** prints a randomly selected (and, we hope, amusing) select from file **/usr/games/lib/fortunes**.

This example is relatively simple. A user's **.profile** can be turned into a complex shell program if you wish.

See Also

Administering COHERENT, ksh, .kshrc, profile, sh, Using COHERENT

Programming COHERENT — Overview

The C language is the "native language" of COHERENT. Most COHERENT programs are written in C.

If you are a beginner and are interested in learning something about C, look at the tutorial *The C Language* in the first part of this manual.

The following Lexicon entries give you information you need to write or port C programs under COHERENT:

C keywords
This lists the C keywords recognized by the COHERENT implementation of C. Each keyword, in turn, is described in full in its own Lexicon entry.

C language
This summarizes the COHERENT implementation of C. It gives the size of each data type, formatting of floating-point data, static limits, and other information.

C preprocessor
This describes the processing directives that the COHERENT preprocessor recognizes. Each directive is described in full in its own Lexicon entry

header files
This entry names the header files included as part of COHERENT. Each header file is described in its own Lexicon entry. Some of the header-file articles are of particular interest.

libraries
This describes the libraries included with COHERENT. Almost every library function and system call has its own Lexicon entry; the only exceptions are the routines kept in **libmisc.a** and **libcurses.a**. Each library has its own summary entry; of particular interest are the entries **libc**, **libm**, **libgdbm**, and **libsocket**.

If you are an experienced C programmer who is new to COHERENT, we suggest you look first at the article for **C language**, to get an overview of the dialect of C that COHERENT supports. Look at the entry for libraries, to see what libraries are available; then look at the entry for each library to see what functions are available.

The following Lexicon entries describe the commands with which you can compile and manage your programs:

ar The archiver. This turns a group of object modules into a library.

as The COHERENT macro-assembler. This assembles modules written in assembly language, and builds object modules that you can link with modules written in C or other languages.

cc The C compiler. This describes the compiler itself, and its options and switches.

cpp The C preprocessor. The preprocessor itself has its own options to help you control the building of your programs.

db The symbolic debugger. With **db**, you can set breakpoints, single-step through code, hot-patch binaries, and otherwise debug your programs. It requires knowledge of 80386 assembly language.

ld The linker. This links object modules into an executable binary. The Lexicon entry describes its switches and features.

make The programming discipline. **make** helps you to manage the building of a complex program. It is indespensible for managing all but the simplest programming projects.

nm This utility prints the contents of a program's symbol table.

sh The Bourne shell. This is of the COHERENT command interpreter. You can write large, complex programs in the shell. These can functions, and draw on a library of prewritten functions. The shell is one of the most powerful tools available to a COHERENT programmer — and one of the most neglected.

strip Strip the symbol table from a program. This makes most programs significantly smaller, with no loss in functionality.

Each command is described in its own Lexicon entry.

Definitions

The following Lexicon entries give technical definitions of interest to programmers:

address
What an "address" is.

alignment
What byte alignment is, and how it applies under the various machine on which COHERENT has been implemented

ANSI A brief introduction to the ANSI Standard for Programming Language C.

arena What an arena is, and how it applies to COHERENT programs.

array What an array is, and elementary information on how to code it.

ASCII The ASCII table.

bit What a bit is.

bit map
What a bit map is, and how to code it under C.

buffer What a buffer is, and how buffering affects your languages.

byte What a byte is.

byte ordering
This describes how bytes and words are ordered on the various machines on which COHERENT has been implemented.

calling conventions
The calling conventions for COHERENT functions. This is particularly important if you are writing modules in assembly language.

cast How to "coerce" one data type into another.

cc0 The COHERENT C parser.

cc1 The COHERENT C code generator.

cc2 The COHERENT C optimizer.

cc3 The COHERENT de-compiler. It generates a file of assembly language for your examination.

data formats
This gives the size of the common data types on the various machines on which COHERENT has been implemented.

data types
The data types that COHERENT C recognizes.

environ
This article introduces the argument **environ**, which by default is the third argument passed to the function **main()** in a C program. It points to image of the process's environment.

errno This global variable holds the error status returned by a COHERENT system call. The article **errno.h** interprets the codes that can appear in this variable.

execution
This describes how each form of the system call **exec()** executes a program.

field Description of what a field is, and how to address it.

FILE Description of the **FILE** structure used by STDIO routines.

file What a file is. It also goes into the "black art" of permissions.

file descriptor
Description of the file descriptor used by COHERENT system calls.

function
What a function is.

GMT A brief introduction to Greenwich Mean Time, which is the internal time for every COHERENT system.

initialization
This describes the rules of initialization for C.

interrupt
What an interrupt is.

Latin 1 The table ISO Latin 1 (ISO 8859.1).

lvalue Definition of the "left value" in a C expression.

macro What a C macro is, and how COHERENT C processes them.

manifest constant
This introduces manifest constants, and lists the constants that COHERENT defines automatically.

modulus
A definition of the modulus arithmetic operation.

NUL Definition of the NUL character.

nybble What a "nybble" is.

object format
Definition of an object format.

operator
A list of the C operators. This article also gives a table of precedence for the operators.

pattern
What a pattern is.

pointer
What a pointer is, and tips for using pointers with COHERENT C.

portability
This gives some tips on how to write portable programs.

POSIX Standard
A brief introduction to the POSIX Standard

random access
A definition of random access.

read-only memory
A definition of ROM, or "read-only memory".

recursion
A definition of this programming technique.

rvalue Definition of the "right value" in a C expression.

signame
This global array holds a string that describes the signal that a program has received.

stack A definition of the program stack, and how to manipulate it under COHERENT C.

standard error
Definition of the standard-error device.

standard input
Definition of the standard-input device.

standard output
Definition of the standard-output device.

stderr The file descriptor of the standard-error device.

stdin The file descriptor of the standard-input device.

STDIO Definition of STDIO — i.e., "standard input and output".

stdout The file descriptor of the standard-output device.

storage class
This entry summarizes the classes of storage that COHERENT C recognizes.

stream Definition of a file stream.

STREAMS
This article summarizes the COHERENT implemenation of STREAMS.

structure
Definition of a structure, and basic information on how to code it.

structure assignment
This details structure assigment under COHERENT C.

stty Summary of the **stty** interface to terminals.

termio Introduction to the **termio** terminal interface.

termios
This summarizes the POSIX Standard extensions to the **termio** terminal interface.

type checking
This details type checking under COHERENT C.

type promotion
This details type promotion under COHERENT C.

Other Languages

COHERENT includes the following programming languages:

awk This interpreted language lets you write programs for text processing. It is especially good at processing tabular information, thus letting you quickly write simple data-base programs.

bc **bc** is a calculator program that offers infinite magnitude and infinite precision. This is an interpreted langauge that you can program on the fly to perform simple tasks, such as computing interest payments on the national debt. You can also write programs that you can run repeatedly. These can also take advantage of a library of routines already written for you.

lex This program reads a set of lexical analysis rules that you write in a standard form, and generates a C program that you can compile and run.

yacc This program reads a set of parsing rules that you write in Backus-Naur Form, and generates a C program that you can compile and run. You can use with code generated by **lex** to write complex programs, such as compilers.

Each of these languages is described in a Lexicon article. The front of the manual has a tutorial for each.

See Also

Administering COHERENT, C language, COHERENT, commands, libraries, Using COHERENT

protocols — System Administration

Name communications protocols
/etc/protocols

The file **/etc/protocols** describes the Internet protocols that your local host recognizes. Each line within this file describes one protocol. A description consists of the following fields:

- The protocol's official name.
- Its number.
- Aliases, if any, for the protocol name.

Any text that follows the character '#' is comment, and is ignored by any program that reads this file.

For example:

```
icmp    1    ICMP    # internet control message protocol
ggp     3    GGP     # gateway-gateway protocol
tcp     6    TCP     # transmission control protocol
```

See Also

Administering COHERENT, hosts, hosts.equiv, inetd.conf, networks, services

prps — Command

Prepare files for PostScript-compatible printer
prps [*options***] [***file ...* **]**

prps reads each *file*, breaks it into pages, writes a header at the top of each page, then writes the paginated text onto the standard output. If no *file* is given, **prps** reads the standard input.

Unike the related command **pr**, **prps** writes its output in the PostScript language, suitable for printing on a PostScript printer such as an Apple LaserWriter or a Hewlett-Packard LaserJet with a PostScript cartridge. The PostScript output program generates a sequence of standard 8.5×11-inch pages, each containing a header line (file name, current time and date, and page number) and a box that encloses the text of *file*. The default output typeface is ten-point Courier.

prps recognizes the following options:

-b Suppress the box around the page text. If the box is present, PostScript clips text that would extend beyond its right border.

-h Suppress the header line.

-i*n* Indent the left margin by an additional *n* characters.

-l Generate "landscape"-format output. **prps** normally generates output pages in "portrait" format (upright 8.5×11 inches). The **-l** option generates output pages in landscape format (11 by 8.5) instead. This option is useful for files with long lines; by default, it prints 46 lines per page.

-l2 Generate landscape-format output pages that each contain two side-by-side "pages" of text. This format is useful for saving paper, especially when used with a small size of type. As it prints in a small size of type, it prints 66 lines per page.

-n*name* Use *name* in place of the file name in the header line.

-t*N* Set tab stops at every *N* characters. The default tab setting is eight.

-ptsize Change the size of type to *ptsize* points. By default, **prps** sets its output in ten-point type. This yields 64 lines per normal output page, 46 lines in landscape format, and 52 lines per half page in **-l2** format. (Note that a "point" is one twelfth of a pica, which in turn is one sixth of an inch; thus, there are 72 points in an inch.) By specifying the *ptsize* on its command line, you can tell **prps** to use a different size of type. For example, **-8** tells **prps** to use eight-point type.

-p*N* Print *N* lines of text on each output page (or half page). Note that the point size determines how many lines fit on a page, and lines per page determine point size. If you specify both, **prps** will use the given values unless the lines do not fit at the given point size.

+*N* Skip the first *N* output pages.

Setting Fonts

prps recognizes the standard **nroff** font specification sequences and translates them into PostScript font specifications. The default font is Courier. Because the naming conventions for PostScipt fonts are anything but uniform, **prps** appends a suffix to the fontname to designate a Roman, boldface and italic font variety. The default suffix is ' ' for Roman, "-Bold" for bold and "-Oblique" for italic. These give the standard PostScript names for the Courier family, "Courier", "Courier-Bold", and "Courier-Oblique".

Option **-f***fontname* specifies an alternative *fontname.* Option **-Fs***Xsuffix* specifies an alternative font suffix, where *X* is one of the three characters **RBI** (for **R**oman, **B**old or **I**talic) and *suffix* is the desired suffix. For example, the option

```
-fTimes -FsR-Roman -FsI-Italic
```

generates the usual PostScript font names for the Times family, namely "Times-Roman", "Times-Bold", and "Times-Italic".

To spare you some of this grief, a few fonts have built-in abbreviations. Option **-F***X*, where *X* is one of the characters **ABHNPST,** specifies a PostScript fontname as follows:

-FA AvantGarde
-FB Bookman
-FH Helvetica
-FN Helvetica-Narrow
-FP Palatino
-FS New Century Schoolbook
-FT Times

These options also set each suffix appropriately for the desired font. However, font naming conventions may differ on various PostScript devices; examine the **prps** output and your device documentation if problems occur.

Examples

prps is especially useful as a way of printing the output of **nroff**, including manual pages. For example,

```
man prps | prps | hpr -B
```

or

```
man prps | prps -l2 | hpr -B
```

prints this Lexicon article in, respectively, portrait mode or two-page landscape mode. It looks nicer if you center the output with an indent:

```
man prps | prps -i8 | hpr -B
```

or

```
man prps | prps -l2 -i4 | hpr -B
```

See Also

commands, hp, hpr, lp, pr, nroff, printer

Notes

When you installed COHERENT onto your system, the installation program asked you whether your printer used the PostScript language. For information on how to install a PostScript printer onto your system, see the Lexicon entries for **lp** and **printer**.

ps — Command

Print process status
ps [-][adefglmnrtwx] [-c *sys***] [***mem***] [-p***pid***,***pid***,...,***pid***]**

ps prints information about a process or processes. It prints the information in fields, followed by the command name and arguments. The fields include the following:

TTY The controlling terminal of the command, printed in short form. For example, "tty44:" means **/dev/tty44**. A dash means there is no controlling terminal.

PID Process id; necessary to know when the process is to be killed.

GROUP PID of the group leader of the process, that is, the shell that started up when the user logged in.

PPID PID of the parent of the process; very often a shell.

UID User id or name of the owner.

K Size of the process, in kilobytes.

F Process flag bits, as follows:

PFCORE	00001	Process is in core
PFLOCK	00002	Process is locked in core
PFSWIO	00004	Swap I/O in progress
PFSWAP	00010	Process is swapped out
PFWAIT	00020	Process is stopped (not waited)
PFSTOP	00040	Process is stopped (waited on)
PFTRAC	00100	Process is being traced
PFKERN	00200	Kernel process
PFAUXM	00400	Auxiliary segments in memory
PFDISP	01000	Dispatch at earliest convenience
PFNDMP	02000	Command mode forbids dump
PFWAKE	04000	Wakeup requested

S State of the process, as follows:

R	Ready to run (waiting for CPU time)
S	Stopped for other reasons (I/O completion, pause, etc.)
T	Being traced by another process
W	Waiting for an existent child
Z	Zombie (dead, but parent not waiting)

EVENT The condition that the process is anticipating. This not applicable if the process is ready to run. The following gives the legal symbolic names of events. If a driver does not support symbolic event names, **ps** prints a unique hexadecimal number instead:

System Sleeps:

bpwait	Wait for a buffer to become valid
bufneed	Wait for a free buffer to become available
bwrite	Wait for a buffer write to finish
ioreq	An IO request is being processed
pause	This process is in the **pause()** system call
pipe data	Wait for data to appear in a pipe
pipe wx	
poll	Wake for polled event, poll timeout, or signal
ptrace	Send a **ptrace** command to a traced child
ptret	Wait for signal processing in a traced child to complete
pwrite	Wait for a pipe to empty enough for a write
swap	Wait for a process to get swapped in
wait	Wait for a child to terminate
waitq	Wait for more character queues to become available

Driver Sleeps

aha:ccb	AHA-154x driver is waiting for a SCSI command to complete
nkbcmd	
nkbcmd...	
nkbcmd2	
nkbcmd2...	**nkb** is waiting for a command to complete
ptycd	Pseudoterminal driver is waiting for carrier
ptyread	Pseudoterminal driver is waiting for a read
ptywrite	Pseudoterminal driver is waiting for a write
ttydrain	Line discipline is waiting for a tty to drain
ttyiodrn	**ioctl()** asked line discipline to let tty output drain
ttyoq	Line discipline is waiting for an output queue to drain
ttywait	Line discipline is waiting for more data

CVAL SVAL IVAL RVAL
Scheduling information; bigger is better.

UTIME Time consumed while running in the program (in seconds).

STIME Time consumed while running in the system (in seconds).

Normally, **ps** displays the **TTY** and **PID** fields of each active process started on the caller's terminal, as well as the command name and arguments. The following flags alter this behavior.

-a Display information about processes started from all terminals.

-c *sys* This option does nothing; it is included to preserve the integrity of some shell scripts.

-d Print information about status of loadable drivers.

-e Same as **-a**. This is included for compatibility with other implementation of **ps**.

-f Blank fields have '-' place-holders. This enables field-oriented commands like **sort** and **awk** to process the output.

-g Print the group leader field GROUP if the **l** option is given.

-k *mem* The next argument *mem* is the memory image (default, **/dev/mem**). Note that this argument currently does nothing; it is included only to preserve old shell scripts. The COHERENT implementation of **ps** reads information from **/dev/ps**. This permits **ps** to be smaller and faster, helps to avoid "ghosts," and to be atomic.

-l Long format. In addition to the TTY and PID fields, prints the PPID, UID, K, F, S and EVENT fields.

-m This option does nothing; it is included to preserve the integrity of some shell scripts.

-n Suppress the header line.

-p*pid*,*pid*,...,*pid*
Print information for each process identifier *pid* in the comma-separated list.

-r Print the real size of the process, which includes the user and auxiliary segments assigned to the process. Because the user segment (usually 1 kilobyte) is shared by all processes owned by that user, this may give a misleading total size for all the user's processes.

-t Print elapsed CPU time fields UTIME and STIME.

-w Wide format output; print 132 columns instead of 80.

-x Display processes which do not have a controlling terminal.

Files

/dev/ps — Device for a system driver
/dev/tty* — List of terminal names

See Also

commands, hmon, kill, mem, ps [device driver]**, size, wait**

Notes

Each process can modify or destroy its command name and arguments. The state of the system changes even as **ps** runs.

ps — Device Driver

Driver to return information about processes
/dev/ps

The file **/dev/ps** accesses the kernel's process table. It is a part of the driver **mem**, which manages memory; thus, it has major number 0 and minor number 6.

/dev/ps is a read-only device that exists only to support the command **ps** and its variants. The command **ps** reads this device to display a "snapshot" of the processes that the COHERENT kernel is executing.

Reading **/dev/ps** deposits an array of the structure **stMonitor** into the read buffer. The number of bytes requested by the system call **read()** should be enough to accommodate the entire process table. Header file **<sys/coh_ps.h>** defines **stMonitor**.

See Also

device drivers, ps [command]

PS1 — Environmental Variable

User's default prompt
PS1=*prompt*

The environmental variable **PS1** sets the prompt for your shell. The default is **$**.

See Also

environmental variables, PS2, sh

PS2 — Environmental Variable

Prompt when user continues command onto additional lines
PS2=*prompt*

The environmental variable **PS2** sets the prompt that is displayed when a command extends onto additional input lines. The default is **>**.

See Also

environmental variables, PS1, sh

PSfont — Command

Cook an Adobe font into PostScript format
PSfont [-qs] [*infile***.pfb [** *outfile* **]]**

The command **PSfont** "cooks" a file that is in Adobe's downloadable-font format into PostScript. The output of **PSfont** can either be loaded into your PostScript printer as a memory-resident font, which can be used across multiple files, or included within the output of **troff**.

PSfont recognizes two options:

-q Quiet option: suppress the printing of warning messages. **PSfont** normally complains about error conditions it finds within fonts, such as extraneous control characters.

-s Suppress the instructions **serverdict** and **exitserver** from the output. Use this option if you wish to include the output of **PSfont** within **troff** output; do *not* use this option if you want the cooked font to be resident within the printer after you download it.

infile is the Adobe font file that **PSfont** cooks into PostScript. It must have the suffix **.pfb**. If you do not name an *infile* on the command line, **PSfont** reads the standard input.

outfile names the file into which **PSfont** writes its output. By convention, it should have the suffix **.ps**, although this is not required. If you do not name an *outfile* on the command line, **PSfont** writes to the standard output.

See Also

commands, fwtable, troff
Supporting downloadable PostScript language fonts, Adobe Technical Note No. 5040, §3.3. Mountain View, Ca., Adobe, Incorporated, 1992.

Notes

For more information on using **PSfont** with **troff**, see the Lexicon entry for **troff**.

ptrace() — System Call (libc)

Trace process execution
#include <signal.h>
int ptrace(*command, pid, location, value***)**
int *command, pid***, ****location***,** *value***;**

ptrace() provides a parent process with primitives to monitor and alter the execution of a child process. These primitives typically are used by a debugger such as **db**, which needs to examine and change memory, plant breakpoints, and single-step the child process being debugged.

Once a child process indicates it wishes to be traced, its parent issues various *command*s to control the child. *pid* identifies the affected process. The parent may issue a command only when the child process is in a stopped state, which occurs when the child encounters a signal. A special return value of 0177 from **wait()** informs the parent that the child has entered the stopped state. The parent may then examine or change the child process memory

space or restart the process at any point.

When the child process issues an **exec()**, the child stops with signal **SIGTRAP** to enable the parent to plant breakpoints. The set user id and set group id modes are ineffective when a traced process performs an **exec()**.

The following list describes each available *command*. A *command* ignores any arguments not mentioned.

0 This is the only *command* the child process may issue. It tells the system that the child wishes to be traced. Parent and child must agree that tracing should occur to achieve the desired effect. Only the *command* argument is significant.

1,2 The **int** at *location* is the return value. Command 1 signifies that *location* is in the instruction space, whereas command 2 signifies *data* space. Often these two spaces are equivalent.

3 The return value is the **int** of the process description, as defined in **sys/uproc.h.** This call may be used to obtain values such as hardware register contents and segment allocation information.

4,5 Modify the child process's memory by changing the **int** at *location* to *value.* Command 4 means instruction space and command 5 means data space. Shared segments may be written only if no other executing process is using them.

6 Modify the **int** at *location* in the process description area, as with command 3. The permissible values for *location* are restricted to such things as hardware registers and bits of machine status registers that the user may safely change.

7 This command restarts the stopped child process after it encounters a signal. The process resumes execution at *location,* or from where the process was stopped if *location* is **(int *)1.** *value* gives a signal number that the process receives as it restarts. This is normally the number of the signal that caused the process to stop, fetched from the process description area by a **3** command. If *value* is zero, the effect of the signal is ignored.

8 Force the child process to exit.

9 Like command **7**, except that the child stops again with signal **SIGTRAP** as soon as practicable after the execution of at least one instruction. The actual hardware method used to implement this command varies from machine to machine, explaining the imprecise nature of its definition. This call may provide part of the basis for breakpoints.

Files

<signal.h>
<sys/uproc.h>

See Also

db, exec, libc, ptrace.h, signal(), wait()

Diagnostics

ptrace() returns -1 if *pid* is not the process id of an eligible child process or if some other argument is invalid or out of bounds. Some commands may return an arbitrary data value, in which case **errno** should be checked to distinguish a return value of -1 from an error return.

Notes

There is no way to specify which signals should not stop the process.

ptrace.h — Header File

Perform process tracing
#include <sys/ptrace.h>

The header file **ptrace.h** holds definitions used by routines that perform process tracing. Among other things, it defines the structure **ptrace**.

See Also

header files

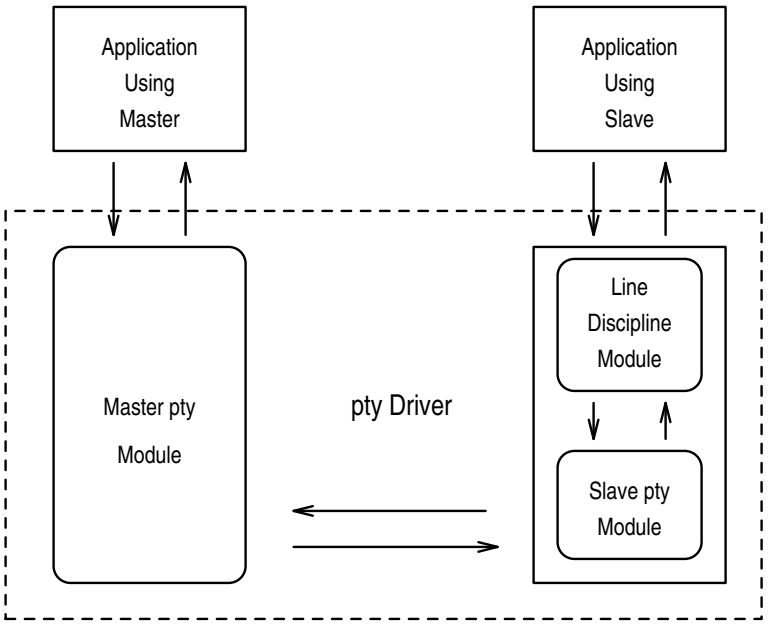
pty — Device Driver

Device driver for pseudoterminals

The COHERENT device driver **pty** lets your system support up to 128 pairs of pseudoterminals, or *ptys*.

A *pseudoterminal* is a means of letting a process masquerade as a terminal. For example, when you run the program **xterm** under X, that program passes what you type into COHERENT through a pseudoterminal device.

Each pseudoterminal consists of a pair of devices: a master device and a slave device. The program that is accepting input from a human at a keyboard (e.g., **xterm**) is "plugged" into the slave device; the program that is accepting and processing the input (e.g., a shell) is plugged into the master device. The following diagram shows how this pair of devices relate to each other:



As you can see, the slave device talks to the keyboard through a sub-module that performs line discipline. *Line-discipline* handles backspace characters, handles special interrupt characters (such as **<ctrl-C>**), and converts line-feed characters into carriage-return—line-feed character pairs: it bundles what you type into a package that can be passed to the master application and processed.

Only one process at a time can open a master device; the device is opened as soon as requested. Several processes can open a slave device, but blocks until the matching master device has been open. When blocked in this way, the slave is said to be "waiting for pseudocarrier."

An attempt to read a master device when no input is available, or to write to a master device when the slave cannot accept data, will block unless nonblocking I/O has been specifically requested; in this case, the system calls **read()** or **write()** fail and **errno** is set to EAGAIN.

You can use the system call **ioctl()** on slave devices with all valid line-discipline commands, including **TCGETA**, **TCSETA**, **TCSETAW**, **TCSETAF**, and **TCFLSH**. There are no valid **ioctl()** commands for master devices.

The system call **poll()** is allowed with both master and slave **pty** devices. However, priority polls (**POLLPRI**) are not supported.

Master devices are named **/dev/pty[p-w][0-f]**. Corresponding slaves are **/dev/tty[p-w][0-f]**. Like any other device, each **pty** has a major and minor number. The major number is 9 (**PTY_MAJOR** in system header file **<sys/devices.h>**). For slave devices, minor numbers are assigned according to the following scheme:

device	*Major number*	*Minor number*
/dev/ttyp0	9	0
/dev/ttyp1	9	1
...		

```
/dev/ttyp9          9                   9
/dev/ttypa          9                  10
/dev/ttypb          9                  11
...
/dev/ttypf          9                  15
/dev/ttyq0          9                  16
...
/dev/ttyw0          9                 112
...
/dev/ttywf          9                 127
```

For master devices, use **pty** instead of **tty** in the device name, and add 128 to the minor number.

The configurable parameter **NUPTY_SPEC** sets the number of **pty** pairs that may be used. The default is eight. If you want to change this value, invoke the script **/etc/conf/pty/mkdev** and enter the new value at the appropriate prompt. Then use the command **/etc/conf/bin/idmkcoh** to build a new kernel that incorporates this change; when the new kernel is built, boot it. For details, see the Lexicon entry for the command **idmkcoh**.

Specifying a value of zero for **NUPTY_SPEC** will cause the **pty** device to be omitted from the next kernel that **idmkcoh** generates.

See Also

device drivers

pushd — Command

Push an item onto the directory stack
pushd *[directory0 ... directoryN]*

The COHERENT shell **sh** maintains an internal "directory stack", which is a stack of names of directories. You can manipulate this stack should you, for any reason, wish to traverse a number of directories quickly and efficiently.

The command **pushd** pushes *directory1* through *directoryN* onto the directory stack, and changes the current directory to the last directory pushed. If called without an argument, it transposes the last two directories on the directory stack.

See Also

commands, dirs, popd, sh

putc() — STDIO Function (libc)

Write character into stream
#include <stdio.h>
int putc(*c***,** *fp***) char** *c***; FILE** **fp***;*

putc() writes character *c* into the file stream to which *fp* points. It returns *c* upon success.

Example

The following example demonstrates **putc()**. It opens an ASCII file and prints its contents on the screen. For another example of **putc()**, see the entry for **getc()**.

```
#include <stdio.h>
main()
{
      FILE *fp;
      int ch;
      int filename[20];

      printf("Enter file name: ");
      gets(filename);
```

```
      if ((fp = fopen(filename,"r")) != NULL) {
            while ((ch = fgetc(fp)) != EOF)
                  putc(ch, stdout);
      } else
            printf("Cannot open %s.\n", filename);
      fclose(fp);
}
```

See Also

fputc(), getc(), libc, putchar()
ANSI Standard, §7.9.7.8
POSIX Standard, §8.1

Diagnostics

putc() returns **EOF** when a write error occurs.

Notes

Because **putc()** is a macro, arguments with side effects may not work as expected.

putchar() — STDIO Function (libc)

Write a character onto the standard output
#include <stdio.h>
int putchar(*c***)**
char *c***;**

putchar() is a macro that expands to **putc(***c***, stdout)**. It writes a character onto the standard output.

Example

For an example of this routine, see the entry for **getchar()**.

See Also

fputc(), libc, putc()
ANSI Standard, §7.9.7.9
POSIX Standard, §8.1

Diagnostics

putchar() returns **EOF** when a write error occurs.

Notes

Because **putchar()** is a macro, arguments with side effects may not work as expected.

putenv() — General Function (libc)

Add a string to the environment
#include <stdlib.h>
int putenv (*envstring***)**
char **envstring***;*

The function **putenv()** puts *envstring* into the user's environment. You can use this function to set a new environmental variable, or to change the definition of an existing variable.

envstring must point to a string of the form *VARIABLE***=***value*, where *VARIABLE* is the environmental variable being set, and *value* is the value to which it is being set.

putenv() returns zero if all goes well. If something goes wrong, it returns a value other than zero.

See Also

environ, environmental variables, getenv(), libc, stdlib.h

Notes

The global variable **environ**, which points to a process's environment, points to an array of pointers to strings rather than to an array of strings. When **putenv()** inserts *envstring* into the environment, it calls **malloc()** to enlarge the array of string pointers to which **environ** points, then inserts a pointer to *envstring* into that array. It does not copy *envstring* anywhere.

If a process uses **putenv()** to insert a string pointer into the environment, it can also call **getenv()** to read back that string; however, the array of strings passed to the process via **envp** (the third argument to the function **main()**) is not affected by a call to **putenv()**. For details on **environ** and **envp**, see their entries in the Lexicon.

It is an error to call **putenv()** with a pointer to an automatic variable as the argument, and then exit the calling function while *envstring* is still part of the environment. For safety's sake, *envstring* should point to a string that is static or global. See the Lexicon entry for **static**, or see the ANSI Standard §3.5.1.

putmsg() — System Call (libc)

Place a message onto a stream
#include <stropts.h>
int putmsg (*fd*, *ctlptr*, *dataptr*, *flags*)
int *fd*, *flags*; **const struct strbuf** **ctlptr*, **dataptr*;

putmsg() creates a message from user-specified buffer (or buffers), and sends the message to a STREAMS file. The message can contain either a data part, a control part, or both. The data and control parts to be sent are distinguished by being placed in separate buffers, as described below. The semantics of each part are defined by the STREAMS module that receives the message.

fd gives a file descriptor that identifies an open stream. *ctlptr* and *dataptr* each point to a structure of tyupe **strbuf**, which contains the following members:

```
int len;    /* Length of data */
void *buf;  /* Pointer to buffer */
```

ctlptr points to the structure that describes the control part (if any) to be included in the message: **buf** points to the buffer wherein the control information resides, and **len** gives the number of bytes to be sent.

Likewise, *dataptr* specifies the data (if any) to be included in the message. *flags* gives the message's type; it is described in detail below.

To send the data part of a message, *dataptr* must not be NULL, and the value of *dataptr***.len** must be no less than zero. To send the control part of a message, the corresponding values must be set for *ctlptr*. **putmsg()** does not send the data portion of the message if *dataptr* is set to NULL or *dataptr***.len** equals -1; likewise, **putmsg()** does not send the control portion of the message if *ctlptr* is NULL or *ctrlptr***.len** equals -1.

If a control part is specified and *flags* equals **RS_HIPRI**, **putmsg()** sends a high-priority message. If no control part is specified and *flags* equals **RS_HIPRI**, **putmsg()** fails and sets **errno** to **EINVAL**. If *flags* is set to zero, **putmsg()** sends a message of normal priority. If neither the control part nor the data part is specified, and if flags is set to zero, **putmsg()** sends no message and returns zero.

The stream head guarantees that the control part of a message generated by **putmsg()** is at least 64 bytes long.

putmsg() usually blocks if the stream head's write queue is full due to internal flow-control conditions. For high-priority messages, **putmsg()** does not block on this condition. For other messages, **putmsg()** does not block when the write queue is full and you have set the mode on *fd* to **O_NDELAY** or **O_NONBLOCK**. **putmsg()** never sends a partial message. For details on **O_NDELAY** and **O_NONBLOCK**, see the Lexicon entry for **open()**.

Upon successful completion, **putmsg()** returns zero. If something goes wrong, **putmsg()** returns -1 and sets **errno** to an appropriate value. **putmsg()** fails if any of the following conditions is true:

- A non-priority message was specified, the mode on *fd* was set to **O_NDELAY** or **O_NONBLOCK**, and the stream-write queue is full due to internal flow-control conditions. **putmsg()** sets **errno** to **EAGAIN**.
- *fd* is not a valid file descriptor. **putmsg()** sets **errno** to **EBADF**.
- *ctlptr* or *dataptr* contains an illegal address. **putmsg()** sets **errno** to **EFAULT**.
- Your application caught a signal while it was executing **putmsg()**. **putmsg()** sets **errno** to **EINTR**.
- *flags* contains an undefined value, or you set *flags* **RS_HIPRI** but did not supply a control part. **putmsg()** sets **errno** to **EINVAL**.
- The stream referenced by *fd* is linked below a multiplexor. **putmsg()** sets **errno** to **EINVAL**.
- **putmsg()** could not allocate buffers for the message it was to send due to insufficient STREAMS memory resources. **putmsg()** sets **errno** to **ENOSR**.

- *fd* does not identify a stream. **putmsg()** sets **errno** to **ENOSTR**.
- A hangup condition was generated downstream for the specified stream, or the other end of the pipe is closed. **putmsg()** sets **errno** to **ENXIO**.
- The size of the message's data portion does not fall within range of legal packet sizes set by topmost stream module, or its control portion exceeds the maximum configured size. **putmsg()** sets **errno** to **ERANGE**.

putmsg() also fails if a STREAMS error message had been processed by the stream head before the call to **putmsg()** was executed. **putmsg()** returns the the value contained in the STREAMS error message.

See Also

getmsg(), libc, STREAMS

putp() — terminfo Function

Write a string into the standard window
#include <curses.h>
putp(*string***)**
char **string***string***;**

COHERENT comes with a set of functions that help you read **terminfo** descriptions to manipulate a terminal. **putp()** writes the *string* into the standard window. It is equivalent to **tputs(***string***, 1, putchar);**.

See Also

curses.h, terminfo, tputs()

puts() — STDIO Function (libc)

Write string onto standard output
#include <stdio.h>
int puts(*string***)**
char **string***string*

puts() appends a newline character onto the string to which *string* points, and writes the result onto the standard output. If all goes well, it returns a nonnegative value (not necessarily -1); if an error occurs, it returns EOF.

Example

The following uses **puts()** to write a string on the screen.

```
#include <stdio.h>

main()
{
      puts("This is a string.");
}
```

See Also

fputs(), libc
ANSI Standard, §7.9.7.10
POSIX Standard, §8.1

Notes

For historical reasons, **fputs()** outputs the string unchanged, whereas **puts()** appends a newline character.

pututline() — General Function (libc)

Write a record into a logging file
#include <utmp.h>
struct utmp **pututline(***record***)**
const struct utmp **record***record***;**

Function **pututline()** writes *record* into the file that logs login events. It is designed to update a record within the logging file.

record points to the record to be insert into the logging file. It is of type **utmp**, which is a structure whose fields describe a login event. (For a detailed description of this structure, see the Lexicon entry for **utmp.h**.)

pututline() assumes that you have first called **getutent()**, **getutid()**, or **getutline()** to open the logging file, and that the file's seek pointer is is at or before the record you wish to update. **pututline()** looks for the first record within the logging file whose field **ut_line** matches *record*.**ut_line**. If it finds such a record, **pututline()** overwrites it with the contents of *record*; otherwise, it appends *record* onto the end of the logging file.

If all goes well, **pututline()** returns the address *record*. It returns NULL if the logging file had not been opened, or if it could not write *record* into the logging file.

By default, **getutid()** updates record in the logging file **/etc/utmp**. If you wish to manipulate another file, use the function **utmpname()**.

See Also

libc, utmp.h

putw() — STDIO Function (libc)

Write word into stream
#include <stdio.h>
int putw(*word***,** *fp***)**
int *word***; FILE** ******fp***;**

putw() writes *word* into the file stream to which *fp* points.

putw() differs from the related routine **putc()** in that **putw()** writes an **int**, whereas **putc()** writes a **char** that is promoted to an **int**.

By default, **putw()** returns the value written. If an error occurs, it returns EOF. You may need to call **ferror()** to distinguish this value from a genuine end-of-file flag.

See Also

ferror(), libc

Notes

Because **putw()** is implemented as a macro as well as a function, arguments with side effects may not work as expected. The bytes of *word* are written in the natural byte order of the machine.

pwd — Command

Print the name of the current directory
pwd

pwd prints the name of the directory that you are in.

See Also

cd, commands, ksh, sh

Notes

Under the Korn shell, **pwd** is an alias for the expression **print -r $PWD**.

pwd.h — Header File

Define password structure
#include <pwd.h>

The header file **pwd.h** defines the structure **passwd**, which is used to build COHERENT's password file. **passwd** is defined as follows:

```
struct passwd {
        char    *pw_name;       /* login user name */
        char    *pw_passwd;     /* login password */
        int     pw_uid;         /* login user id */
        int     pw_gid;         /* login group id */
        int     pw_quota;       /* file quota (unused) */
        char    *pw_comment;    /* comments (unused) */
        char    *pw_gecos;      /* (unused) */
        char    *pw_dir;        /* working directory */
        char    *pw_shell;      /* initial program */
};
```

For detailed descriptions of the above fields, see the entry for **passwd**.

See Also

endpwent(), getpwent(), getpwnam(), getpwuid(), header files, setpwent()
POSIX Standard, §9.2.2

qfind — Command

Quickly find all files with a given name
qfind [-adpv] *name* **...**
qfind [-bv] [-s*directory***]**

The command **qfind** prints the full path name of each file with a given *name*. It reads a prebuilt data base, for the sake of speed. This makes **qfind** much faster than **find** for locating a file; but it does mean that changes to the file system since the data base was last updated will not be reflected in what **qfind** prints.

The option **-b** tells **qfind** to build its data base in file **/usr/adm/qffiles**. By default, this data base names every file in your system. If you wish to suppress a directory, name it with the **-s** option. For example, to build the data base but suppress the directory **/usr/spool**, use the command:

```
qfind -b -s/usr/spool
```

This command excludes the contents of directory **/usr/spool** and all of its children from the **qfind** data base.

When invoked without the **-b** option, **qfind** reads its data base to find file names.

Normally, **qfind** prints the full path name of each file in the COHERENT system that ends with the given *name* (as it was when you last executed **qfind -b**.) With the **-d** option, **qfind** prints matching directories instead of files. With the **-a** option, **qfind** prints both matching files and matching directories.

Option **-p** specifies partial name matching. For example, **qfind -p foo** matches files **/src/foo.c** and **/doc/foo.r** as well as file **/usr/bin/foo.**

Finally, option **-v** tells **qfind** to print verbose output.

Files

/usr/adm/qffiles

See Also

commands, cron, find, whereis, which

Notes

Building the **qfind** data base with the **-b** option is slow, but it speeds finding files. You may find it convenient to use **cron** to execute **qfind -b** to rebuild the data base at night, or some other time when the machine is otherwise idle.

If you want to include all files in the data base, the superuser **root** must run **qfind -b**.

qpac — Command

Map the file system
qpac *raw_device*

Command **qpac** builds a map of the file system *raw_device*. It quits before it writes to or changes the file system. You can use this to examine how your file system is laid out.

See Also

commands, dpac, fmap, fsck, spac, upac

Notes

qpac is a link to the command **dpac**.

qpac was written by Randy Wright (rw@rwsys.wimsey.bc.ca).

qsort() — General Function (libc)

Sort arrays in memory
#include <stdlib.h>
void qsort(*data***,** *n***,** *size***,** *comp***)**
char **data***; **int** *n*, *size*; **int (***comp***)();**

qsort() is a generalized algorithm for sorting arrays of data in memory, using C. A. R. Hoare's "quicksort" algorithm. **qsort()** works with a sequential array of memory called *data*, which is divided into *n* parts of *size* bytes each. In practice, *data* is usually an array of pointers or structures, and *size* is the **sizeof** the pointer or structure. Each routine compares pairs of items and exchanges them as required. The user-supplied routine to which *comp* points performs the comparison. It is called repeatedly, as follows:

```
(*comp)(p1, p2)
char *p1, *p2;
```

Here, *p1* and *p2* each point to a block of *size* bytes in the *data* array. In practice, they are usually pointers to pointers or pointers to structures. The comparison routine must return a negative, zero, or positive result, depending on whether *p1* is logically less than, equal to, or greater than *p2*, respectively.

Example

For an example of this function, see the entry for **malloc()**.

See Also

libc, shellsort(), strcmp(), stdlib.h, strncmp()
The Art of Computer Programming, vol. 3
ANSI Standard, §7.10.5.2
POSIX Standard, §8.1

Notes

The COHERENT library also includes the sorting function **shellsort()**. These functions use different algorithms for sorting items; each algorithm has its strengths and weaknesses. In general, the quicksort algorithm is faster than the shellsort algorithm for large arrays, whereas the shellsort algorithm is faster for small arrays (say, 50 items or fewer). The quicksort algorithm also performs poorly on arrays with a small number of keys, e.g., an array of 1,000 items whose keys are all '7' and '8'.

To get around these limitations, the COHERENT implementation of **qsort()** has an adaptive algorithm that recognizes when the quicksort algorithm is performing badly, and calls **shellsort()** in its place.

quot — Command

Summarize file-system usage
quot [-c] [-f] [-n] [-t] *filesystem*

quot produces several different summaries about the ownership of files for each *filesystem* argument given. When no options are specified, **quot** produces a two-column listing that gives the amount of space used by each user, sorted in decreasing order of file space used; the first column gives the number of blocks used and the second gives the use name. Space is always given in blocks.

Options are available to modify the normal output or specify a completely different action.

quot recognizes the following options:

-c Give a three-column breakdown of files by size. The first column contains all file sizes, in increasing order. The second column gives the number of files of the size indicated in the first. The third gives a cumulative sum of the sizes of all files less than or equal to the current size.

-f Add an initial column that contains the number of files to the front of the normal output.

-n Takes as input a list of i-numbers and file names, one per line and sorted in ascending order by i-number; ignore all lines not in this form. The output is in two columns: the first gives the owner and the second contains the file name for each entry in the output. This conforms to usage with the following pipeline:

```
ncheck filesystem | sort +0n | quot -n filesystem
```

-t To the normal output, add a line that contains totals.

quot runs much faster with a raw device for *filesystem*.

Only the superuser **root** can run **quot**.

Files

/etc/passwd

See Also

ac, commands, ncheck, sort

Notes

Sparse files are recorded as if they had all of their blocks allocated.

raise() — General Function (libc)

Let a process send a signal to itself
#include <signal.h>
int raise(*signal***)**
int *signal***;**

raise() sends *signal* to the program that is currently being executed. If called from within a signal handler, the processing of this signal may be deferred until the signal handler exits.

Example

This example sets a signal, raises it itself, then allows the signal to be raised interactivly. Finally, it clears the signal and exits.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void gotcha(void);

void
setgotcha(void)
{
      if(signal(SIGINT, gotcha) == SIG_ERR) {
            printf("Couldn't set signal\n");
            abort();
      }
}

void
gotcha(void)
{
      char buf[10];

      printf("Do you want to quit this program? <y/n> ");
      fflush(stdout);
      gets(buf);

      if(tolower(buf[0]) == 'y')
            abort();

      setgotcha();
}

main(void)
{
      char buf[80];

      setgotcha();
      printf("Set signal; let's pretend we get one.\n");
      raise(SIGINT);

      printf("Returned from signal\n");
      printf("Try typing <ctrl-c> to signal <enter> to exit");
      fflush(stdout);
      gets(buf);
```

```
      if(signal(SIGINT, SIG_DFL) == SIG_ERR) {
            printf("Couldn't lower signal\n");
            abort();
      }

      printf("Signal lowered\n");
      exit(EXIT_SUCCESS);
}
```

See Also

libc, signal(), signal.h
ANSI Standard, §7.7.2.1

ram — Device Driver

Driver for manipulating RAM

The COHERENT **ram** devices let you allocate and use the random-access memory (RAM) of the computer system directly. A typical use is for a RAM disk, which is a COHERENT file system kept in memory rather than on a floppy disk or hard disk.

The COHERENT RAM device driver has major number 8. You can access it either as a block-special device or as a character-special device. The high-order bit of the minor number gives the RAM device number (0 or 1); as you can see, you can have no more than two RAM devices in memory at any one time. The low-order seven bits give the device's size in 64-kilobyte chunks.

The first call to **open()** on a RAM device with nonzero size (1 to 127) allocates memory for the device; **open()** fails if sufficient memory is not available. Accessing a RAM device with a minor number that specifies size zero frees the allocated memory, provided all earlier **open()** calls have been closed.

Initially, COHERENT includes two block-special devices for RAM disks: the 512-kilobyte device **/dev/ram0** (8, 8) and the 192-kilobyte device **/dev/ram1** (8, 131). It also includes the devices **/dev/ram0close** (8, 0) and **/dev/ram1close** (8, 128). You should resize the RAM devices to suit the amount of memory available on your system.

Examples

The following example formats and mounts a 512-kilobyte RAM disk on directory **/fast**.

```
mkdir /fast
/etc/mkfs /dev/ram0 1024
/etc/mount /dev/ram0 /fast
```

When the RAM disk is no longer needed, its allocated memory can be freed as follows:

```
/etc/umount /dev/ram0
cat /dev/null >/dev/rram0close
```

The next example replaces the default **/dev/ram0** with a one-megabyte device that contains a COHERENT file system. The minor number 16 specifies RAM device 0 and a size of one megabyte (i.e., 16 chunks of 64 kilobytes each). The new RAM device contains 2,048 blocks of 512 bytes each.

```
rm /dev/ram0
/etc/mknod /dev/ram0 b 8 16
/etc/mknod /dev/rram0 c 8 16
/etc/mkfs /dev/ram0 2048
chmod ugo=rw /dev/ram0
chmod ugo=rw /dev/rram0
```

The command **chmod** is necessary to make the new RAM drive accessible.

Files

/dev/ram*

See Also

compress, device drivers, fsck, mkfs, mount, ramdisk, umount, uncompress, zcat

Notes

Moving frequently used commands or files to a RAM disk can improve system performance substantially. However, the contents of a RAM device are lost if the system loses power, reboots, or crashes. Therefore, you should frequently back up files from the RAM disk to a more permanent medium.

If a RAM device uses most but not all available system memory, its **open()** call will succeed but subsequent commands may fail because insufficient memory remains for the system.

The COHERENT installation program **/etc/build** uses RAM device **/dev/ram1** as a RAM disk during installation. Commands **compress**, **uncompress**, **zcat**, and **fsck** sometimes use **/dev/ram1** as a temporary storage device. Users should avoid using **/dev/ram1** as a RAM disk because of these programs. In addition, users of **compress**, **uncompress**, and **zcat** may have to change the size of **/dev/ram1** from the default size of 192 to 512 kilobytes, to handle files compressed to 16 bits. The following script makes this change; note that it must be run by the superuser **root**:

```
cat /dev/null >/dev/rram1close
rm /dev/ram1 /dev/rram1
mknod /dev/ram1  b 8 136
mknod /dev/rram1 c 8 136
```

ramdisk — System Administration

Script to create a RAM-disk
/usr/bin/ramdisk

ramdisk is a script that creates a 500-kilobyte RAM disk that is accessed via device **/dev/ram0**.

To use **ramdisk** to create a RAM disk for you at boot-time, do the following:

1. Log in as the superuser **root**.
2. Type:

   ```
   touch /dev/ram0close
   ```

   This closes the RAM disk and removes it from memory.
3. Remake the ram disk as a smaller size device. As an example, we'll make one that is 64 kilobytes. Type the command:

   ```
   /etc/mknod /dev/ram0 b 8 1
   ```

   To break down this command:

   /etc/mknod
   This is the command that creates a special file (e.g., a block-special file) through which a device like a printer or RAM is accessed.

   /dev/ram0
   The directory path and name of your RAM disk.

   b This argument tells **mknod** to build a block-special file. Every device like a printer, floppy drive, COM port, or RAM drives, are considered a "block special file."

   8 This is the major device number for a RAM drive. All major-device numbers are listed in the Lexicon entry for "device drivers."

   1 This is the minor device number of your new **ram0.** This shows that the **ram0** you are building will be 64 kilobytes in size. If the minor device number would have been '2', then the size of **ram0** would have been two times 64, or 128 kilobytes. Each increment in the minor-device number is equal to an additional 64 kilobytes for the RAM device. A minor device of 16 multiplied by 64 kilobytes would equal a one megabyte size RAM drive.
4. Next, make a file system in **ram0**:

   ```
   /etc/mkfs /dev/ram0 128
   ```

   The number "128" is exactly twice the memory size, in this case 64 kilobytes. Whatever size memory you choose to allocate to a RAM device, the block size you specify in the **mkfs** command will be double. A one-megabyte RAM device for example would have 2,048 blocks.

5. Your new RAM disk is now ready to be mounted. Typically, you would mount ram0 in a directory named **fast** or some other unique name, so to mount, type;

```
/etc/mkdir /fast
/etc/mount /dev/ram0 /fast
```

If **/fast** already exists, do not create it.

Once you have created your RAM disk, you should load commonly used utilities into it.

If you wish to create a RAM disk automatically whenever you boot COHERENT, un-comment and edit the appropriate lines in file **/etc/rc**.

See Also

Administering COHERENT, ram, rc

Notes

This script only works in machines that have sufficient memory.

rand() — Random-Number Function (libc)

Generate pseudo-random numbers
#include <stdlib.h>
int rand()

rand() generates a set of pseudo-random numbers. It returns integers in the range 0 to 32,767, and purportedly has a period of 2^32. **rand()** will always return the same series of random numbers unless you first call the function **srand()** to change **rand()**'s *seed*, or beginning-point.

Example

The following example uses **rand()** to implement the "Let's Make a Deal" game of probability described by Massimo Piattelli-Palmarini in the March/April 1991 issue of *Bostonia* magazine. In brief, an investigator places a dollar bill into one of three boxes. A subject enters the room and guesses which box holds the bill. The investigator then opens one of the two unselected boxes (one that is always empty), shows it to the subject, then offers the subject a choice: either stand pat with the box he has selected, or switch to the other non-selected box. The laws of probability state that the subject should always switch from the box he has selected; this example program tests that hypothesis.

```
#include <stdio.h>
#include <time.h>

main()
{
      int box[3], win, i, j;

      srand(time(NULL));

      /* Test 1:  the subject always stands pat.  For the sake of simplicity,
       * the subject always chooses box 0. */
      for (i = 0, win = 0; i < 1500; i++) {
            for (j = 0; j < 3; j++)
                  box[j] = 0;

            box[rand()%3]++;

            if (box[0])
                  win++;
      }
      printf("Test 1, always stand pat:  1500 iterations, %d winners\n", win);

      /* Test 2:  the subject always switches boxes. */
      for (i = 0, win = 0; i < 1500; i++) {
            for (j = 0; j < 3; j++)
                  box[j] = 0;

            box[rand()%3]++;
```

```
            /* if box 2 is empty, pick box 1 */
            if (!box[2])
                  win += box[1];
            else
                  win += box[2];
      }
      printf("Test 2, always switch:  1500 iterations, %d winners\n", win);

      /* Test 3:  the subject switches boxes randomly. */
      for (i = 0, win = 0; i < 1500; i++) {
            for (j = 0; j < 3; j++)
                  box[j] = 0;

            box[rand()%3]++;

            /* if box 2 is empty, pick box 1 */
            if (!box[2]) {
                  if (rand()%2)
                        win += box[1];
                  else
                        win += box[0];
            } else {
                  if (rand()%2)
                        win += box[2];
                  else
                        win += box[0];
            }
      }
      printf("Test 3, randomly switch:  1500 iterations, %d winners\n", win);
}
```

See Also

libc, RAND_MAX, srand(), stdlib.h
The Art of Computer Programming, vol. 2
ANSI Standard, §7.10.2.1
POSIX Standard, §8.1

Notes

This function cannot be used with any of the "rand48" functions. For an overview of these functions, see the entry for **srand48()**.

RAND_MAX — Manifest Constant

Largest size of a pseudo-random number
#include <stdlib.h>

RAND_MAX is a manifest constant that is defined in the header **stdlib.h**. It indicates the largest pseudo-random number that can be returned by the function **rand()**.

Example

For an example of using this manifest constant in a program, see **rand()**.

See Also

manifest constant, rand(), stdlib.h
ANSI Standard, §7.10

random() — Sockets Function (libsocket)

Return a random number
int random();

The function **random()** returns a random number. It is a synonym for **rand()**.

See Also

libsocket, rand()

random access — Definition

In the context of computing, **random access** means that an entity, such as memory, can be accessed at any point, not just at the beginning. This means that all points within memory can be accessed equally quickly. This contrasts with *sequential access*, in which entities must be accessed in a particular order, so that some entities take longer to access than do others.

A tape drive is an example of a sequential access device, i.e., the order in which data are read is dictated by the order in which they stream past the tape head. Random-access memory (RAM) is an example of random access. Hard disks and floppy disks combine elements of random access and sequential access.

RAM, which usually consists of semiconductor integrated circuits, is also strictly random access. In this regard, the term "RAM" is slightly misleading; a more accurate name would be "read/write memory", to contrast RAM with read-only memory (ROM), which is also random access memory.

See Also

read-only memory, Programming COHERENT

ranlib — Command

Create index for object library
ranlib *library ...*

The **ranlib** is a "directory" that appears at the beginning of each library. It contains the name of each global symbol (i.e., function name) that appears within the library, and a pointer to the module in which that symbol is defined. Thus, the ranlib eliminates the need for the linker to search the entire library sequentially to find a given global symbol, which speeds up linking noticeably.

If the date on the library file is later than that in the ranlib header, the linker will ignore the ranlib and perform a sequential search through the library; the linker will also send the warning message

```
Outdated ranlib
```

to the standard error device. This is done to prevent the accidental use of an outdated ranlib, which could be disastrous.

The command **ranlib** creates a ranlib header for an archive. If the header already exists, **ranlib** updates it.

Files

__.SYMDEF — Index module

See Also

ar, ar.h, commands, ld

Diagnostics

ranlib issues appropriate messages for I/O errors or bad format files. It does not rewrite a library until the last possible moment, so the library is usually unchanged in case of error. **ranlib** processes each library independently. The exit status is the number of libraries in which errors were encountered.

ranlib is a link to the archiver **ar**.

rc — System Administration

Perform standard maintenance chores
/etc/rc

The shell script **/etc/rc** is executed by the **init** process when the COHERENT system enters multi-user mode. The commands in **rc** do such things as set the local time zone and initialize file **/usr/adm/wtmp**, which holds records of user logins.

See Also

Administering COHERENT, brc, init

read-only memory — Definition

As its name suggests, **read-only memory,** or ROM, is memory that can be read but not overwritten. It most often is used to store material that is used frequently or in key situations, such as a language interpreter or a boot routine.

See Also

Programming COHERENT, random access

read — Command

Assign values to shell variables
read *name ...*

read reads a line from the standard input. It assigns each token of the input to the corresponding shell variable *name*. If the input contains fewer tokens than the number of names specified, **read** assigns the null string to each extra variable. If the input contains more tokens than the number of names specified, **read** assigns the last *name* in the list the remainder of the input.

read normally returns an exit status of zero. If it encounters end of file or is interrupted while reading the standard input, it returns one.

The shell executes **read** directly.

Example

The command

```
read foo bar baz
hello how are you
```

parses the line "hello how are you" and assigns the tokens to, respectively, the shell variables **foo**, **bar**, and **baz**. If you further type

```
echo $foo
echo $bar
echo $baz
```

you will see:

```
hello
how
are you
```

See Also

commands, ksh, sh

read() — System Call (libc)

Read from a file
#include <unistd.h>
int read(*fd***,** *buffer***,** *n***)**
int *fd***; char** **buffer***; **int** n;*

read() reads up to *n* bytes of data from the file descriptor *fd* and writes them into *buffer*. The amount of data actually read may be less than that requested if **read()** detects **EOF**. The data are read beginning at the current seek position in the file, which was set by the most recently executed **read()** or **lseek()** routine. **read()** advances the seek pointer by the number of characters read.

If all goes well, **read()** returns the number of bytes read; thus, zero bytes signals the end of the file. It returns -1 if an error occurs, e.g., *fd* does not describe an open file, or if *buffer* contains an illegal address.

Example

For an example of how to use this function, see the entry for **open()**.

See Also

libc, unistd.h
POSIX Standard, §6.4.1

Notes

read() is a low-level call that passes data directly to COHERENT. It should not be mixed with the STDIO routines **fread()**, **fwrite()**, or **fopen()**.

readdir() — General Function (libc)

Read a directory stream
#include <sys/types.h>
#include <dirent.h>
struct dirent ***readdir(***dirp**)**
DIR **dirp**;

The COHERENT function **readdir()** is one of a set of COHERENT routines that manipulate directories in a device-independent manner. It reads the directory stream pointed to by *dirp* and returns information about the next active entry within the stream. It does not report on inactive entries.

readdir() returns a pointer to a structure of type **dirent**, which contains information about the next active entry within the stream. The internal structure may be overwritten by another operation on the same directory stream. The amount of memory needed to hold a copy of the internal structure is given by the value of a macro, **DIRENTSIZ(strlen(direntp->d_name))**, not by **sizeof(struct dirent)** as one might expect.

readdir() returns NULL if it has reached the end of the directory, has detected an invalid location within the directory, or if an error occurs while it is reading the directory. If an error occurs, **readdir()** exits and sets **errno** to an appropriate value.

Example

For an example of this function, see the Lexicon entry for **opendir()**.

See Also

closedir(), dirent.h, getdents(), libc, opendir(), rewinddir(), seekdir(), telldir()
POSIX Standard, §5.1.2

Notes

The **dirent** routines buffer directories; and because directory entries can appear and disappear as other users manipulate the directory, your application should continually rescan a directory to keep an accurate picture of its active entries.

The COHERENT implementation of the **dirent** routines was written by D. Gwynn.

readline() — Editing Function (libedit)

Read and edit a line of input
char ***readline(***prompt**)**
char ***prompt**;

The function **readline()** displays on the standard output the text to which *prompt* points, then accepts what the user types. It lets the user type simple, EMACS-style commands to edit what she has typed; when the user types (¢), **readline()** returns the line of text with the trailing newline removed.

readline() returns a pointer to the newly entered line. This return value can be passed to the function **add_history()**, which adds it to an internal "history" buffer. The user can use a command within **readline()** recall a saved line, re-edit it, and re-submit it.

readline() returns NULL when the user types EOF, or if it cannot allocate space for the line of input. Otherwise, it returns the address of the edited string that the user input.

Editing Commands

readline() provides a simple, EMACS-like editing interface. You can type control characters or escape sequences to edit a line before it is sent to the calling program, much like the EMACS editing feature of the Korn shell.

readline() recognizes the following editing commands:

<ctrl-A> Move the cursor to the beginning of the line.
<ctrl-B> Move the cursor one character to the left (backwards).

<ctrl-D> Delete the character under which the cursor is positioned (the "current character").
<ctrl-E> Move the cursor to end of line.
<ctrl-F> Move the cursor one character to the right (forwards).
<ctrl-G> Ring the bell.
<ctrl-H> Delete the character to the left of the cursor. Note that **<ctrl-H>** is the character that normally is output by the **<backspace>** key.
<ctrl-I> Complete file name. Note that **<ctrl-I>** is the character that normally is output by the **<tab>** key.
<ctrl-J> Submit the line for processing. Note that **<ctrl-J>** is the character that normally is output by the (¢) key.
<ctrl-K> Kill all text from the cursor to end of line.
<ctrl-L> Redisplay the line.
<ctrl-M> Submit the line for processing. Note that on some systems, **<ctrl-M>** is output by the (¢) key.
<ctrl-N> Get the next line from the history buffer.
<ctrl-P> Get the previous line from the history buffer.
<ctrl-R> Search backwards through the history buffer for a given string.
<ctrl-T> Transpose the character over the cursor with the character to its left.
<ctrl-V> Insert next character into the line, even if it is a control character. Note that under MicroEMACS, this command is bound to **<ctrl-Q>**.
<ctrl-W> Kill (wipe) all text from the cursor to the mark.
<ctrl-X><ctrl-X>
Move the cursor from current position to the mark; reset the mark at the previous position of the cursor.
<ctrl-Y> Yank back the most recently killed text.
<ctrl-]>*c* Move the cursor forward to next character *c*.
<ctrl-?> Delete the character under which the cursor is positioned. This command is identical with **<ctrl-D>**. Note that **<ctrl-?>** is the character that normally is output by the **<del>** key.
<ctrl-[> Begin an escape sequence. Note that **<ctrl-[>** is the character that normally is output by the **<esc>** key.
<esc><ctrl-H>
Delete the previous word (the word to the left of the cursor). A word is delineated by white space.
<esc><del> Delete the current word — that is, from the cursor to the end of the word as delineated by white space or the end of the line.
<esc><space>
Set the mark.
<esc>. Get the last (or *n*'th) word from previous line.
<esc>? Show possible completions. This feature is detailed below.
<esc>< Move the cursor to the beginning of the history buffer.
<esc>> Move the cursor to the end of the history buffer.
<esc>B Move the cursor backwards (to the left) by one word.
<esc>D Delete the word under which the cursor is positioned.
<esc>F Move the cursor forward (to the right) by one word.
<esc>L Make the current word lower case.
<esc>M Toggle displaying eight-bit characters normally (meta-mode), or displaying them prefixed with the string **M-**. In the meta-mode, you can generate characters with the top bit set by pressing the **<alt>** key with an alphanumeric key; this is interpreted the same as **<esc><***key***>**.
<esc>U Make the current word upper case.
<esc>Y Yank back the most recently killed text.
<esc>V Show the version of the library **libedit.a**.
<esc>W Make yankable all text from the cursor to the mark.
<esc>*n* Set the argument to integer *n*.
<esc>*C* Read input from environment variable *_C_*, where *C* is an upper-case letter.

Most editing commands can be given an argument *n*, where *n* is an integer. To enter a numeric argument, type **<esc>**, the number, and then the command to execute. For example,

```
<esc> 4 <ctrl-F>
```

moves the cursor four characters forward.

Note that you can type an editing command on the line of input, not just at the beginning. Likewise, you can type (¢) to submit a line for processing, regardless of where on the line the cursor is positioned.

readline() has a modest macro facility. If you type **<esc>** followed by an upper-case letter, then **readline()** reads the contents of environment variable *_C_* as if you had typed them at the keyboard.

readline() also can complete a file name. For example, suppose that the root directory contains the following files:

```
coherent
coherent.old
```

If you type

```
rm /c
```

into **readline()** and then press the **<tab>** key, **readline()** completes as much of the name as it can — in this case, by adding **oherent**. Because the name is not unique, **readline()** then beeps. If you press **<esc>?**, **readline()** displays the two choices. If you then enter a tie-break character (in this case, '.'), followed by the **<tab>** character, **readline()** completes the file name for you.

Using Line Editing

To include **readline()** in your program, simply call it as you do any other function. You must link the library **libedit.a** into your program.

Example

The following brief example lets you enter a line and edit it, and then displays it.

```
#include <stdlib.h>

extern char *readline();
extern void add_history();

int main(ac, av)
int ac; char *av[];
{
      char *p;

      while ((p = readline ("Enter a line:")) != NULL) {
            (void) printf ("%s\n", p);
            add_history (p);
            free (p);
      }
      return 0;
}
```

See Also

add_history(), libedit

Notes

readline() calls **malloc()** to allocate space for the text that the user enters. Therefore, an application must call **free()** to free this space when it has finished with it.

readline() cannot handle lines longer than 80 characters.

The original manual page was written by David W. Sanderson <dws@ssec.wisc.edu>.

readonly — Command

Mark a shell variable as read only
readonly

Mark each *variable* as a read-only shell variable. The shell will not permit subsequent assignments to a readonly variable. With no arguments, **readonly** prints the name and value of each read-only variable.

See Also

commands, ksh, sh

readonly — C Keyword

Storage class

readonly is a C keyword that modifies data declarations. As its name implies, the **readonly** modifier declares that data are to be read only; this helps protect key data against casual modification by the user or another programmer.

LEXICON

See Also

C keywords

Notes

The ANSI Standard eliminates this keyword.

realloc() — General Function (libc)

Reallocate dynamic memory
#include <stdlib.h>
char *realloc(*ptr***,** *size***)**
char ****ptr*; unsigned** *size***;**

realloc() helps you manage a program's arena. It returns a block of *size* bytes that holds the contents of the old block, up to the smaller of the old and new sizes. **realloc()** tries to return the same block, truncated or extended; if *size* is smaller than the size of the old block, **realloc()** will return the same *ptr*.

If *ptr* is set to NULL, **realloc()** behaves like **malloc()**.

Example

For an example of this function, see the entry for **calloc()**.

See Also

alloca(), arena, calloc(), free(), libc, malloc(), memok(), setbuf(), stdlib.h
ANSI Standard, §7.10.3.4
POSIX Standard, §8.1

Diagnostics

realloc() returns NULL if insufficient memory is available. It prints a message and calls **abort()** if it discovers that the arena has been corrupted, which most often occurs by storing past the bounds of an allocated block. **realloc()** behaves unpredictably if handed an incorrect *ptr*.

reboot — Command

Reboot the COHERENT system
/etc/reboot [-p]

reboot reboots the COHERENT system. The option **-p** prompts the user if she really wishes to reboot before executing the reboot.

reboot can be run only by the superuser **root**.

The COHERENT system can be rebooted only from the console. *It should be rebooted only while in single-user mode. Failure to return to single-user mode before rebooting could damage the* COHERENT *file system and destroy data.*

See Also

commands, shutdown

Notes

No message is broadcast unless the command **shutdown** had been executed before invoking **reboot**.

recursion — Definition

Recursion is the technique by which a program or function calls itself. Because under C, all variables in a function function have local scope; therefore, when a function calls itself, it in effect recreates itself but with a fresh copy of each of its variables. The "states" of the previous call or calls to that function are stored on the stack, and are not modified when the function calls itself.

Recursion is a useful way to loop through a complex procedure. Be careful, however, that you do not lose track of how the number of times you have called a given function; and be careful not to pile the stack too high, or you may have problems.

Example

The following program demonstrates recursion. In it, the function **recur()** calls itself ten times.

```
#include <stdio.h>
```

```
main()
{
      printf("Before recursion ... \n");
      recur(1);
      printf("After recursion ... \n");
}

recur(level)
int level;
{
      printf("Entering call to recur() number %d\n", level);

      if (level < 10)
            recur(level+1);

      printf("Leaving call to recur() number %d\n", level);
}
```

See Also

programming COHERENT

recv() — Sockets Function (libsocket)

Receive a message from a connected socket
#include <sys/types.h>
#include <sys/socket.h>
int recv(*socket***,** *buffer***,** *length***,** *flags***)**
int *socket***;**
char **buffer***;*
int *length***,** *flags***;**

The function **recv()** receives messages from a connected socket.

socket is the socket from which the messages are received. It must have been created by the function **socket()**, and connected with the function **connect()**. *buffer* points to the chunk of memory into which the message is to be written; *length* gives the amount of allocated memory to which *buffer* points.

flags ORs together either or both of the following flags:

MSG_OOB
Read any out-of-band data present on *socket*, rather than the regular, in-band data.

MSG_PEEK
"Peek" at the data present on the socket: the data are copied but not erased from the socket, so another call to **recv()** or **recvfrom()** retrieves the same data.

If all goes well, **recv()** returns the number of bytes it read from *socket*. If something went wrong, it returns -1 and sets **errno** to one of the following values:

EAGAIN
If no message is queued at *socket*, **recv()** normally waits for a message to arrive (which is a blocking operation). *socket*, however, is marked as non-blocking.

EBADF *socket* does not identify a valid socket.

EINTR A signal interrupted **recv()** before it could receive any data.

ENOMEM
Insufficient user memory was available to complete the operation.

ENOTSOCK
socket describes a file, not a socket.

See Also

connect(), libsocket, recvfrom(), send(), socket()

recvfrom() — Sockets Function (libsocket)

Receive a message from a socket
#include <sys/compat.h>
#include <sys/socket.h>
#include "socketvar.h"
int recvfrom (*socket***,** *buffer***,** *length***,** *flags***,** *address***,** *addrlen***)**
int *socket***; char** **buffer***; **int** *length***; int** *flags***;**
sockaddr_t **from***; **int** **alen***;

recvfrom() receives messages from another socket. Unlike the related function **recv()**, **recvfrom()** receives data regardless of whether the socket is connected or not.

socket is the file descriptor of the socket from which data are to be received. It may or may not be connected. *buffer* is the chunk of memory in user space into which the data are to be written; it is *length* bytes long. If a received message is longer than *length* bytes, excess bytes can be discarded, depending on the type of socket from which the message is received. If *from* is not NULL, **recvfrom()** initializes it to the the source address of the message. It initializes *alen* to the size of the buffer associated with *address*, and modifies it upon return to the size of the address stored there.

If no messages are waiting at *socket*, **recvfrom()** waits for a message to arrive, unless the socket is nonblocking. In this case, it returns -1 and sets **errno**, as described below.

flags ORs together either or both of the following flags:

MSG_OOB
Read any out-of-band data present on *socket*, rather than the regular, in-band data.

MSG_PEEK
"Peek" at the data present on *socket*. The data are returned but remain on *socket*; therefore, another call to **recvfrom()** or **recv()** retrieves the same data.

If all goes well, **recvfrom()** returns the number of bytes received. If an error occurs, it returns -1 and sets **errno** to one of the following values:

EBADF *socket* is an invalid descriptor.

ENOTSOCK
socket is the descriptor of a file, not a socket.

EINTR The operation was interrupted by delivery of a signal before any data was available to be received.

EAGAIN
socket is marked non-blocking, but the requested operation would block.

ENOMEM
Too little user memory is available to complete the operation.

See Also

libsocket, recv()

Notes

At present, the COHERENT implementation of **recvfrom()** always behaves as if *address* were initialized to NULL.

ref — Command

Display a C function header
ref *function*

ref looks up the function header of *function* in any of a series of reference files built by the command **ctags**. It is used by the **elvis** editor's **<shift-K>** command. This command checks the file **refs** in the current directory.

See Also

commands, ctags, elvis

Notes

Release 1.7 of **ref** tells you which source file it is looking in. It does not show argument lines for macros, because it now knows that they do not have any.

regcomp() — Regular-Expression Function (libc)

Compile a regular expression into a structure
#include <regexp.h>
regexp ***regcomp(***_expression_**)**
char **_expression_**;**

Function **regcomp()** compiles *expression* into a structure of type **regexp**, and returns a pointer to it. For details on the structure **regexp**, see the Lexicon entry for **regexp.h**. *expression* must be a regular expression; the rules that define a regular expression are described in the Lexicon entry **regexp.h**.

See Also

libc, regexp.h

Notes

regcomp() calls **malloc()** to allocate the memory that holds the structure it creates. To free this structure, your program must call **free()**.

regerror() — Regular-Expression Function (libc)

Return an error message from a regular-expression function
#include <regexp.h>
void regerror(*message***)**
char **_message_**;**

Function **regerror()** is the function that is called by default when an error is detected in any of the regular-expression functions **regcomp()**, **regexec()**, or **regsub()**. It prints *message* onto the standard-error device, plus some text to indicate whence the message originates; then calls **exit()** to abort the program in which the error occurred.

You are not obliged to use **regerror()** to report an error with a regular-expression function. You can substitute another function of your choosing, should you prefer.

See Also

libc, regexp.h

regexec() — Regular-Expression Function (libc)

Compare a string with a regular expression
#include <regexp.h>
int regexec(*expression***,** *string***)**
regexp ***_expression_**;**
char ***_string_**;**

Function **regexec()** compares *string* with *expression*, which is a regular expression compiled by function **regcomp()**. If *string* matches *expression*, **regexec()** returns one; otherwise, it returns zero and readjusts the sub-string pointers within *expression*.

For details on the structure **regexp** and its sub-string pointers, see the Lexicon entry for **regexp.h**. In brief, if **regexec()** successfully matches *string* with *expression*, it initializes arrays **startp[]** and **endp[]** within *expression*. Each **startp/endp** pair point to one substring within *string*: the **startp** element points to the first character of the substring and the **endp** element points to the first character that follows the substring. The pair **startp[0]/endp[0]** points to the substring of *string* that matched the whole of *expression*. The other pairs point to the substrings within *string* that matched parenthesized expressions within *expression*, with parenthesized expressions numbered in left-to-right order of their opening parentheses.

See Also

libc, regexp.h

regexp.h — Header File

Header file for regular-expression functions
#include <regexp.h>

Header file **<regexp.h>** is used with regular-expression function **regcomp()**, **regexec()**, and **regsub()**. These functions manipulate a regular expression, which is stored in structure **regexp**. **<regexp.h>** defines this structure as follows:

```
typedef struct regexp {
      char *startp[NSUBEXP];
      char *endp[NSUBEXP];
      char regstart;
      char reganch;
      char *regmust;
      int regmlen;
      char program[1];
} regexp;
```

Fields **regstart** through **program** are used internally, and should not be manipulated by a user's program. Fields **startp[]** and **endp[]** are arrays of pointers to sub-strings within the expression. For details on how these pointers are used, see the Lexicon entry for **regexec()**. **NSUBEXP** gives the number of sub-strings that can be addressed at one time; as of this writing, it is set to ten.

Syntax of a Regular Expression

The following describes the rules with which the **regexp** functions define a regular expression.

A regular expression consists of zero or more *branches*. Branches are separated from each other by a pipe character '|'. A string matches an expression when it matches any branch within the expression.

A branch, in turn, consists of zero or more *pieces*, which are concatenated. Each piece is a string, or *atom*, which can be followed by '*', '+', or '?'. An atom followed by '*' can be matched with a sequence of zero or more matches of the atom. An atom followed by '+' can be matched with a sequence of one or more matches of the atom. An atom followed by '?' can be matched with either the atom or the null string.

An atom, in turn, is built from the following:

(*expression***)**
A regular expression between parentheses This matches a match for the regular expression.

[*string***]** Match any character within *string*. If string contains a hyphen '-', this represents a range of characters. For example, "0-9" represents all digits; or "a-z" represents all lower-case characters. To include a literal '-' within *string*, make it the first or last character within *string*. To include a literal ']' in the sequence, make it the first character, after a possible '^'.

[^*string***]**
Match any character that is *not* in *string*.
a *range* (see below), '.'

^ Match the null string at the beginning of the input string.

$ Match the null string at the end of the input string.

*****c Match the single character *c* literally; ignore any special significance that *c* might have.

Ambiguity

A string can match more than one part of an regular expression. The following rules describe how to choose which part to match.

The basic rule is that if a regular expression could match two parts of a string, it matches the one that begins earlier.

If both parts begin in the same place but match different lengths of the expression, or match the same length in different ways, life gets messier, as follows.

In general, the possibilities in a list of branches are considered in left-to-right order, the possibilities for '*', '+', and '?' are considered longest-first, nested constructs are considered from the outermost in, and concatenated constructs are considered leftmost-first. The match that is chosen is the one that uses the earliest possibility in

the first choice that has to be made. If there is more than one choice, the next will be made in the same manner (earliest possibility) subject to the decision on the first choice.

For example, "(ab|a)b*c" could match "abc" in one of two ways. The first choice is between "ab" and 'a'; since "ab" is earlier, and lead to a successful overall match, it is chosen. Since the 'b' is already spoken for, the "b*" must match its last possibility — the empty string — because it must respect the earlier choice.

In the particular case where no '|'s are present and there is only one '*', '+', or '?', the net effect is that the longest possible match will be chosen. So "ab*", presented with "xabbbby", will match "abbbb". Note that if "ab*" is tried against "xabyabbbz", it will match "ab" just after 'x', due to the begins-earliest rule. In effect, the decision on where to start the match is the first choice to be made, hence subsequent choices must respect it even if this leads them to less-preferred alternatives.

See Also

header files, regcomp(), regerror(), regexec(), regsub()

Notes

The code used for the **regexp()** was written by Harry Spencer at the University of Toronto. It is copyright © 1986 by the University of Toronto. These routines are intended to be compatible with the Bell System-8 **regexp()** but are not derived from Bell code. The above description of regular expressions is derived from the manual page written by Harry Spencer.

register — C Keyword

Storage class

register is a C keyword that declares a class of data storage. A variable so declared may be stored in a register, which may increase the speed with which it is read by a program.

See Also

auto, C keywords, extern, register variable, static
ANSI Standard, §6.5.1

register variable — Definition

register is a C storage class. A **register** declaration tells the compiler to try to keep the defined local data item in a machine register. Under COHERENT C, the **int foo** can be declared to be a register variable with the following statement:

```
register int foo;
```

The COHERENT C compiler makes three registers available for variables: **ESI**, **EDI**, and **EBX**. Subsequent **register** declarations are ignored, because no registers are left to hold them.

By definition of the C language, registers have no addresses, so you cannot pass the address of register variable as an argument to a function. For example, the following code will generate an error message when compiled:

```
register int i;
  . . .
dosomething(&i);  /* WRONG */
```

This rule applies whether or not the variable is actually kept in a register.

Placing heavily-used local variables into registers often improves performance, but in some cases declaring **register** variables can degrade performance somewhat.

See Also

auto, extern, Programming COHERENT, static, storage class

regsub() — Regular-Expression Function (libc)

Use regular expression to build a string
"#include <regexp.h>"
regsub(*expression***,** *source***,** *dest***)**
regexp **expression***;*
char **source***,* **dest***;*

Function **regsub()** builds a string from a string and a regular expression.

source is the source string that is being interpreted. *expression* is the regular expression through which *source* is being interpreted; it must have been built by a call to **regcomp()**. Before you call **regsub()**, you must first have called **regexec()** to compare them and initialize the sub-string pointers within *expression*.

dest points to the memory into which **regsub()** writes its substituted string. It replaces each instance of '&' within *source* with the substring indicated by **startp[0]** and **endp[0]**. It also replaces each instance of '*n*', where *n* is a digit, with the substring **startp**[*n*] and **endp**[*n*].

For details on how these pointers are initialized, see the Lexicon entry for **regsub()**. For more details on the structure **regexp**, see the Lexicon entry for **regexp.h**. The rules that describe a regular expression also appear in function **regexp.h**.

See Also

libc, regexp.h

remove() — General Function (libc)

Remove a file
#include <stdio.h>
int
remove(*filename***)**
const char **filename***;*

remove() breaks the link between between *filename* and the actual file that it represents. In effect, it removes a file. Thereafter, any attempt to use *filename* to open that file will fail. It is equivalent to the system call **unlink()**.

remove() will remove a file that is currently open. **remove()** returns zero if it could remove *filename*, and nonzero if it could not.

Example

This example removes the file named on the command line.

```
#include <stdio.h>
#include <stdlib.h>

main(argc,argv)
int argc, char *argv[])
{
      if(argc != 1) {
            fprintf(stderr, "usage: remove filename\n");
            exit(EXIT_FAILURE);
      }

      if(remove(argv[1])) {
            perror("remove failed");
            exit(EXIT_FAILURE);
      }
      return(EXIT_SUCCESS);
}
```

See Also

libc, unlink()
ANSI Standard, §7.9.4.1
POSIX Standard, §8.1

rename() — System Call (libc)

Rename a file
#include <stdio.h>
int rename(*old***;** *new***)**
char **old***,* **new***;*

The COHERENT system call **rename()** changes the name of a file, from the name pointed to by *old* to that pointed to by *new*. Both *old* and *new* must point to a valid file name. If *new* names a file that already exists, the old file is replaced by the file being renamed.

rename() returns zero if it could rename *old*, and nonzero if it could not. If **rename()** could not rename *old*, its name remains unchanged.

Example

This example renames the file named in the first command-line argument to the name given in the second argument.

```
#include <stdio.h>
#include <stdlib.h>

main(argc, argv)
int argc; char *argv[];
{
      if (argc != 3) {
            fprintf(stderr, "usage: rename from to\n");
            exit(EXIT_FAILURE);
      }

      if(rename(argv[1], argv[2])) {
            perror("rename failed");
            exit(EXIT_FAILURE);
      }
      exit(EXIT_SUCCESS);
}
```

See Also

libc, link(), stdio.h, unlink()
ANSI Standard, §7.9.4.2
POSIX Standard, §5.5.3

Notes

The ANSI Standard states that **rename()** fails if *old* is open, or if its contents must be copied in order to rename it. Under COHERENT, it also fails if *new* is already open.

reprint — Command

Reprint a spooled print job
reprint [*job* [*page* **[***page***]]]**

The command **reprint** reprints each spooled *job*, where *job* identifies a job spooled with the command **lp**. If you do not specify a *job*, **reprint** reprints the job that you spooled most recently.

If you specify a *page*, **reprint** will attempt to reprint the document from that page to its end. If you specify two *page*s, **reprint** will attempt to reprint the document from the first *page* to the second.

Note that the printer daemon **lpsched** identifies pages by counting lines of input, so this feature works only with straight text. It does *not* work correctly with "cooked" input, such as files of PostScript or PCL.

See Also

commands, lp, printer

Notes

You should be *very* careful that jobs to print sensitive information — e.g., the payroll checks or your resume — do not linger in spool directory where other users can reprint them. For information on resetting a job's lifetime, see the Lexicon entries for **chreq**, **printer**, and **MLP_LIFE**. The article **controls** describes how to change the default life expectancies for spooled jobs.

resetterm() — terminfo Function

Reset the terminal to its previous settings
#include <curses.h>
resetterm()

COHERENT comes with a set of functions that let you use **terminfo** descriptions to manipulate a terminal. **resetterm()** restores the terminal to the condition it was in when before the current program began to manipulate its settings. Your program should call **resetterm()** before it calls **system()** or **exit()**.

See Also

curses.h, fixterm(), terminfo

restor — Command

Restore file system
restor *command* **[***dump_device***] [***filesystem***] [***file ...***]**

restor copies to the hard disk one or more files from floppy disks or tapes written by the command **dump**.

restor recognizes the following *commands*:

r Mass restore both full and incremental dump disks/tapes into the *filesystem*. The target file system must have enough data blocks and i-nodes to hold the dump.

The mass restoration is performed in three phases. In phase 1, **restor** clears all i-nodes that were either clear at dump time or are going to be restored. Any allocated blocks are released. Second, it restores all files on the disk. The i-numbering is preserved; however, data blocks are allocated in the standard fashion. Third, a pass is made over the i-nodes and the list of free i-nodes in the superblock is updated.

Restoration begins immediately, using the currently mounted disk or tape.

R Like the **r** command, except that it pauses to ask for numbers of disks or reels.

t Read the header from the dump. Display the date the dump was written and the "dump since" date that produced the dump.

x Extract each *file* from the dump and restore it to the hard disk. All file names are absolute path names starting at the root of the dump (the first directory dumped, which is always the root directory of the file system). A numeric file name is taken to be an i-number on the dumped file system, permitting restore by i-number.

restor looks up each *argument* file in the directories of the dumped file system and prints out each name and associated i-number. **restor** extracts the files from the currently mounted dump disk or tape, and writes the extracted files into the current directory. Extracted files are named after their i-numbers.

X Like command x, except that before it begins, it asks you for the number of the disk (or the reel number of the dump tape). It continues asking for dump disks until all files have been extracted or you types **<ctrl-D>**.

Each of the above commands can be modified either or both of the following modifiers:

f Tell **restor** to take the next *argument* as the path name of the dump device (floppy-disk drive or tape drive). If the **f** modifier is not specified, **restor** uses the device **/dev/dump**.

v Verbose output. Tell **restor** to print a step-by-step trace of its actions when restoring an entire file system. This is for discovering what went wrong when a mass restore runs into trouble.

Restoring from a Damaged Medium

As noted below, **dump** requires that its output be written to disks or tapes that are free of defects. Restoring a file system from a damaged medium is difficult and is not associated with a high probability of success; if, however, you must try to do so, the following directions will give you a fighting chance of restoring your data.

1. Use the command **fdformat** to format a blank disk. Use the command **badscan** to examine it for bad sectors; if it does have bad sectors, put it aside and try another.
2. Use the command **dd** to copy the bad disk to directory **/tmp***foo1* **dd** should die at the bad sector in the disk.
3. **dd** again to directory **/tmp/***foo2* using that command's **skip=***n* to skip past the bad sector (or sectors).
4. Repeat step 3 (if it died too) until the end of the disk is reached. Now you have a set of directories named **/tmp/***foo[1...n]* that contain parts of the bad disk.
5. Use the command

```
dd if=/tmp/foo1 of=/dev/fha0
```

with the new, defect-free disk.

6. Now, use the command

   ```
   dd if=/tmp/foo2 of=/dev/fha0 seek=whatever
   ```

   to place *foo2* into the right place on the new disk.

7. Repeat 6 for each directory *foo3* through *fooN*.

8. Create a 512-byte file that contain the string

   ```
   GARBAGE\n
   ```

   repeated 64 times. Use **dd** to copy it into new disk where the bad sectors were.

Now, you *should* have a disk that is a mirror image of the old, damaged dump disk. Each bad sectors will have been replaced by 64 iterations of the string **GARBAGE\n**. As noted, there is no guarantee that this scheme will work in every instance (the chances of error are quite high), but it will give you a fighting chance to save your data.

Files

/dev/dump — Dump device
/etc/ddate — Dump date file

See Also

commands, dump, dumpdir

Diagnostics

Most of the diagnostics produced by **restor** are self-explanatory, and are caused by internal table overflows or I/O errors on the dump medium or file system.

If the dump spans multiple disks or reels, **restor** asks you to mount the next disk at the appropriate time. Type a newline when the disk has been mounted. **restor** verifies that this is the correct disk, and gives you another chance if the disk number in the dump header is incorrect.

Notes

You cannot perform a mass restore onto a live root partition. Instead, boot a stand-alone version of COHERENT on a floppy-disk drive, or boot from an alternative COHERENT file system on another hard-disk partition before you attempt to do a mass restoration.

The handling of tapes with multiple dumps on them (created by dumping to the no rewind special files) is not very general. Basically, **restor** assumes that tapes holding multiple dumps and tapes holding dumps that span multiple reels are mutually exclusive. You can restore from any file on a reel by positioning the tape and then restoring with the **x** or **r** commands, which do not reposition the tape. It is (almost) impossible to use the **X** or **R** commands, as the position of the dump tape will be lost when **restor** closes it.

dump requires that its output be written to disks that are free of bad sectors. If you write a dump to a disk with bad sectors, you will not be able to restore files from that disk. See **dump** for directions on processing disks to ensure that they are free of bad sectors.

return — C Keyword

Return a value and control to calling function

return is a C statement that returns a value from a function to the function that called it. **return** can be used without a value, to return control of the program to the calling function; also, the calling function is free to ignore the value **return** hands it. Note that it is good programming practice to declare functions that return nothing to be of type **void**.

A function can return only one value to the function that called it. Most often, this value is used to signal whether the function performed successfully or not.

See Also

C keywords
ANSI Standard, §6.6.6.4

rev — Command

Print text backwards
rev [*file ...***]**

rev reverses the order of the characters in each line of each input *file* and writes the result to the standard output. If no *file* is specified, the standard input is used instead.

Example

The following allows you to give a command like Mandrake the Magician. Typing

```
rev
Rocks break down wall!
<ctrl-D>
```

displays:

```
!llaw nwod kaerb skcoR
```

on your screen.

See Also

commands

rewind() — STDIO Function (libc)

Reset file pointer
#include <stdio.h>
void rewind(*fp***)**
FILE **fp*;**

rewind() resets the file pointer to the beginning of stream *fp*. It is a synonym for **fseek(***fp***, 0L, 0**).

Example

For an example of this routine, see the entry for **fscanf()**.

See Also

fseek(), ftell(), libc, lseek()
ANSI Standard, §7.9.9.5
POSIX Standard, §8.1

Notes

Release 4.2 of COHERENT has changed **rewind()** to conform to the ANSI Standard. Prior to release 4.2, **rewind()** returned EOF if an error occurs, and otherwise returned zero. **rewind()** now returns nothing. Programs that depend upon the return value of **rewind()** should be modified to conform to this change.

rewinddir() — General Function (libc)

Rewind a directory stream
#include <dirent.h>
void rewinddir(*dirp***)**
DIR **dirp*;**

The COHERENT function **rewinddir()** is one of a set of COHERENT routines that manipulate directories in a device-independent manner. It resets the current position within the directory stream pointed to by *dirp* to the beginning of the directory.

rewinddir() discards all buffered data for its data stream. This ensures that your program knows about all modifications to the directory that occurred since the last time the directory stream was opened or rewound.

If an error occurs, **rewinddir()** exits and sets **errno** to an appropriate value.

See Also

closedir(), dirent.h, getdents(), libc, opendir(), readdir(), seekdir(), telldir()
POSIX Standard, §5.1.2

Notes

Because directory entries can dynamically appear and disappear, and because directory contents are buffered by these routines, an application may need to continually rescan a directory to maintain an accurate picture of its active entries.

The COHERENT implementation of the **dirent** routines was written by D. Gwynn.

rindex() — String Function (libc)

Find rightmost occurrence of a character in a string
#include <string.h>
char *rindex(*string***,** *c***) char ****string***; char** *c***;**

rindex() scans *string* for the last occurrence of character *c*. If *c* is found, **rindex()** returns a pointer to it. If it is not found, **rindex()** returns NULL.

Example

This example uses **rindex()** to help strip a sample file name of the path information.

```
#include <stdio.h>
#include <string.h>
#include <misc.h>
#define PATHSEP '/'           /* path name separator */

main()
{
      char *testpath = "/foo/bar/baz";
      printf("Before massaging: %s\n", testpath);
      printf("After massaging: %s\n", basename(testpath));
      return(EXIT_SUCCESS);
}

char *basename(path)
char *path;
{
      char *cp;
      return (((cp = rindex(path, PATHSEP)) == NULL)
            ? path : ++cp);
}
```

See Also

libc, strchr(), strrchr(), string.h

Notes

You *must* include header file **string.h** in any program that uses **rindex()**, or that program will not link correctly.

rindex() is now obsolete. You should use **strrchr()** instead.

rm — Command

Remove files
rm [-firtv] *file ...*

rm removes each *file*. If no other links exist, **rm** frees the data blocks associated with the file.

To remove a file, a user must have write and execute permission on the directory in which the file resides, and must also have write permission on the file itself. The force option **-f** forces the file to be removed if the user does not have write permission on the file itself. It suppresses all error messages and prompts.

The interactive option **-i** tells **rm** to prompt for permission to delete each *file*.

The recursive removal option **-r** causes **rm** to descend into every directory, search for and delete files, and descend further into subdirectories. Directories are removed if the directory is empty, is not the current directory, and is not the root directory.

The test option **-t** performs all access testing but removes no files.

The verbose option **-v** tells **rm** to print each file rm and the action taken. In conjunction with the **-t** option, this allows the extent of possible damage to be previewed.

See Also

commands, ln, rmdir

Notes

Absence of delete permission in parent directories is reported with the message *file*: **permission denied**. Write protection is not inherited by subdirectories; they must be protected individually.

Note that unlike the similarly named command under MS-DOS, COHERENT's version of **rm** will *not* prompt you if you request a mass deletion. Thus, the command

```
rm *
```

will silently and immediately delete all files in the current directory. *Caveat utilitor!*

rmail — Command

Receive mail from remote sites
rmail [-LlRr] -q *num* **-u** *uuxflags address ...*

Command **rmail** receives and processes mail from remote sites. It reads and interprets the address on the mail. If the mail is addressed to a user on your local system, it hands the mail to the local-mail deliverer **lmail** for delivery; if the mail is addressed to a remote system, it queues the mail for forwarding to that system.

It is very unusual for a user to invoke **rmail** from the command line. **rmail** usually is invoked by another program; in particular, the command **uuxqt** invokes it to process mail uploaded from another machine via UUCP.

Options

The command **uux** can pass options to **rmail** to control its behavior. **rmail** recognizes the following command-line options:

-L Hand all mail that whose address includes a UUCP path to the local mailer **lmail** for processing, presumably to make use of other transport mechanisms (e.g., Ethernet). This option, and option **-l**, defers all routing until **lmail** has re-forwarded the mail to **smail** for further processing.

-l Hand all mail whose address includes a domain name to the local mailer **lmail** for processing, so they can be processed for non-UUCP domains.

-q *number*
Set the queuing threshold to *number*. When routing mail to a given host, **rmail** checks the "cost" of contacting the host; this cost set in file **/usr/lib/mail/paths**. If the cost is less the queueing threshold, then **rmail** sends the mail immediately; otherwise, it queues the mail for later shipment. Under COHERENT, default queueing threshold is 100.

-R Reroute UUCP paths, trying successively larger righthand substrings of a path until a component is recognized.

-r Route the first component of a UUCP path (**host!address**) in addition to routing domain addresses (**user@domain**).

-u *uuxflags*
Pass all *uuxflags* to the command **uux** for inclusion in the remote-mail command. This overrides any of the default values and other queueing strategies.

Files

/usr/lib/mail/aliases — File from which aliases data base is built
/usr/lib/mail/paths — File from which paths data base is built
/usr/spool/uucp/.Log/mail/mail — Log of mail
/bin/lmail — Local mailer
/bin/mail — Mail user agent

See Also

aliases, commands, lmail, mail [overview], paths, smail

Notes

rmail is a link to command **smail**. For information on how **rmail** parses addresses and constructs headers, see the Lexicon entry for **smail**.

Because **rmail** is a link to **smail**, it actually recognizes all of **smail**'s command-line options; however, it ignores all except those listed above.

rmdir — Command

Remove directories
rmdir [-f] *directory ...*

rmdir removes each *directory*. This will not be allowed if a *directory* is the current working directory or is not empty. The force option **-f** allows the superuser to override these restrictions. **rmdir** removes the '.' and '..' entries automatically. Note that using the **-f** option on a directory that is not empty will damage the file system, and require that it be fixed with **fsck**.

See Also

commands, mkdir, rm

Notes

rmdir -f does *not* remove files from a nonempty directory: it simply orphans them. To remove a nonempty directory and its contents, use **rm -r** instead.

rmdir() — System Call (libc)

Remove a directory
#include <unistd.h>
int rmdir(*path***)**
char **path***path***;**

The COHERENT system call **rmdir()** removes the directory specified by argument *path*. To remove the directory, the following conditions must apply:

- *path* must exist and be accessible, it must be empty (i.e., contain only entries for '.' and '..').
- You must have permission to remove the directory.
- The file system that contains *path* must not be mounted "read only".
- The directory must not be the current directory for any process.
- The directory must not be a mount point for another file system.

If the directory is successfully removed, **rmdir()** returns zero. If an error occurs, it returns -1 and sets **errno** to an appropriate value.

See Also

libc, mkdir, mkdir(), rmdir, unistd.h, unlink()
POSIX Standard, §5.5.2

root — Definition

root is the login name for the superuser.

See Also

superuser, Using COHERENT

route — Command

Show or reset a user's default printer
route [*printer***]**

The command **route** shows or resets your default printer. When invoked without an argument, it displays your default printer, plus a list of available alternative printers. When invoked with the argument *printer*, it changes your default printer to that printer. *printer* must name a printer that has been described to the spooler by the command **lpadmin**.

Note that this feature is an extension to the version of **lp** that is included with UNIX System V.

See Also

commands, lp, lpadmin, printer

Notes

route is unique to the MLP implementation of the **lp** spooler. Scripts that use it will not be portable to other implementations of **lp**.

route is a link to **lpstat**.

routers — System Administration

Rules for resolving mail addresses to remote systems
/usr/lib/mail/routers

File **/usr/lib/mail/routers** defines one or more *routers*. Each router defines a method by which **smail** routes mail to a remote system.

Each entry within **routers** names a router and sets its attributes. The order of entries is important, because **smail** invokes routers in the order in which they appear in this file. Each entry consists of the following information:

- The name of the router. This attribute begins the definition of a router. The name must be unique, it must appear flush with the left margin, and must be followed by a single colon ':'.
- The name of the *driver*, or program that implements the router. This can be a command that is part of **smail**'s suite of utilities (which are contained in directory **/usr/lib/mail**), or can be an ordinary COHERENT command. If the latter, then the full name of the command that implements the driver is given with a **cmd** attribute; this is shown below.
- A set of generic attributes for the router. These attributes are "generic" because they can come from a set that can be applied to any router.
- A set of driver-specific attributes. These can be applied only to routers that use this driver.

To extend an entry across multiple lines, begin successive lines with white space.

For example, the following entry gives the attributes for a director that reads aliases from a file named **/private/usr/lib/aliases**:

```
# read aliases from a file private to one machine on the network
private_aliases:
      driver=aliasfile, owner=owner-$user ;
      file=/private/usr/lib/aliases
```

This entry is named **private_aliases**. It depends upon the low-level director-driver routine named **aliasfile**, which is built into **smail**, and which implements a general mechanism for looking up aliases within a data base. By default, the driver **aliasfile** reads file **/usr/lib/mail/aliases** (which is simply a file that contains ASCII records in no particular order); this routers tells it instead to read file **/private/usr/lib/mail/aliases**. (For details on the format of an aliases file, see the Lexicon entry **aliases**). Finally, this router tells **smail** that if this director discovers an error while it is processing its input, then it (**smail**) should sends a mail message to an address formed by prefixing the string "owner-" onto the name of the alias.

Attributes of a Router

The following gives the generic attributes can be used in router entry. Each attribute is followed by its type (Boolean or string). To set a string attribute, its name should be followed by an '=', then the value to which you are setting it. To set a Boolean attribute, prefix it with a '+'; to unset a Boolean attribute, prefix it with a '-'.

always (Boolean)

A router will not always find a complete match for a particular host name. For example, if a routing data base has a route to the domain **amdahl.com** but not to the host name **futatsu.uts.amdahl.com**, then the routing driver might return the route to **amdahl.com**.

In general, **smail** uses the route that matches the largest "chunk" of the target host. However, if you set the attribute **always**, then **smail** uses any match found by this router in preference to any route returned by any router that appears below it within **routers.**

This attribute is useful for hard-wiring a certain number of routes within a small data base. For example, this is useful for an Internet site that is the gateway for a small number of UUCP sites within the UUCP zone.

driver (string)
This attribute gives the set of low-level functions that do the work of routing remote mail. This attribute is required.

method (string)
transport (string)
A router driver can internally set the transport it uses to deliver mail to a remote site. If it does not do so, then you must set either a **method** or a **transport** attribute, to specify how the mail is to be delivered. The attribute **method** names the file whose contents relate host names to transports. The attribute **transport** specifies a particular transport that is defined in file **/usr/lib/mail/transports**. If the file named in a **method** attribute does not contain a match for all hosts, then **smail** uses the transport named with the **transport** attribute. The format of a method file is given in the next section.

Method Files

Method files relates a set of host names with the set of transports to be used to deliver mail to those hosts. Each entry should have the form:

hostname transport-name

which states that **smail** should use *transport-name* to deliver mail to *hostname*. As a special case, if *hostname* is the special string '*', the entry matches any host. You should use this catch-all feature only in the last entry in a method file.

You can associate an entry in a method file with a particular grade of message. This lets you assign each grade of mail its own transport; for example, you may wish to use non-demand UUCP for messages with a "bulk" or "junk" precedence. To specify a range of grades, append the range of grade-letters to the host name, separated by '/'. Entries with grades can be in any of the forms:

hostname/X transport-name
hostname/X- transport-name*
hostname/-Y transport-name*
hostname/X-Y transport-name

For a discussion of grade letters and their correlation with message-precedence strings, see the description of attribute **grades** in the Lexicon entry for **config (smail).** In the first form, the transport is used for an exact match of the grade letter. In the second form, a match requires a grade a character value of at least *X*. In the third, form a match requires a grade character value of at most *Y*. The final form specifies a range of grades from character value *X* to character value *Y*.

The Default Configuration

The following gives the routers defined in the default version of file **/usr/lib/mail/routers** that is included with COHERENT.

The first router is named **paths**. It processes the contents of file **/usr/lib/mail/paths**:

```
# paths - route using a paths file, like that produced by the pathalias program
paths:      driver=pathalias,    # general-use paths router
      transport=uux;       # for matches, deliver over UUCP

      file=paths,          # sorted file containing path info
      proto=dbm,           # use a DBM-style data base
      optional,            # ignore if the file does not exist
      -required,           # no required domains
      domain=uucp,         # strip ending ".uucp" before searching
```

The command **pathalias**, which this router uses to read file **paths**, is described in its own Lexicon entry; as is command **uux**, which this router invokes to transport the files to the remote site.

The next router, named **uucp_neighbors**, matches nearby systems that are accessible via UUCP:

LEXICON

```
# uucp_neighbors - match neighbors accessible over UUCP
uucp_neighbors:
      driver=uuname,          # use a program which returns neighbors
      transport=uux;

      cmd=/usr/bin/uuname,    # specifically, use the uuname program
      domain=uucp,            # strip ending ".uucp" before searching
```

Command **uuname** is part of the Taylor UUCP package that is included with COHERENT. It is described in its own Lexicon entry. Under COHERENT, this command always returns the name of your local host.

The final router describes how to route mail to the "smart host." This is a system that knows how to access more remote systems than your system does, and that you trust to handle mail correctly. **smail** forwards to the smart host all mail that it does not know how to route, in the hope that the smart host will know what to do with it.

```
# smart_host - a partically specified smarthost director
#
# If the config file attribute smart_path is defined as a path from the
# local host to a remote host, then host names not matched otherwise will
# be sent off to the stated remote host.  The config file attribute
# smart_transport can be used to specify a different transport.
#
# If the smart_path attribute is not defined, this router is ignored.
smart_host:
      driver=smarthost, # special-case driver
      transport=uux;          # by default deliver over UUCP

      -path,                  # use smart_path config file variable?
```

See Also

Administering COHERENT, config [smail], directors, mail [overview], smail, transports

Notes

For information on how the configuration files **directors**, **routers**, and **transports** relate to each other, see the Lexicon entry for **directors**.

rpow() — Multiple-Precision Mathematics (libmp)

Raise multiple-precision integer to power
#include <mprec.h>
void rpow(*a, b, c***)**
mint **a, *b, *c**;**

rpow() sets the multiple-precision integer (or **mint**) pointed to by *c* to the value pointed to by *a* raised to power of the value pointed to by *b*.

See Also

libmp

RS-232 — Technical Information

Serial port wiring

This article details the connections (pinouts) of EIA standard RS-232C. This connector consists of a D-shaped plug with 25 pins in two rows: 13 pins in the upper row and 12 in the lower. This interface is commonly used by devices that require a serial interface to a computer; these devices include modems, terminals, serial printers, and such specialized devices as bar-code scanners. In addition, this articles gives the pinouts of the nine-pin DB-9P connector, which is a nine-pin version of the RS-232 that is commonly used in AT and AT-compatible computers.

RS-232 Pinout

The following table gives the 25-pin EIA standard RS-232C pinouts. It also gives:

- Nine-pin DB-9P convention
- Common abbreviations of signal names
- Abbreviations of RS-232 signal names
- Equivalent CCITT signal-number designations
- Signal direction (as appropriate)
- Signal description

Please note that in this table, **DTE** stands for "data terminal equipment" and refers to terminal-type equipment such as a PC or a terminal, whereas **DCE** stands for "data communications equipment" and refers to modems and modem-type equipment.

DB-25 Pin #	*DB-9 Pin #*	*Common Name*	*EIA*	*CCITT*	*DTE-DCE*	*Description*
1		FG	AA	101	—	Frame ground
2	3	TD	BA	103	→	Transmitted data
3	2	RD	BB	104	←	Received data
4	7	RTS	CA	105	→	Request to send
5	8	CTS	CB	106	←	Clear to send
6	6	DSR	CC	107	←	Data set ready
7	5	SG	AB	102	—	Signal ground
8	1	DCD	CF	109	←	Data carrier detect
9		—	—	—	—	Positive DC test voltage
10		—	—	—	—	Negative DC test voltage
11		QM	—	—	←	Equalizer mode
12		SDCD	SCF	122	←	Secondary carrier detect
13		SCTS	SCB	121	←	Secondary clear to send
14		STD	SBA	118	→	Secondary transmitted data
15		TC	DB	114	←	Transmitter clock
16		SRD	SBB	119	←	Secondary receiver clock
17		RC	DD	115	→	Receiver clock
18		DCR	—	—	←	Divided clock receiver
19		SRTS	SCA	120	→	Secondary request to send
20	4	DTR	CD	108.2	→	Data terminal ready
21		SQ	CG	110	←	Signal quality
22	9	RI	CE	125	←	Ring indicator
23		—	CH	111	→	Data rate selector
24		TC	DA	113	←	Transmitted clock
25						

Files

/usr/pub/rs232 — On-line version of above table

See Also

Administering COHERENT, asy, modem, terminal
Seyer, M.D.: *RS-232 Made Easy: Connecting Computers, Printers, Terminals, and Modems*. Englewood Cliffs, NJ, Prentice-Hall Inc., 1984.

Notes

Serial ports on the back of the PC use either a 25-pin male (DB-25P) or a nine-pin male (DB-9P) connector. Due to what can only be regarded as extreme stupidity, the 25-pin female (DB-25S) connector was chosen for the parallel (printer) port, rather than using the usual 36-pin parallel connector. Do not confuse these ports when wiring custom-cable assemblies, as you can damage your equipment!

rsmtp — Command

Run batched SMTP mail
/bin/rsmtp

Command **rsmpt** reads and executes Simple Mail Transfer Protocol (SMTP) commands from the standard input. It normally is used to execute a batched form of SMTP between machines via a remote execution service, e.g., UUCP. **rsmtp** reports failures through return mail.

See Also

commands, mail [overview], smail

Notes

rsmtp is a link to **smail**.

rubik — Command

Play Rubik's cube
/usr/games/rubik

The command **rubik** lets you fiddle with an electronic version of Rubik's cube. By issuing commands, you can "rotate" the segments of the virtual cube and, with some agony, align all the "colors".

rubik is written in **m4**, and is a good example of extended programming in this utility.

See Also

commands, m4

runq — Command

Periodically process the mail queue
/bin/runq

Command **runq** checks the spool directory that holds incoming mail, and processes it. It is equivalent to the command **smail -q**.

See Also

commands, mail [overview], smail

Notes

runq is a link to **smail**.

rvalue — Definition

An **rvalue** is the value of an expression. The name comes from the assignment expression **e1=e2;**, in which the right operand is an rvalue.

Unlike an lvalue, an rvalue can be either a variable or a constant.

See Also

lvalue, Programming COHERENT
ANSI Standard, §6.2.2.1

sa — Command

Print a summary of process accounting
sa [-abcjlmnrstu] [-v *N***] [***file***]**

One of the accounting mechanisms available on the COHERENT system is *process accounting* (also called *shell accounting*), which records the commands executed by each user. The command **accton** enables or disables shell accounting.

The command **sa** scans the accounting information in *file* and prints a summary. If *file* is omitted, it reads the file **/usr/adm/acct** by default. For each command executed, **sa** prints the number of calls made, the total CPU time (user and system), and the total real time. The output is ordered by decreasing CPU time.

sa recognizes the following options:

a Place commands executed only once and command names with unprintable characters in the category "***other".

b Sort by average CPU time per call.

c Also print CPU time as a percentage of all CPU time used.

j Print average times per call rather than totals.

l Separate user and system time.

m Accumulate information for each user rather for each command.

n Sort by number of calls.

r Reverse the order of the sort.

s After scanning, condense the accounting file and merge it into the summary files.

t Also print the CPU time as a percentage of real time.

u Print the user and command for each accounting record; this option overrides all others.

v *N* For commands called no more than *N* times, where *N* is a digit, **sa** asks whether to place the command in the category "**junk**".

sa uses the summary files **/usr/adm/savacct** and **/usr/adm/usracct** to lessen disk usage.

Files

/usr/adm/acct — Default account data
/usr/adm/savacct — Summary
/usr/adm/usracct — Summary

See Also

ac, acct(), acct.h, accton, commands

Notes

The file **/usr/adm/acct** can become very large; therefore, you should truncate it periodically. Special care should be taken if process accounting is enabled on a COHERENT system with limited disk space.

savelog — Command

Save a mail log

/usr/lib/mail/savelog [-c *cycle***] [-g** *group***] [-l]**
[-m *mode***] [-u** *user***] [-t]** *file ...*

The script **savelog** archives each *file*. This script normally is used to save copies of **smail**'s log files, and of other files that grow relentlessly. It copies each *file* into a special archiving directory, and gives the copy a name that reflects how recently it was created. Unless you request otherwise, it also compresses each *file*.

When it saves *file*, **savelog** copies it into directory **$PWD/OLD**, where **$PWD** represents the directory within which the file normally resides. If sub-directory **OLD** does not exist, **savelog** creates it, and gives it mode 0755.

As you probably will invoke **savelog** periodically to save a log file, this directory can hold an indefinite number of archives of *file*, each created at a different time in the past. To help you distinguish among these archives, **savelog** names them as follows:

*file***.***number***[.***compression_suffix***]**

number represents the order in which the archives were created, zero being the newest; and *compression_suffix* indicates the suffix that the compression program gives the file — **.Z** if the archive is compressed with **compress** (which **savelog** uses by default), or **.gz** if compressed with **gzip**. Note that archive '0' is never compressed, on the off chance that a process still has its corresponding file opened for input.

If *file* does not exist or has zero length, **savelog** performs no further processing. To override this behavior, use option **-t**.

When *file* exists and has a length greater than zero, **savelog** performs the following actions:

- First, it increases by one the version number of each existing copy of *file*. For example, if you are saving file **foo** for the seventh time, then **savelog** moves file **foo.6** to **foo.7**; then moves **foo.5** to **foo.6**; and so on. **savelog** does this regardless of whether an archive is compressed, or whether you used option **-t** on the command line. By default, **savelog** keeps only seven versions of a given *file*, and throws away those versions that exceed that limit. To increase or decrease this limit, use command-line option **-c**, described below.
- If you did *not* use command-line option **-t**, **savelog** next compresses the new *file***.1**. It also changes this file, subject to the command-line options **-m**, **-u**, and **-g** (described below).
- It moves *file* to **OLD/***file.0*.
- If you use any of the command-line options **-m**, **-u**, **-g**, or **-t**, **savelog** re-creates *file*, subject to the given flags.
- Finally, **savelog** modifies the newly created file **OLD/***file***.0**, subject to the settings of command-line options **-m**, **-u**, and **-g**.

Command-line Options

savelog recognizes the following command-line options:

-c *cycle* Save no more than *cycle* versions of *file*. The default is seven, numbered '0' through '6'. *cycle* must be no less than two. Note that because numbering begins with zero, version number *cycle* of *file* is never created.

-g *group*
Use the command **chgrp** to give *group* the group ownership of *file* and its archives.

-l Do not compress any log files.

-m *mode*
Invoke the command **chmod** to set permissions on the log files to *mode*.

-t Touch *file*— that is, create a new, empty copy of *file* after archiving it. This lets you ensure that the log file is re-created with correct permissions.

-u *user* Invoke the command **chown** to make *user* the owner of the archives of *file*.

See Also

commands, mail [overview], uulog

Notes

If you do not use any of the command-line options **-m**, **-u**, or **-g**, **savelog** does not re-create *file* after archiving it.

sbrk() — General Function (libc)

Increase a program's data space
#include <unistd.h>
char *__**sbrk(***__increment__**)** **unsigned int** *increment*__**;**__

sbrk() increases a program's data space by *increment* bytes. **malloc()** calls **sbrk()** should you attempt to allocate more space than is available in the program's data space.

If all goes well, **sbrk()** returns the old break value. Otherwise, if an error occurs, **sbrk()** returns -1 and sets **errno** to an appropriate value.

See Also

brk(), libc, malloc()

Notes

sbrk() will not increase the size of the program data area if the physical memory requested exceeds the physical memory allocated by COHERENT. **sbrk()** does not keep track of how space is used; therefore, memory seized with **sbrk()** cannot be freed. *Caveat utilitor.*

scanf() — STDIO Function (libc)

Accept and format input
#include <stdio.h>
int scanf(*format, arg1, … argN***)**
char **format***__;__* **[data type]** **arg1, … *argN__;__*

scanf() reads the standard input, and uses the string *format* to specify a format for each *arg1* through *argN*, each of which must be a pointer.

scanf() reads one character at a time from *format*; white space characters are ignored. The percent sign character '%' marks the beginning of a conversion specification. '%' may be followed by characters that indicate the width of the input field and the type of conversion to be done.

scanf() reads the standard input until the return key is pressed. Inappropriate characters are thrown away; e.g., it will not try to write an alphabetic character into an **int**.

scanf() returns the number of arguments filled. It returns EOF if no arguments can be filled or if an error occurs.

Modifiers

The following modifiers can be used within the conversion string:

1. An asterisk '*', which tells **scanf** to skip the next conversion; that is, read the next token but do not write it into the corresponding argument.
2. A decimal integer, which tells **scanf** the maximum width of the next field being read. How the field width is used varies among conversion specifier. See the table of specifiers below for more information.
3. One of the three modifiers **h**, **l**, or **L**, whose use is described below.

Modifiers

The following three modifiers may be used before a conversion specifier:

h When used before the conversion specifiers **d**, **i**, **o**, **u**, **x**, or **X**, it specifies that the corresponding argument points to a **short int** or an **unsigned short int**. When used before **n**, it indicates that the corresponding argument points to a **short int**. In implementations where **short int** and **int** are synonymous, it is not needed. However, it is useful in writing portable code.

l When used before the conversion specifiers **e**, **E**, **f**, **F**, or **G**, it indicates that the corresponding argument points to a **double** rather than a **float**.

L When used before the conversion specifiers **e**, **E**, **f**, **F**, or **G**, it indicates that the corresponding argument points to a **long double** rather than a **float**.

Conversion Specifiers

scanf() recognizes the following conversion specifiers:

c Assign the next input character to the next *arg*, which should be of type **char** *. The field width specifies the number of characters (default, one). **scanf()** does not write a null character at the end of the array it creates. This specifier forces **scanf()** to read and store white-space characters and numerals, as well as letters.

d Convert the token to a decimal integer. The format should be equivalent to that expected by the function **strtol()** with a base argument of ten. The corresponding argument should point to an **int**.

D Assign the decimal integer from the next input field to the next *arg*, which should be of type **long** *.

e Convert the token to a floating-point number. The format of the token should be that expected by the function **strtod()** for a floating-point number that uses exponential notation. The corresponding argument should point to a **float** if no modifiers are present, to a **double** if the **l** modifier is present, or to a **long double** if the **L** modifier is present.

E Same as **e**. Prior to release 4.2 of COHERENT, this conversion specifier converted the token to a **double**. This change has been made to conform to the ANSI Standard, and may require that some code be rewritten.

f Convert the token to a floating-point number. The format of the token should be that expected by the function **strtod()** for a floating-point number that uses decimal notation. The corresponding argument should point to a **double**.

g Convert the token to a floating-point number. The format of the token should of that expected by the function **strtod()** for a floating-point number that uses either exponential notation or decimal notation. The corresponding argument should point to a **float** if no modifiers are present, to a **double** if the **l** modifier is present, or to a **long double** if the **L** modifier is present.

G Same as **g**.

i Convert the token to a decimal integer. The format should be equivalent to that expected by the function **strtol()** with a base argument of zero. The corresponding argument should point to an **int**.

n Do not read any text. Write into the corresponding argument the number of characters that **scanf()** has read up to this point. The corresponding argument should point to an **int**.

o Assign the octal integer from the next input field to the next *arg*, which should be of type **int** *.

O Assign the octal integer from the next input field to the next *arg*, which should be of type **long** *.

p The ANSI standard states that the behavior of the **%p** conversion specificer is implementation-specific. Under COHERENT, **%p** converts a strings of digits in hexadecimal notation into an address. For example, in the code

```
char buf[] = "0x7FFFFDBC";
char *foo;
...
sscanf(buf, "%p", &foo);
```

the **%p** specifier reads the contents of **buf** and turns them into an address, which it then uses to initialize the pointer **foo**. You can use the **%p** specifier to turn back into an address the output of **printf()**'s **%p** specifier. Please note that abuse of this specifier can create all manner of fascinating bugs within your programs: *Caveat utilitor*.

r The next argument points to an array of new arguments that may be used recursively. The first argument of the list is a **char** * that contains a new format string. When the list is exhausted, the routine continues from where it left off in the original format string.

s Assign the string from the next input field to the next *arg*, which should be of type **char** *. The array to which the **char** * points should be long enough to accept the string and a terminating null character.

u Convert the token to an unsigned integer. The format should be equivalent to that expected by the function **strtoul()** with a base argument of ten. See **strtoul** for more information. The corresponding argument should point to an **unsigned int**.

x Convert the token from hexadecimal notation to a signed integer. The format should be equivalent to that expected by the function **strtol** with a base argument of 16. See the Lexicon entry for **strtol()** for more information. The corresponding argument should point to an **unsigned int**.

X Same as **x**. Prior to release 4.2 of COHERENT, **X** meant the same as the current **lx**; that is, the corresponding argument points to a **long** instead of an **int**. This has been changed to conform to the ANSI Standard, and may require that some code be rewritten.

It is important to remember that **scanf()** reads up, but not through, the newline character: the newline remains in the standard input device's buffer until you dispose of it somehow. Programmers have been known to forget to empty the buffer before calling **scanf()** a second time, which leads to unexpected results.

Example

The following example uses **scanf()** in a brief dialogue with the user.

```
#include <stdio.h>

main()
{
      int left, right;

      printf("No. of fingers on your left hand:  ");
      /* force message to appear on screen */
      fflush(stdout);
      scanf("%d", &left);

      /* eat newline char */
      while(getchar() != '\n')
            ;

      printf("No. of fingers on your right hand:  ");
      fflush(stdout);
      scanf("%d", &right);

      /* again, eat newline */
      while(getchar() != '\n')
            ;

      printf("You've %d left fingers, %d right, & %d total\n",
            left, right, left+right);
}
```

See Also

fscanf(), libc, sscanf()
ANSI Standard, §7.9.6.4
POSIX Standard, §8.1

Notes

Because C does not perform type checking, it is essential that an argument match its specification. For that reason, **scanf()** is best used to process only data that you are certain are in the correct data format. Rather than use **scanf()** to obtain a string from the keyboard: we recommend that you use **gets()** to obtain the string, and use **strtok()** or **sscanf()** to parse it.

scat — Command

Print text files one screenful at a time
scat [[*option ...***] [***file* **...]] ...**

scat prints each *file* on the standard output, one screenful (24 lines) at a time if the output is a screen. **scat** reads and prints the standard input if no *file* is named.

The text is processed to allow convenient viewing on a screen; the options described below select the nature of the processing. Options begin with '-' and may be interspersed with file names.

scat scans two argument lists. The first is in the environmental **SCAT**. It should consist of arguments separated by white space (space, tab, or newline characters), with no quoting or shell metacharacters. This string is a useful place to set terminal-dependent parameters (such as page width and length) and to place invocation lists (see below). The second argument list is supplied on the command line.

scat recognizes the following options:

-l Do not stop at EOF if exactly one file was specified on the command line.

-b*n* Begin output at input line *n.*

-c Represent all control characters unambiguously. With this option, **scat** prints control characters in the range 0-037 as a character in the range 0100-0137 prefixed by a carat '^'; for example, SOH appears as "^A" and DEL as "^?" It prints mark-parity characters (in the range of 0200-0377) with '~'; for example, mark-parity 'A' and SOH appear as "~A" and "~^A", respectively. It also prefixes the characters '^', '~', and '\' with a '\'. This option overrides the option **-t**.

-cs Like **-c,** but map space ' ' to underscore '_' and prefix underscore '_' with '\'.

-ct Like **-c,** but map tabs to spaces, not "^I".

-i*n* Shift the display window right *n* columns into the text field. This is useful for viewing long lines.

-l*n* Set the display window length to *n* lines. The default is 24 normally, 34 for the Tek 4012.

-n Number input lines; wrapped lines are not numbered.

-r Remote; the output is not paged.

-s Skip empty lines.

-S*n* Seek *n* bytes into input before processing.

-t Truncate long lines. Normally, **scat** wraps each long line, with the interrupted portion delimited by a '\'.

-w*n* Set the display window width to *n* columns. The default is 80 normally, 72 for the Tek 4012.

-x Expand tabs.

-. *suffix* Invoke options by file-name suffix. If a file name ends with *.suffix*, then **scat** scans the argument sublist starting immediately after the invocation flag. New options will apply to the invoking file only. A sublist is terminated by the end of the argument list, by a file name, by the "--" flag, or by another "-." (invocation lists do not nest).

-- Terminate a sublist (see previous option).

Numbers may begin with **0** to indicate octal, and may end in **b** or **k** to be scaled by 512 or 1,024, respectively.

If the output is being paged, **scat** waits for a user response, which may be one of the following:

newline	Display next page
/	Display next half-page
space	Display next line
f	Print current file name and line number
n	**scat** next file
q	Quit

Example

The following shows how to use the environment argument list, invocation lists, and sublists:

```
SCAT="-l24 -.c -n -.s -b5"
export SCAT
scat *.c *.s
```

After processing the **SCAT** argument list, **scat** processes the command line argument list "***.c** ***.s**" with the page length at 24 lines. If a file is a C source ("***.c**") the invoke option in the **SCAT** argument list numbers the output lines. If a file is an assembly source ("***.s**") **scat** skips the first four lines.

See Also

cat, commands, pr

sched.h — Header File

Define constants used with scheduling
#include <sys/sched.h>

sched.h defines constants and structures that are used by routines that perform scheduling.

See Also

header files

script — Command

Capture a terminal session into a file
script [-l *logfile***] [***command***]**

The COHERENT command **script** executes *command* while copying all terminal output to *logfile*. *logfile* defaults to file **Log.***pid* in your current directory, where *pid* is the number of the recording process. *command* must specify a full path name. If the terminal echoes keyboard input, **script** records these keystrokes in *logfile*.

If no *command* is specified, **script** executes the command specified by environmental variable **SHELL** by default. If **SHELL** is not defined, **script** assumes **/bin/sh**.

To exit from **script**, just type **exit** from a command prompt.

See Also

commands

Notes

script is intended to capture what you type for purposes of debugging. What it captures cannot be replayed into the shell.

sdevice — System Administration

Configure drivers included within kernel
/etc/conf/sdevice

File **sdevice** configures the drivers that can be included within the COHERENT kernel. Command **idmkcoh** reads this file when it builds a new COHERENT kernel, and uses the information within it to configure the suite of drivers it links into the kernel.

There is one line within the file for each type of hardware device; if a driver manipulates more than one type of device, then it has one entry for each type of device it manipulates. A driver's entry within file **/etc/conf/mdevice** indicates how many entries a driver can have with **sdevice**: if field 3 contains flag 'o', the device can have only one entry; whereas if field 3 does not contain this flag, it can have more than one entry (although it is not required to do so). An entry that begins with a pound sign '#' is a comment, and is ignored by **idmkcoh**.

Each entry within **sdevice** consists of ten fields, as follows:

1. *Name*
This gives the name of driver, and must match the name given in **mdevice**. It cannot exceed eight characters.

2. *Included in Kernel?*
This field indicates whether the driver is to be linked into the kernel: 'Y' indicates that it is, 'N' that it is not.

3. *Number of Units*
The number of the hardware units that this driver can manipulate. Under COHERENT, this is always set to zero.

4. *Interrupt Priority*
The device's interrupt priority. This must be a value between 0 and 8: zero indicates that this device is not interrupt driven, whereas a value from 1 to 8 gives the interrupt priority.

5. *Interrupt Type*
The type of interrupt for this device. The legal values are as follows:

0 This device is not interrupt driven.

1 The device is interrupt driven. If the driver controls more than one device, each requires a separate interrupt.

2 The device is interrupt driven. If the driver supports more than one device, all share the same interrupt.

3 The device requires an interrupt line. If the driver supports more than one device, all share the same interrupt. Multiple device drivers that the same interrupt priority can share this interrupt; however, this requires special hardware support.

6. *Interrupt Vector*
The interrupt vector used by the device. If field 5 is set to zero, this must be also.

7. *Low I/O Address*
The low I/O address through which the driver communicates with the device. Set this field to zero if it is not used.

8. *High I/O Address*
The high I/O address through which the driver communicates with the device. Set this field to zero if it is not used.

9. *Low Memory Address*
The low address of memory within the controller of the device being manipulated. Set this field to zero if it is not used.

10. *High Memory Address*
The high address of memory within the controller of the device being manipulated. Set this field to zero if it is not used.

Note that all COHERENT drivers current set fields 7 through 10 to zero.

For examples of settings for this, read the file itself. For an example of modifying this file to add a new driver, see the Lexicon entry for **device drivers**.

See Also

Administering COHERENT, device drivers, mdevice, mtune, stune

sdiv() — Multiple-Precision Mathematics (libmp)

Divide multiple-precision integers
#include <mprec.h>
void sdiv(*a, n, q, ip***)**
mint **a*, **q*; **int** *n*, **ip*;*

sdiv() divides the multiple-precision integer (or **mint**) pointed to by *a* with the integer *n*, which is in the range 1 <= *n* <= 128. It writes the quotient into the **mint** pointed to by *q* and the remainder into the integer pointed to by *ip*.

See Also

libmp

SECONDS — Environmental Variable

Number of seconds since current shell started

The Korn shell stores in environmental variable **SECONDS** the number of seconds since the current shell was started.

See Also

environmental variables, ksh

security — System Administration

Because COHERENT is a multi-user, multi-tasking operating system which can support users from remote terminals, steps must be taken to ensure that the system is secure. Sensitive information that is stored on the system must be protected from being read or copied by unauthorized persons; files must be protected against vandalization by intruders. Unless a reasonable degree can be guaranteed, no multi-user operating system can be trusted to archive important information.

In one sense, it is easy to achieve perfect security in a computer system. As Grampp and Morris have noted, "It is easy to run a secure computer system. You merely disconnect all dial-up connections, put the machine and its terminals in a shielded room, and post a guard at the door." For practical uses, however, security means balancing ease of access against restrictiveness: users should have easy access to what is properly theirs, and should be barred from system facilities that do not belong to them.

The COHERENT system has the following tools to assist with security.

Passwords Every user account can be "locked" with a password. Each user can assign her own password, and the system administrator can set passwords for the superusers **root** and **bin**.

Passwords should be changed frequently. A password should have at least six characters, should *not* be a common name or word, and preferably should include a mixture of upper- and lower-case letters, to prevent decryption by brute-force methods.

Passwords should be guarded jealously. In particular, the password for the superuser **root** should be kept secret, as she can read every file and execute every program throughout the system.

Permissions Execution of system-level programs, such as **mount**, is restricted to the superuser **root**. This prevents intruders from seizing superuser permissions through unauthorized manipulation of system services. Ordinary users are also restricted from directly access system devices, for the same reason.

One potential hole in security is the setting the **setuid** bit on programs that are owned by the superuser **root**. Setting this bit grant superuser privileges to whoever runs the program. Two commands often have this bit set: **/etc/enable** and **/etc/disable**. This is done to permit users, in particular user **uucp**, to enable and disable a port. This, however, permits any user to enable or disable a device — including the console device; which means that a cracker who breaks into your system could lock you out of it if she wished.

The lesson is that you should not set the **setuid** bit on any program that is owned by **root** unless you have an excellent reason to do so.

Encryption The command **crypt** performs rotary encryption, similar to that used by the German Enigma machine. Files of sensitive information should be encrypted, to protect them against being read by unauthorized persons. Note that encryption is the only true defense against unauthorized reading: not even the superuser can read an encrypted file unless she has the encryption key.

Many COHERENT systems have only one user and are not networked; for such installations, the normal level of security may be an annoyance. Passwords can be turned off by using the command **passwd** to set the password to **<return>**. The command **chmod** can be used to widen access to devices and system-level utilities; see the Lexicon entry for **chmod** for more information on file access.

Security ultimately is a system-wide responsibility. To quote Grampp and Morris, "By far, the greatest security hazard for a system ... is the set of people who use it. If the people who use a machine are naive about security issues, the machine will be vulnerable regardless of what is done by the local management. This applies particularly to the system's administrators, but ordinary users should also take heed."

See Also

Administering COHERENT, chmod, crypt, login, passwd

Grampp, F.T., Morris, R.H.: UNIX operating system security. *AT&T Bell Lab Tech J* 1984;8:1649-1672.

sed — Command

Stream editor

sed [-n] [-e *command***] [-f** *script***]** *... file ...*

sed is a non-interactive text editor. It reads input from each *file*, or from the standard input if no file is named. It edits the input according to commands given in the *commands* argument and the *script* files. It then writes the edited text onto the standard output.

sed resembles the interactive editor **ed**, but its operation is fundamentally different. **sed** normally edits one line at a time, so it may be used to edit very large files. After it constructs a list of commands from its *commands* and *script* arguments, **sed** reads the input one line at a time into a *work area*. Then **sed** executes each command that applies to the line, as explained below. Finally, it copies the work area to the standard output (unless the **-n** option is specified), erases the work area, and reads the next input line.

Line Identifiers

sed identifies input lines by integer line numbers, beginning with one for the first line of the first *file* and continuing through each successive *file.* The following special forms identify lines:

n A decimal number *n* addresses the *n*th line of the input.

. A period '.' addresses the current input line.

**$** A dollar sign '$' addresses the last line of input.

/*pattern***/** A *pattern* enclosed within slashes addresses the next input line that contains *pattern*. Patterns, also called *regular expressions*, are described in detail below.

Commands

Each command must be on a separate line. Most commands may be optionally preceded by a line identifier (abbreviated as *[n]* in the command summary below) or by two-line identifiers separated by a comma (abbreviated as *[n[,m]]*). If no line identifier precedes a command, **sed** applies the command to every input line. If one line identifier precedes a command, **sed** applies the command to each input line selected by the identifier. If two-line identifiers precede a command, **sed** begins to apply the command when an input line is selected by the first, and continues applying it through an input line selected by the second.

sed recognizes the following commands:

*[n]***=** Output the current input line number.

*[n[,m]]***!***command*
Apply *command* to each line *not* identified by *[n[,m]]***.**

*[n[,m]]***{***command...***}**
Execute each enclosed *command* on the given lines.

:*label* Define *label* for use in branch or test command.

*[n]***a** Append new text after given line. New text is terminated by any line not ending in '\'.

b *[label]* Branch to *label,* which must be defined in a ':' command. If *label* is omitted, branch to end of command script.

*[n[,m]]***c** Change specified lines to new text and proceed with next input line. New text is terminated by any line not ending in '\'.

*[n[,m]]***d** Delete specified lines and proceed with next input line.

*[n[,m]]***D** Delete first line in work area and proceed with next input line.

*[n[,m]]***g** Copy secondary work area to work area, destroying previous contents.

*[n[,m]]***G** Append secondary work area to work area.

*[n[,m]]***h** Copy work area to secondary work area, destroying previous contents.

*[n[,m]]***H** Append work area to secondary work area.

*[n]***i** Insert new text before given line. New text is terminated by any line not ending in '\'.

*[n[,m]]***l** Print selected lines, interpreting non-graphic characters.

*[n[,m]]***n** Print the work area and replace it with the next input line.

*[n[,m]]***N** Append next input line preceded by a newline to work area.

*[n[,m]]***p** Print work area.

*[n[,m]]***P** Print first line of work area.

*[n]***q** Quit without reading any more input.

*[n]***r** *file* Copy *file* to output.

*[n[,m]]***s***[k]/pattern1/pattern2/***[g][p][w** *file]*
Search for *pattern1* and substitute *pattern2* for *k*th occurrence (default, first). If optional **g** is given, substitute all occurrences. If optional **p** is given, print the resulting line. If optional **w** is given, append the resulting line to *file*. Patterns are described in detail below.

*[n[,m]]***t***[label]*
Test if substitutions have been made. If so, branch to *label*, which must be defined in a ':' command. If *label* is omitted, branch to end of command script.

*[n[,m]]***w** *file* Append lines to *file*.

[n[,m]] **x** Exchange the work area and the secondary work area.

*[n[,m]]***y***/chars/replacements/*
Translate characters in *chars* to the corresponding characters in *replacements*.

Patterns

Substitution commands and search specifications may include *patterns*, also called *regular expressions*. Pattern specifications are identical to those of **ed**, except that the special characters '**\n**' match a newline character in the input.

A non-special character in a pattern matches itself. Special characters include the following:

^ Match beginning of line, unless it appears immediately after '[' (see below).

$ Match end of line.

\n Match the newline character.

. Match any character except newline.

* Match zero or more repetitions of preceding character.

[*chars***]** Match any one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

[^*chars***]** Match any character *except* one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

*****c* Disregard special meaning of character *c*.

\(*pattern***\)** Delimit substring *pattern*; for use with *****d*, described below.

In addition, the replacement part *pattern2* of the substitute command may also use the following:

& Insert characters matched by *pattern1*.

*****d* Insert substring delimited by *d*th occurrence of delimiters '\(' and '\)', where *d* is a digit.

Options

sed recognizes the following options:

-e Next argument gives a **sed** command. **sed**'s command line can have more than one **-e** option.

-f Next argument gives file name of command script.

-n Output lines only when explicit **p** or **P** commands are given.

Limits

The COHERENT implementation of **sed** sets the following limits on input and output:

Characters per input record	512
Characters per output record	512
Characters per field	512

See Also

commands, ed, elvis, ex, me, vi
Introduction to the sed Stream Editor

LEXICON

seed48() — Random-Number Function (libc)

Initialize values from which 48-bit random numbers are computed
unsigned short *seed48(*param***)**
unsigned short *param***[3];**

Computation of 48-bit pseudo-random numbers uses two 48-bit integers and one 16-bit integer. One of the 48-bit values holds the "seed" value from which the 48-bit pseudo-random value is computed. This seed can be set explicitly, or is the previously computed pseudo-random number. The other 48-bit integer holds the multiplier from which the pseudo-random number is computed; and the 16-bit integer gives holds the addend.

Function **seed48()** initializes the "seed" from which a 48-bit pseudo-random number is computed. *param* is an array of three unsigned short integers that together comprise the new 48-bit seed value.

seed48() returns a pointer to an array of three unsigned short integers that holds the old seed.

See Also

libc, srand48()

seekdir() — General Function (libc)

Reset the position within a directory stream
void seekdir (*dirp***,** *loc***)**
DIR **dirp***;*
off_t *loc***;**

The function **seekdir()** is one of a set of COHERENT routines that manipulate directories in a device-independent manner. It resets the current position within the directory stream pointed to by *dirp* to *loc*. *loc* must be a position indicator returned by a previous call to **telldir()**.

If an error occurs, **seekdir()** exits and sets **errno** to an appropriate value.

See Also

closedir(), dirent.h, getdents(), libc, opendir(), readdir(), rewinddir(), telldir()

Notes

telldir() and **seekdir()** are unreliable when the directory stream has been closed and reopened. It is best to avoid using **telldir()** and **seekdir()** altogether.

Because directory entries can dynamically appear and disappear, and because directory contents are buffered by these routines, an application may need to continually rescan a directory to maintain an accurate picture of its active entries.

The COHERENT implementation of the **dirent** routines was written by D. Gwynn.

seg.h — Header File

Definitions used with segmentation
#include <seg.h>

seg.h defines structures and constants used by routines that handle memory segmentation.

See Also

header files

select() — General Function (libsocket)

Check if devices are ready for activity
#include <sys/types.h>
#include <sys/time.h>
#include <sys/select.h>
#include <unistd.h>
int select(*nfds***,** *readfds***,** *writefds***,** *exceptfds***,** *timeout***)**
int *nfds***;**
fd_set **readfds***,* **writefds***,* **exceptfds***;*
struct timeval **timeout***;*

The function **select()** examines file descriptors, and tells you which are ready for a given type of activity. **select()** can be used with descriptors for sockets, pipes, and most character devices including the console, serial terminals connected via the **asy** driver, and pseudoterminals using the **pty** driver.

readfds, *writefds*, and *exceptfds* each gives the address of a a bit-map whose bits correspond to the file descriptors of the sockets that interest you. Respectively, these arguments identify the sockets that may have data to be read, those that to which you wish to write data, and those that may have an exception condition pending. (What an "error condition" may be, is described below.) **select()** examines descriptors zero through *nfds* in each set and checks whether the corresponding socket is ready for the activity in question. If the socket is not ready, **select()** flips off the bits that correspond to that socket.

Please note that although *readfds*, *writefds*, and *exceptfds* each is pointer to **int**, the bit-map it points to can be longer than 32 bits. You can, for example, declare that these pointers points to an array of **int**s. The number of file descriptors you can ask **select()** to examine limited by the manifest constant **FD_SETSIZE**, which is defined in header file **<sys/select.h>**. COHERENT sets this constant to 256; thus, if you set *nfds* to a value greater than 256, only the first 256 file descriptors will be examined.

If you are not interested in a given activity, set the corresponding pointer to NULL. For example, if you are interested only in reading and writing, but not in exception handling, set *exceptfds* to NULL.

timeout gives the address a **timeval** structure that holds the maximum time you are willing to wait for the selection to complete. If it is NULL, **select()** waits indefinitely.

By manipulating the value of *timeout*, you can perform some useful tricks. For example, if you set to zero the fields **tv_sec** and **tv_usec** within the **timeval** structure to which *timeout* points, **select()** performs a nonblocking poll of the indicated devices; this is demonstrated below. Another trick is to set field **tv_usec** within *timeout* to a nonzero value, but set *nfds* to zero. This tells **select()** to examine no sockets, but to wait the specified number of microseconds while not doing it. This lets you "sleep" for an interval shorter than is possible through the system call **sleep()**, whose minimum delay is one second.

If all goes well, **select()** returns the number of sockets that are ready. If the time limit expires, it returns zero. If an error occurs, it leaves all three bit maps unmodified, returns -1, and sets **errno** to one of the following values:

EBADF A descriptor set specifies an invalid descriptor. For example, this error occurs if one of the file descriptors does not describes an ordinary file instead of a socket.

EINTR **select()** received a signal before the time limit expired and before it could finish examining the sockets.

EINVAL
The time structure to which *timeout* points contains invalid data: one of its components is negative or too large.

The following example code demonstrates how to set up a socket and examine it with **select()**. is taken from a program written by Jon Dhuse (jdhuse@sedona.intel.com), and was slightly modified for clarity. The entire program appears in the Lexicon entry **libsocket**:

```
int sd[2], rdfds[2], wrtfds[2], i;
struct timeval timeout;
      ...

/* create socket */
sd = socket(AF_UNIX, SOCK_STREAM, 0);
      ...

/* initialize the arrays of ints */
for (i = 1; i < 2; i++)
      rdfs[i] = 0;
      wrtfds[i] = 0;
}
      ...
```

```
/* Check whether socket is ready */
rdfds[0] = 1 << sd;     /* initialize bit map to check for reading */
wrtfds[0] = 1 << sd;    /* initialize bit map to check for writing */
timeout.tv_sec = 0;
timeout.tv_usec = 0;
i = select(sd+1, rdfds, wrtfds, (int *)NULL, &timeout);
if (i < 0)
        printf("select() returned error %d\n", errno);
else {
      if (rdfds & (1 << sd)) /* check if socket has data */
            printf("socket has data to be read\n");
      if (wrtfds & (1 << sd)) /* check if socket can be written to */
            printf("data can be written to socket\n");
}
```

Associated Macros

The header file **<sys/select.h>** defines the following macros, which are meant to help you manipulate sets of file descriptors:

FD_ZERO (&*fdset*)
Initialize the bit map *fdset* to zero.

FD_SET (*fd*, &*fdset*)
Turn on bit *fd* within the bit map *fdset*.

FD_CLR (*fd*, &*fdset*)
Turn off bit *fd* within the bit map *fdset*.

FD_ISSET (*fd*, *fdset*)
This macro evaluates to a non-zero value if bit *fd* is turned on within *fdset*; otherwise, it evaluates to zero.

The behavior of these macros is undefined if a descriptor's value is less than zero or greater than or equal to **FD_SETSIZE**.

Exception Conditions

As noted above, the bit map *exceptfds* identifies sockets that may have an exception condition pending. As of this writing, COHERENT defines an "exception condition" to be one of the following:

POLLHUP
A hangup has occurred, i.e., loss of carrier on a modem line or closure of the associated master device when **select()** queries a slave pseudo-tty.

POLLNVAL
The file descriptor does not correspond to an open device.

See Also

accept(), connect(), libsocket, poll(), read(), write()

Notes

The system call **poll()** uses a different calling sequence to do much the same work as **socket()**.

sem — Kernel Module

Kernel module for semaphores

The kernel module **sem** enables System V-style semaphores. It is called a *kernel module* because you can link it into your kernel or exclude it, as you wish, just like a device driver; yet it is not a true device devicer because it does not perform I/O with a peripheral device.

See Also

device drivers, kernel, semctl()

sem.h — Header File

Definitions used by semaphore facility
#include <sys/sem.h>

sem.h defines constants and structures used by the COHERENT semaphore facility.

See Also

header files, semget()

semctl() — General Function (libc)

Control semaphore operations
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(*id***,** *number***,** *command***,** *arg***)**
int *id***,** *command***,** *number***;**
union semun {
    int *value***;**
    struct semid_ds **buffer***;*
    unsigned short *array***[];**
} *arg***;**

The function **semctl()** controls the COHERENT system's semaphore facility.

A set of semaphores consists of a copy of structure **semid_ds**, which is defined in header file **<sys/sem.h>**. This structure points to the set of semaphores, notes how many semaphores are in the set, and gives information on who can manipulate it, and how. The semaphores themselves consist of an array of structures of type **sem**, which is also defined in **sem.h**. When the function **semget()** creates a set of semaphores, it assigns to that set an identification number and returns that number to the calling process. For details on this process, see the Lexicon entry for **semget()**

id identifies the set of semaphores to be manipulated. This value must have been returned by a call to **semget()**. *number* gives the offset within the set identified by *id* of the semaphore that interests you. *arg* gives information to be passed to, or received from, the semaphore in question. *command* names the operation that you want **semctl()** to perform.

The following *command*s manipulate semaphore *number* within the set identified by *id*:

GETVAL Return the value of **semval**, which is the field in structure **sem** that gives the address of the semaphore's text map.

SETVAL Set **semval** to *arg.value*. If an "adjust value" had been created for this semaphore (by changing or setting a semaphore through **semop()** with the flag **SEM_UNDO** set), it is erased.

GETPID Return the value of **sempid**, which is the field in **sem** that identifies the last process to have manipulated this semaphore.

GETNCNT Return the value of **semncnt**, which gives the number of processes that await an increase in field **sem.semval**.

GETZCNT Return the value of **semzcnt**, which gives the number of processes that are waiting for the value of **sem.semval** to become zero.

The following *command*s return or set field **semval** within every semaphore in the set identified by *id*:

GETALL Write every **semval** into *arg.array*.

SETALL Initialize every **semval** to the corresponding value within *arg.array*. All "adjust values" for this semaphores are erased.

semctl() also recognizes the following *command*s:

IPC_STAT Copy the value of each semaphore in the set identified by *id* into the structure pointed to by *arg.buffer*.

IPC_SET Copy fields **sem_perm.uid**, **sem_perm.gid**, and **sem_perm.mode** (low nine bits only) from the **ipc_perm** associated with *id* into that pointed to *arg.buffer*. Only the superuser **root** or the user whose effective user ID matches the value of field **uid** in the data structure identified by *id* can invoke this command.

IPC_RMID Destroy the **semid_ds** structure identified by *id*, plus its array of semaphores. Only the superuser **root** or the user whose effective user ID matches the value of field **uid** can invoke this command.

semctl() fails if one or more of the following is true:

- *id* is not a valid semaphore identifier. **semctl()** sets the global variable **errno** to **EINVAL**.
- *number* is less than zero or greater than field **sem_nsems** in structure **semid_ds**, which gives the number of semaphores in the set identified by *id* (**EINVAL**).
- *command* is not a valid command (**EINVAL**).
- The calling process is denied operation permission (**EACCES**).
- *command* is **SETVAL** or **SETALL**, but the value of **semval** exceeds the system-imposed maximum (**ERANGE**).
- *command* is **IPC_RMID** or **IPC_SET**, but the calling process is owned neither by **root** nor by the user who created the set of semaphores being manipulated (**EPERM**).
- *arg.buffer* points to an illegal address (**EFAULT**).

semctl() returns the following values upon successful completion of their following commands:

Command	*Return Value*
GETVAL	Value of **semval**
GETPID	Value of **sempid**
GETNCNT	Value of **semncnt**
GETZCNT	Value of **semzcnt**

For all other commands, **semctl()** returns zero to indicate successful completion.

If it could not execute a command successfully, **semctl()** returns -1 and sets **errno** to an appropriate value.

Files

/usr/include/sys/ipc.h
/usr/include/sys/sem.h

See Also

libc, semget(), semop()

Notes

For information on other methods of interprocess communication, see the Lexicon entries for **msgctl()** and **shmctl()**.

semget() — General Function (libc)

Create or get a set of semaphores
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
semget(*semkey***,** *number***,** *flag***)**
key_t *semkey***; int** *number***,** *flag***;**

semget() creates a set of semaphores plus its associated data structure and identifier, links them to the identifier *semkey*, and returns the identifier that it has associated with *semkey*.

semkey is an identifier that your application generates to identify its semaphores.

number gives the of semaphores you want **shmget()** to create.

flag can be bitwise OR'd to include the following constants:

IPC_ALLOC This process already has a set of semaphores; please fetch it.

IPC_CREAT If this process does not have a set of semaphores, please create one.

IPC_EXCL Fail if this process already has a set of semaphores.

IPC_NOWAIT Fail if the process must wait to obtain a set of semaphores.

When it creates a set of semaphores, **semget()** also creates a copy of structure **semid_ds**, which the header file **<sys/sem.h>** defines as follows:

```
struct semid_ds {
        struct ipc_perm sem_perm;         /* operation permission struct */
        struct sem *sem_base;             /* pointer to first semaphore in set */
        unsigned short sem_nsems;         /* # of semaphores in set */
        time_t sem_otime;                 /* last semop time */
        time_t sem_ctime;                 /* last change time */
};
```

Field **sem_base** points the semaphores themselves. Each semaphore is a structure of type **sem**, which header file **<sys/sem.h>** defines as follows:

```
struct sem {
        unsigned short semval;            /* semaphore text map address */
        short sempid;                     /* pid of last operation */
        unsigned short semncnt;           /* # awaiting semval > cval */
        unsigned short semzcnt;           /* # awaiting semval = 0 */
};
```

Field **sem_perm** is a structure of type **ipc_perm**, which header file **<sys/ipc.h>** defines as follows:

```
struct ipc_perm {
        unsigned short uid;               /* owner's user id */
        unsigned short gid;               /* owner's group id */
        unsigned short cuid;              /* creator's user id */
        unsigned short cgid;              /* creator's group id */
        unsigned short mode;              /* access modes */
        unsigned short seq;               /* slot usage sequence number */
        key_t key;                        /* key */
};
```

semget() initializes **semid_ds** as follows:

- It sets the fields **sem_perm.cuid**, **sem_perm.uid**, **sem_perm.cgid**, and **sem_perm.gid** to, respectively, the effective user and group identifiers of the calling process.
- It sets the low-order nine bits of **sem_perm.mode** to the low-order nine bits of *flag*. These nine bits define access permissions: the top three bits give the owner's access permissions (read, write, execute), the middle three bits the owning group's access permissions, and the low three bits access permissions for others.
- It sets **sem_nsems** to *number*. This gives the number of semaphores to which **sem_base** points.
- It sets field **sem_otime** to zero, and field **sem_ctime** to the current time.

semget() fails if any of the following are true:

- *number* is less than one and the set of semaphores identified by *semkey* does not exist. **semget()** sets **errno** to **EINVAL**.
- *number* exceeds the system-imposed limit (**EINVAL**).
- A semaphore identifier exists for *semkey*, but permission, as specified **flag**'s low-order nine bits, is not granted (**EACCES**).
- A semaphore identifier exists for *semkey*, but the number of semaphores in its set is less than *number*, and *number* does not equal zero (**EINVAL**).
- A semaphore identifier does not exist for *semkey* and (*flag* & **IPC_CREAT**) is false (**ENOENT**).
- **semget()** tried to create a set of semaphores, but could not because the maximum number of sets allowable by the system always exists (**ENOSPC**).

- A semaphore identifier already exists for *semkey* but *flag* requests that **semget()** create an exclusive set for it — i.e.

  ```
  ( (flag & IPC_CREAT) && (flag & IPC_EXCL) )
  ```

  is true (**EEXIST**).

If all goes well, **semget()** returns a semaphore identifier, which is always a non-negative integer. Otherwise, it returns -1 and sets **errno** to an appropriate value.

Files

/usr/include/sys/ipc.h
/usr/include/sys/sem.h

See Also

ftok(), ipcrm, ipcs, libc, libsocket, semctl(), semop()

Notes

Prior to release 4.2, COHERENT implemented semaphores through the driver **sem**. In release 4.2, and subsequent releases, COHERENT has implemented semaphores as a set of functions that conform in large part to the UNIX System-V standard.

The kernel variables **SEMMNI** and **SHMMNS** set, respectively, the maximum number of identifiers that can exist at any given time and the maximum number of semaphores that a set can hold. Daredevil system operators who have large amounts of memory at their disposal may wish to change these variables to increase the system-defined limits. For details on how to do so, see the Lexicon entry **mtune**.

semop() — General Function (libc)

Perform semaphore operations
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(*id***,** *operation***,** *nops***)**
int *id***,** *nops***; struct sembuf** *operation***[];**

semop() performs semaphore operations.

id identifies the set of semaphores to be manipulated. It must have been returned by a call to **semget()**.

nops gives the number of structures in the array pointed to by *operation*.

operation points to an array of structures of type **sembuf**, which the header file **sem.h** defines as follows:

```
struct sembuf {
        unsigned short sem_num;                 /* semaphore # */
        short sem_op;                           /* semaphore operation */
        short sem_flg;                          /* operation flags */
};
```

Each **sembuf** describes a semaphore operation. Field **sem_op** identifies the operation to perform on the semaphore in the set identified by *id* and with offset *sem_num*. **sem_op** specifies one of three semaphore operations, as follows:

1. If **sem_op** is negative, one of the following occurs:

   A. If **semval** in the semaphore structure identified by *id* is greater than or equal to the absolute value of **sem_op**, **semop()** subtracts the absolute value of **sem_op** from **semval**.

   B. If **semval** is less than the absolute value of **sem_op** and (**sem_flg** & **IPC_NOWAIT**) is true, **semop()** sets **errno** to **EGAIN** and immediately returns -1.

   C. If **semval** is less than the absolute value of **sem_op** and (**sem_flg** & **IPC_NOWAIT**) is false, then **semop()** increments the **semncnt** associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:

      a. **semval** equals or exceeds the absolute value of **sem_op**. When this occurs, **semop()** decrements the value of **semncnt** associated with the specified semaphore, and subtracts the absolute value of **sem_op** from **semval**.

b. The *id* for which the calling process is awaiting action is removed from the system.

c. The calling process receives a signal. When this occurs, **semop()** decrements the field **semncnt** in the **sem** structure that *id* identifies, and the calling process resumes execution in the manner defined by the signal. (See the Lexicon entry for **signal()** for details of what behavior each signal initiates.)

2. If **sem_op** is positive, **semop()** adds **sem_op** to **semval**.

3. If **sem_op** is zero, one of the following occurs:

A. If **semval** is zero, **semop()** returns immediately.

B. If **semval** does not equal zero and (**sem_flg** & **IPC_NOWAIT**) is true, **semop()** sets **errno** to **EGAIN**, and immediately returns -1.

C. If **semval** does not equal zero and (**sem_flg** & **IPC_NOWAIT**) is false, **semop()** increments the **semzcnt** associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:

a. **semval** becomes zero. **semop()** decrements the value of the field **semzcnt** associated with the specified semaphore.

b. The set of semaphores identified by *id* is removed from the system.

c. The calling process receives a signal. **semop()** then decrements the value of the **semzcnt** associated with the specified semaphore, and the calling process resumes execution in the manner prescribed by the signal.

If field **sem_flg** in a **sembuf** structure contains value **SEM_UNDO** (i.e., expression (**sem_flg** & **SEM_UNDO**) is true) then the system stores an *adjust value* for this semaphore operation for this semaphore and links it to the process that has invoked **semop()**. The adjust value is the inversion of this semaphore operation; when the process dies, the system executes these adjust values, to undo each of these semaphore operations. If you use the function **semctl()** to change the value of a semaphore or a set of semaphores, then the system erases all adjust values for those semaphores.

semop() returns -1 and sets **errno** to the value in parentheses if any of the following error conditions occurs:

- *id* is not a valid semaphore identifier (**EINVAL**).
- **sem_num** is less than zero or greater than or equal to the number of semaphores in the set associated with *id* (**EFBIG**).
- *nops* exceeds the system-imposed maximum (**E2BIG**).
- Permission is denied to the calling process (**EACCES**).
- *operation* would suspend the calling process but (**sem_flg** & **IPC_NOWAIT**) is true (**EAGAIN**).
- *operation* would cause **semval** to overflow the system-imposed limit (**ERANGE**).
- *operation* points to an illegal address (**EFAULT**).
- The calling processing receives a signal (**EINTR**).
- The set of semaphores identified by *id* has been removed from the system (**EDOM**).

If all goes well, **semop()** sets the **sempid** of each semaphore specified in the array pointed to by *operation* to the process identifier of the calling process. It then returns the value that **semval** had had at the time that the last operation in the array pointed to by *operation* was executed.

Files

/usr/include/sys/ipc.h
/usr/include/sys/sem.h

See Also

libc, semctl(), semget()

Notes

The COHERENT implementation of semaphores does not permit a process to lock or unlock a semaphore unless it can gain access to all of the semaphores that it requests. This is to prevent the situation in which two processes have each locked semaphores that the other wants, and each has **IPC_NOWAIT** set to false — thus suspending each other forever.

send() — Sockets Function (libsocket)

Send a message to a socket
#include <sys/socket.h>
#include <sys/types.h>
int send(*socket***,** *message***,** *length***,** *flags***)**
int *socket***;**
char **message***;*
int *length***,** *flags***;**

The function **send()** sends a message to a socket.

socket is the socket to which the messages are sent. It must have been created by the function **socket()**, and connected by the function **connect()**. *buffer* points to the chunk of memory that holds the message to be sent; *length* gives the amount of allocated memory to which *buffer* points.

flags ORs together either or both of the following flags:

MSG_OOB
Send "out-of-band" data on sockets that support this notion. The underlying protocol must also support out-of-band data.

MSG_DONTROUTE
The socket turned on for the duration of the operation. It is used only by diagnostic or routing programs.

If all goes well, **send()** returns the number of bytes it sent. If something goes wrong, it returns -1 and sets **errno** to one of the following values:

EAGAIN
If *socket* has no buffer space available, **send()** normally waits until space becomes available (which is a blocking operation). *socket*, however, is marked as non-blocking.

EBADF *socket* does not identify a valid socket.

EINTR A signal interrupted **send()** before it could send any data.

EMSGSIZE
socket requires that message be sent atomically, and the message was too long.

ENOMEM
Insufficient user memory was available to complete the operation.

ENOTSOCK
socket describes a file, not a socket.

EPROTO
A protocol error has occurred.

See Also

connect(), libsocket, recv(), sendto(), socket()

sendto() — Sockets Function (libsocket)

Send a message to a socket
#include <sys/socket.h>
#include <sys/types.h>
int send(*socket***,** *message***,** *length***,** *flags***,** *addr***,** *alen***)**
int *socket***;** char **message***;* **int** *length***,** *flags***;**
sockaddr_t **addr***;* **int** *alen***;**

The function **sendto()** sends a message to a socket. Unlike the related function **sendto()**, it works regardless of whether the socket is connected.

socket is the socket to which the messages are sent. It must have been created by the function **socket()**. *buffer* points to the chunk of memory into which the message is to be written; *length* gives the amount of allocated memory to which *buffer* points. If *from* is not NULL, **sendto()** initializes it to the the source address of the message. It initializes *alen* to the size of the buffer associated with *address*, and modifies it upon return to the size of the address stored there.

flags ORs together either or both of the following flags:

MSG_OOB
Send "out-of-band" data on sockets that support this notion. The underlying protocol must also support out-of-band data.

MSG_DONTROUTE
The socket turned on for the duration of the operation. It is used only by diagnostic or routing programs.

If all goes well, **sendto()** returns the number of bytes it sent. If something goes wrong, it returns -1 and sets **errno** to one of the following values:

EAGAIN
If *socket* has no buffer space available, **sendto()** normally waits until space becomes available (which is a blocking operation). *socket*, however, is marked as non-blocking.

EBADF *socket* does not identify a valid socket.

EINTR A signal interrupted **sendto()** before it could send any data.

EMSGSIZE
socket requires that message be sent atomically, and the message was too long.

ENOMEM
Insufficient user memory was available to complete the operation.

ENOTSOCK
socket describes a file, not a socket.

EPROTO
A protocol error has occurred.

See Also

connect(), libsocket, recv(), send(), socket()

Notes

At present, the COHERENT implementation of **sendto()** always behaves as if *address* were initialized to NULL.

serialno — System Administration

Hold the serial number of your system
/etc/serialno

The file **/etc/serialno** holds your system's serial number. This is the number assigned to your system when you installed COHERENT onto your computer. You need this number when you update your COHERENT system, or when you contact Mark Williams Company.

See Also

Administering COHERENT

services — System Administration

List supported TCP/IP services
/etc/services

The file **/etc/services** names the services provided by TCP/IP and related protocols.

Each line within this file describes one services. A line consists of four fields, which respectively give the official service name, well-known port number by which it is accessed, the name of its protocol, and any aliases by which it is known. For example:

```
smtp            25/tcp          mail
time            37/tcp          timserver
time            37/udp          timserver
```

Fields are separated by white space, with the exception of fields that give the port and the protocol name; these are separated by virgule '/'. The fourth, aliases field is optional. A pound-sign character '#' introduces a comment; all text from that character to the end of the line is ignored.

See Also

Administering COHERENT, hosts, hosts.equiv, inetd.conf, networks, protocols

set — Command

Set shell option flags and positional parameters
set [-ceiknstuvx [*name ...***]]** (Bourne shell)
set [[+-]aefhkmnuvx] [[+-]o *name***]** (Korn shell)

set changes the options of the current shell and optionally sets the values of positional parameters. This command is used implemented by both the Bourne and Korn shells; however, its syntax and options vary from one shell to the other.

Bourne Shell

The shell variable '$-' contains the currently set shell flags. If the optional *name* list is given, **set** assigns the positional parameters **$1**, **$2** ... to the given shell variables.

set recognizes the following options:

-c *string*
Read shell commands from *string.*

-e Exit on any error (command not found or command returning nonzero status) if the shell is not interactive.

-i The shell is interactive, even if the terminal is not attached to it; print prompt strings. For a shell reading a script, ignore signals **SIGTERM** and **SIGINT.**

-k Place all keyword arguments into the environment. Normally, the shell places only assignments to variables preceding the command into the environment.

-n Read commands but do not execute them.

-s Read commands from the standard input and write shell output to the standard error.

-t Read and execute one command rather than the entire file.

-u If the actual value of a shell variable is blank, report an error rather than substituting the null string.

-v Print each line as it is read.

-x Print each command and its arguments as it is executed.

- Cancel the **-x -v** options.

The shell executes **set** directly.

Korn Shell

set recognizes the following options. Preceding an option with '-' turns on the option; preceding it with '+' turns it off.

-a **allexport**: Automatically export all new variables.

-e **errexit**: Exit from the shell when non-zero status is received.

-f **noglob**: Do not expand file names. This globally turns off the special meaning of characters '*' and '?'.

-h **trackall**: Automatically add all commands to the shell's hash table.

-k **keyword**: Recognize variable assignments anywhere in a command.

-m **monitor**: Enable job control. See the Lexicon article on **ksh** for details on job control and how to use it.

-n **noexec**: Compile an input command, but do not execute it.

-o *option*
Set *option*. **set** recognizes the following *option*s:

allexport	Same as **-a** option, above.
emacs	Turn on MicroEMACS-style editing of command lines.
errexit	Same as **-e** option, above.
ignoreeof	Tell the shell not to exit when reading EOF: must use **exit** command to exit from the shell.
keyword	Same as **-k** option, above.
monitor	Same as **-m** option, above.
noexec	Same as **-n** option, above.
noglob	Same as **-f** option, above.
trackall	Same as **-h** option, above.
nounset	Same as **-u** option, below.
verbose	Same as **-v** option, below.
xtrace	Same as **-x** option, below.

-u **nounset**: Treat dollar-sign expansion of an unset variable as an error.

-v **verbose**: When compiling a command, echo its compiled (i.e., expanded) version on the standard output before executing it.

-x **xtrace**: Echo simple commands while executing.

The shells execute **set** directly.

See Also

commands, ksh, sh, unset

setbuf() — STDIO Function (libc)

Set alternative stream buffer
#include <stdio.h>
void setbuf(*fp*, *buffer***)**
FILE **fp***; **char** **buffer***;*

The standard I/O library STDIO automatically buffers all data read and written in streams, with the exception of streams to terminal devices. STDIO normally uses **malloc()** to allocate the buffer, which is a **char** array **BUFSIZ** characters long; **BUFSIZ** is a manifest constant defined in the header file **stdio.h**.

setbuf()'s arguments are the file stream *fp* and the *buffer* to be associated with the stream. The call should be issued after the stream has been opened, but before any input or output request has been issued. If *buffer* is NULL, the stream will be unbuffered. If *buffer* is not NULL, the arena of memory it points to must contain at least **BUFSIZ** bytes.

setbuf() returns nothing.

See Also

fopen(), libc, malloc(), setvbuf()
ANSI Standard, §7.9.5.5
POSIX Standard, §8.1

setgid() — System Call (libc)

Set group id and user id
#include <unistd.h>
int setgid(*id***) int** *id***;**

The *group identifier* is the number that identifies the user group that "owns" a given file. File **/etc/group** establishes the set of groups that your COHERENT system recognizes. (For details on how this file is laid out, see the Lexicon entry for **group**). When a file is executable, the executing process inherits its group identifier (and thus, its group-level permissions) from the file in which it resides on disk. For example, the program **troff** resides in file **/bin/troff**. This file is "owned" by group **bin**; thus, when you execute **troff**, its group-level permissions are those of group **bin**.

The group identifier comes in three forms:

real This is the group identifier of the user who is running the process.

effective
This is the group identifier that determines the access rights of the process. These rights are the same as those of the real group identifier unless they have been altered by executing a file whose **setgid** bit is set. For example, the program **troff** does *not* have the setgid bit set; thus, when you execute **troff**, the group permissions of the **troff** process remain those of your group, not those of the group **bin**. On the other hand, the program **/usr/lib/uucp/uucico** does have the setgid bit set; thus, when you invoke **uucico**, the **uucico** process uses the permissions of **uucico**'s group (that is, of group **uucp**), instead of your group.

saved effective
This permits a process to step back and forth between its real and effective group identifiers. If you return from an effective group identifier to your real one, the system saves the previously effective identifier so you can revert to it if need be.

The system call **setgid()** lets you set the real and effective group identifiers of the calling process to the group identifier *gid*. The behavior of **setgid()** varies depending upon the following:

1. If **setgid()** is invoked by a user whose effective user identifier is that of the superuser **root**, **setgid()** sets the real, effective, and saved effective group identifiers to *gid*.
2. If **setgid()** is invoked by a user whose real group identifier is the same as *gid*, **setgid()** sets the effective group identifier to *gid*.
3. If **setgid()** is invoked by a user whose saved effective group identifier is same as *gid*, **setgid()** sets the effective group identifier to *gid*.

If all goes well, **setgid()** returns zero. If a problem arises, it returns -1.

See Also

execution, getuid(), libc, login, setuid(), unistd.h
POSIX Standard, §4.2.2

setgrent() — General Function (libc)

Rewind group file
#include <grp.h>
void setgrent();

setgrent() rewinds the file **/etc/group**. It returns nothing.

Files

/etc/group
<grp.h>

See Also

group, libc

setgroups() — System Call (libc)

Set the supplemental group-access list
#include <unistd.h>
int setgroups(*ngroups***,** *grouplist***)**
int *ngroups***; const gid_t** ******grouplist***;**

The "supplemental group-access list" is the list of group identifiers that are used in addition to the effective group identifier when determining the level of access that a process has to a file. **setgroups()** fills the calling process's supplemental group-access list with the group identifiers in the array to which *grouplist* points. *ngroups* gives the number of identifiers in the array, and cannot exceed **NGROUPS_MAX**.

If all goes well, **setgroups()** returns zero. It fails and returns -1 if any of the following occur:

- The value of *ngroups* exceeds **NGROUPS_MAX**. **setgroups** sets **errno** to **EINVAL**.
- The effective user identifier is not that of the super-user **root**. **setgroups()** sets **errno** to **EPERM**.

- *grouplist* contains an illegal address. **setgroups()** sets **errno** to **EFAULT**.

See Also

getgroups(), initgroups(), libc, limits.h, unistd.h

Notes

This function may be invoked only by the superuser **root**.

sethostent() — Sockets Function (libsocket)

Open and rewind file /etc/hosts
#include <netdb.h>
void sethostent(*stayopen***)**
int *stayopen***;**

The function **sethostent()** is one of a set of functions that interrogate the file **/etc/hosts** to look up information about a remote host on a network. It opens and rewinds **/etc/hosts**.

Flag *stayopen* indicates whether **/etc/hosts** is to stay open after it has been interrogated by **gethostbyaddr()** or **gethostbyname()**: if it is zero, then **/etc/hosts** is closed after it is interrogated; if it is nonzero, then **/etc/hosts** remains open.

See Also

endhostent(), gethostbyaddr(), gethostbyname(), libsocket

setjmp() — General Function (libc)

Save machine state for non-local goto
#include <setjmp.h>
int setjmp(*env***) jmp_buf** *env***;**

The function call is the only mechanism that C provides to transfer control between functions. This mechanism, however, is inadequate for some purposes, such as handling unexpected errors or interrupts at lower levels of a program. To answer this need, **setjmp** helps to provide a non-local *goto* facility. **setjmp()** saves a stack context in *env*, and returns value zero. The stack context can be restored with the function **longjmp()**. The type declaration for **jmp_buf** is in the header file **setjmp.h**. The context saved includes the program counter, stack pointer, and stack frame.

Example

The following gives a simple example of **setjmp()** and **longjmp()**.

```
#include <setjmp.h>
#include <stdio.h>

jmp_buf env;      /* place for setjmp to store its environment */

main()
{
      int rc;

      if(rc = setjmp(env)) { /* we come here on return */
            printf("First char was %c\n", rc);
            exit(EXIT_SUCCESS);
      }
      subfun();   /* this never returns */
}

subfun()
{
      char buf[80];

      do {
            printf("Enter some data\n");
            gets(buf);  /* get data */
      } while(!buf[0]); /* retry on null line */

      longjmp(env, buf[0]);   /* buf[0] must be non zero */
}
```

See Also

getenv(), libc, longjmp(), sigsetjmp()
ANSI Standard, §7.6.1.1
POSIX Standard, §8.1

Notes

Programmers should note that many user-level routines cannot be interrupted and reentered safely. For that reason, improper use of **setjmp()** and **longjmp()** can create mysterious and irreproducible bugs. The use of **longjmp()** to exit interrupt exception or signal handlers is particularly hazardous.

setjmp.h — Header File

Define setjmp() and longjmp()
#include <setjmp.h>

setjmp.h defines the structure **jmp_buf** for a **setjmp()** environment.

See Also

header file, longjmp(), setjmp()
ANSI Standard, §7.6

setnetent() — Sockets Function (libsocket)

Open and rewind file /etc/networks

#include <netdb.h> int setnetent(*stayopen***) int** *stayopen***;**

Function **setnetent()** opens or rewinds file **/etc/networks**, which describes all entities on your local network. If flag *stayopen* is set to a non-zero value, **/etc/networks** is not closed after each call to **getnetbyaddr()** or **getnetbyname()**.

See Also

endnetent(), getnetbyaddr(), getnetbyname(), getnetent(), libsocket, netdb.h

setpgid() — System Call (libc)

Set the process-group identifier
#include <sys/types.h>
#include <unistd.h>
int setpgid(*pid***,** *pgid***)**
pid_t *pid***,** *pgid***;**

setpgid() sets to *pgid* the process-group identifier of the process with identifier *pid*. If *pgid* equals *pid*, the process becomes a process-group leader. If *pgid* does not equal *pid*, the process becomes a member of an existing process group.

If *pid* equals zero, **setpgid()** uses the process identifier of the calling process. If *pgid* equals zero, the process specified by *pid* becomes a process-group leader.

If all goes well, **setpgid()** returns a value of zero. Otherwise, it returns -1 and sets **errno** to an appropriate value. **setpgid()** if any of the following are true:

- *pid* matches the process identifier of a child process of the calling process, and that child process has successfully executed an **exec()** function. **setpgid()** sets **errno** to **EACCES**.
- *pgid* is less than zero or greater than or equal to **PID_MAX**. **setpgid()** sets **errno** to **EINVAL**.
- The calling process has a controlling terminal that does not support job control. **setpgid()** sets **errno** to **EINVAL**.
- The process identified by *pid* argument is a session leader. **setpgid()** sets **errno** to **EPERM**.
- *pid* equals the process identifier of a child process of the calling process, and the child process is not in the same session as the calling process. **setpgid()** sets **errno** to **EPERM**.
- *pgid* does not match the process identifier of the process indicated by *pid*, and the call process's session has no process with a process-group identifier that equals *pgid*. **setpgid()** sets **errno** to **EPERM**.

- *pid* does not match the process identifier of the calling process or of a child process of the calling process. **setpgid()** sets **errno** to **ESRCH**.

See Also

libc, unistd.h
POSIX Standard, §4.3.3

setpgrp() — System Call (libc)

Make a process a process-group leader
int setpgrp()

setpgrp() sets the requesting process's process-group identifier to its own process identifier. This detaches the process from its parent group and makes it the leader of its own processing group. If the process is not already a process-group leader, it is detached from its controlling terminal.

setpgrp() returns the new process-group identifier.

See Also

getpgrp(), libc

Notes

This function is obsolete, and is being phased out in favor of the function **setsid()**.

setprotoent() — Sockets Function (libsocket)

Open the protocols file

#include <netdb.h> int setprotoent(*stayopen***) int** *stayopen***;**

Function **setprotoent()** opens or rewinds file **/etc/protocols**, which describes all protocols recognized on your local network. If flag *stayopen* is set to a non-zero value, **/etc/protocols** is not closed after each call to **getprotobyaddr()** or **getprotobyname()**.

See Also

getprotobyaddr(), getprotobyname(), getprotoent(), endprotoent(), libsocket, netdb.h

setpwent() — General Function (libc)

Rewind password file
#include <pwd.h>
setpwent()

The COHERENT system has five routines that search the file **/etc/passwd**, which contains information about every user of the system. **setpwent()** rewinds the password file, which allows searches to begin from the beginning of the file. Please note that this function does not return a meaningful value.

Example

For an example of this function, see the entry for **getpwent()**.

Files

/etc/passwd
pwd.h

See Also

libc

setservent() — Sockets Function (libsocket)

Open the services file

#include <netdb.h> int setservent(*stayopen***) int** *stayopen***;**

Function **setservent()** opens or rewinds file **/etc/services**, which describes the services offered by TCP/IP on your local network. If flag *stayopen* is set to a non-zero value, **/etc/services** is not closed after each call to **getservbyport()** or **getservbyname()**.

See Also

getservbyname(), getservbyport(), getservent(), endservent(), libsocket, netdb.h

setsid() — System Call (libc)

Set session identifier
#include <sys/types.h>
#include <unistd.h>
pid_t setsid ();

If the calling process is not a process-group leader, **setsid()** sets its process-group and session identifiers to its process identifier, and releases the its controlling terminal.

If all goes well, **setsid()** returns the calling process's session identifier. If the calling process is already a process-group leader, or if process-group identifier of another process equals that of the calling process, **setsid()** returns -1 and sets **errno** to **EPERM**.

See Also

libc, unistd.h
POSIX Standard, §4.3.2

Notes

If the calling process is the last member of a pipeline started by a job-control shell, the shell may make the calling process a process-group leader. The other processes of the pipeline become members of that process group. If this happens, the call to **setsid()** fails.

For this reason, a process that calls **setsid()** and expects to be part of a pipeline should first fork: the parent should exit and the child should call **setsid()**. This will ensure that the process works reliably when started by both job-control shells and non-job-control shells.

setsockopt() — Sockets Function (libsocket)

Set a socket option
#include <sys/types.h>
#include <sys/socket.h>
int setsockopt(*socket***,** *level***,** *option***,** *buffer***,** *length***)**
int *socket***,** *level***,** *option***,** *length***;**
char **buffer***;*

Function **setsockopt()** sets options on a socket.

socket gives the identifier of the socket, as returned by the function **socket()**.

level gives the level at which the options are set. To retrieve options set on the socket level, set *level* to **SOL_SOCKET** whereas to retrieve options set the TCP level, set *level* to the number of the TCP protocol.

option gives the number of the option to set. A list of options that are recognized at the socket level appears below. Options at other levels are set by their respective protocols.

buffer gives the address of the buffer that holds the option. *length* gives the length of *buffer*, in bytes.

The following options are recognized at the socket level. They are set in header file **<sys/socket.h>**:

SO_BROADCAST
Toggle permission to transmit broadcast messages.

SO_KEEPALIVE
Toggle whether to keep a connection alive by periodically transmitting messages. If the connected party fails to respond to a message, the connection is considered broken and processes that use the socket are notified via the signal **SIGPIPE**.

SO_LINGER
Control the action taken when a socket is closed but contains unsent messages. If **SO_LINGER** is set and the socket promises reliable delivery of data, the system blocks the process that is attempting to close *socket* until *socket* can transmit its data or its attempts to do so time out. If **SO_LINGER** is not enabled, the socket is closed immediately and the unsent messages are thrown away.

SO_OOBINLINE
Toggle whether a band can receive out-of-band data. Such data can then be read by the function **recv()** or sent by the function **send()**, when invoked with the flag **MSG_OOB**.

SO_RCVBUF
SO_SNDBUF
Set the size of the receive or send buffer, respectively. You can increase the size of a buffer to speed high-volume connections, or decrease it to limit the amount of data that are backlogged. The system places an absolute limit on these values.

SO_REUSEADDR
Toggle whether local addresses can be reused.

If all goes well, **setsockopt()** returns zero. If something goes wrong, it returns -1 and set **errno** to one of the following values:

EBADF *socket* does not identify a valid socket.

ENOMEM
The available user memory was insufficient to complete the operation.

ENOPROTOOPT
option gives an unknown option.

ENOTSOCK
socket identifies a file, not a socket.

See Also

getsockopt(), libsocket

setspent() — General Function (libc)

Rewind the shadow-password file
#include <shadow.h>
setspent()

The COHERENT system has four routines that search the file **/etc/shadow**, which contains the password of every user of the system. **setspent()** rewinds the password file — that is, it resets the seek pointer so that subsequent searches of the file start at the beginning of the file. This function does not return a meaningful value.

See Also

endspent(), getspent(), libc, shadow, shadow.h

setuid() — System Call (libc)

Set user identifier
#include <unistd.h>
int setuid(*id***)**
int *id***;**

The *user identifier* is the number that identifies the user who "owns" a given file. The suite of users is defined in file **/etc/passwd**. When a file is executable, the executing process inherits its user identifier (and thus, its user-level permissions) from the file where it lives on disk. The user identifier comes in three forms:

real This is the identifier of the user who is running the process.

effective
This is the user identifier that determines the access rights of the process. These rights are the same as those of the real user identifier unless they have been altered by executing a file whose **setuid** bit is set.

saved effective
This permits a process to step back and forth between its real and effective user identifiers. If you return from an effective user identifier to your real one, the previously effective id is saved so you can revert to it if need be.

The system call **setuid()** lets you alter the real and effective user identifiers of the calling process to the user identifier *uid*. The behavior of **setuid()** varies depending upon the following:

1. If the effective user identifier is that of the superuser, **setuid()** sets the real, effective, and saved effective user identifiers to *uid*.
2. If the real user identifier is the same as *uid*, **setuid()** sets the effective user identifier to *uid*.
3. If the saved effective user identifier is same as *uid*, **setuid()** sets the effective user identifier to *uid*.

To **setuid** an existing executable file, use the command **chmod**.

See Also

chmod, execution, getuid(), libc, login, setgid(), unistd.h
POSIX Standard, §4.2.2

Diagnostics

setuid() returns zero on success, or -1 on failure.

Notes

For more information on the user id, see the Lexicon entry for **execution**.

setupterm() — terminfo Function

Initialize a terminal
#include <curses.h>
setupterm(*term*,*fd*,*errret***)**
char **term*;
int *fd*, **errret*;

COHERENT comes with a set of functions that let you use **terminfo** descriptions to manipulate a terminal. **setupterm()** initializes terminal capabilities for terminal type *term*, which is accessed via file-descriptor *fd*. It inhales all capabilities at once, and performs all other system-dependent initialization — which is one reason why **terminfo** is much faster than **termcap**.

If *term* is initialized to NULL, **setupterm()** uses the contents of the environmental variable **TERM** as a default.

errret points to an integer into which **setupterm()** writes the terminal's status: zero if there is no such terminal type, one if all went well, or -1 if something has gone wrong. If *erret* is NULL, **setupterm()** prints an error message and exits if the terminal cannot be found.

See Also

terminfo

setutent() — General Function (libc)

Rewind the input stream for a login logging file
#include <utmp.h>
void setutent()

Function **setutent()** rewinds the input stream that is reads the file that records login events. This lets you search this file multiple times without having to close and reopen it.

By default, **setutent()** manipulates a stream that reads file **/etc/utmp**. If you wish to manipulate another logging file, use the function **utmpname()**.

See Also

libc, utmp.h

setvbuf() — STDIO Function (libc)

Set alternative file-stream buffer
#include <stdio.h>
int setvbuf(*fp*, *buffer*, *mode*, *size***)**
FILE **fp*; **char** **buffer*; **int** *mode*; **size_t** *size*;

When the functions **fopen()** and **freopen()** open a stream, they automatically establish a buffer for it. The buffer is **BUFSIZ** bytes long. **BUFSIZ** is a manifest constant that is defined in the header **stdio.h**.

The function **setvbuf()** alters the buffer used with the stream pointed to by *fp* from its default buffer to *buffer*. Unlike the related function **setbuf()**, it also allows you set the size of the new buffer as well as the form of buffering.

buffer is the address of the new buffer. *size* is its size, in bytes. *mode* is the manner in which you wish the stream to be buffered, as follows:

_IOFBF	Fully buffered
_IOLBF	Line-buffered
_IONBF	No buffering

These constants are defined in the header **stdio.h**.

You should call **setvbuf()** after a stream has been opened but before any data have been written to or read from the stream. For example, the following give *fp* a 50-byte buffer that is line-buffered:

```
char buffer[50];
FILE *fp;

fopen(fp, "r");
setvbuf(fp, buffer, _IOLBF, sizeof(buffer));
```

On the other hand, the following turns off buffering for the standard output stream:

```
setvbuf(stdout, NULL, _IONBF, 0);
```

setvbuf() returns zero if the new buffer could be established correctly. It returns a value other than zero if something went wrong or if an invalid parameter is given for *mode* or *size*.

Example

This example uses **setvbuf()** to turn off buffering and echo.

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>

main(void)
{
      int c;

      if(setvbuf(stdin,  NULL, _IONBF, 0))
            fprintf(stderr, "Couldn't turn off stdin buffer\n");

      if(setvbuf(stdout,  NULL, _IONBF, 0))
            fprintf(stderr, "Couldn't turn off stdout buffer\n");

      while((c = getchar()) != EOF)
            putchar(c);
}
```

See Also

fclose(), fflush(), fopen(), freopen(), libc, setbuf()
ANSI Standard, §7.9.5.6

Notes

setvbuf() affects the buffering of an I/O stream but does not affect any buffering that performed by the device upon which the text is typed. Some devices (e.g., **/dev/tty**) are buffered by default. To turn off the buffering of what a user types, you must both turn off buffering on the input stream and turn off buffering on the device itself. For example, to turn off buffering on a terminal device, you must both call **setvbuf()** to change the size of the input buffering to zero, and call **stty()** to put the terminal device into raw mode.

sgtty — Device Driver

General terminal interface

COHERENT uses two method for controlling terminals: **sgtty** and **termio**. To use **sgtty**, simply include the statement **#include <sgtty.h>** in your sources. To use **termio**, include the statement **#include <termio.h>**.

The rest of this article discusses the **sgtty** method of controlling terminals.

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, users' programs seldom open these files; they are opened by the program *getty* and become a user's standard input, output, and error files. The very first terminal file opened by the process group leader of a terminal

file not already associated with a process group becomes the *controlling terminal* for that process group. The controlling terminal plays a special role in handling **quit** and **interrupt** signals, as discussed below. The controlling terminal is inherited by a child process during a call to **fork**. A process can break this association by changing its process group using **setpgrp**.

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters can be typed at any time, even while output is occurring, and are only lost when the system's input buffers become completely full, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. Currently, this limit is 256 characters. When the input limit is reached, the sytems throws away all the saved characters without notice.

Normally, terminal input is processed in units of lines. A line is delimited by a newline character (ASCII LF) or an end-of-file character (ASCII EOT). Unless otherwise directed, a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, the system normally processes **erase** and **kill** characters. By default, the backspace character erases the last character typed, except that it will not erase beyond the beginning of the line. By default, the **<ctrl-U>** kills (deletes) the entire input line, and optionally outputs a newline character. Both these characters operate on a keystroke-by-keystroke basis, independently of any backspacing or tabbing which may have been done. Both the erase and kill characters may be entered literally by preceding them with the escape character (\). In this case, the escape character is not read. You may change the erase and kill characters via command **stty**.

Certain characters have special functions on input. These functions and their default character values are summarized as follows:

INTR (**<ctrl-C>** or ASCII ETX) generates an *interrupt* signal that is sent to all processes associated with the controlling terminal. Normally, each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location; see the Lexicon entry for **signal**.

QUIT (Control-\ or ASCII ES) generates a *quit* signal. Its treatment is identical to that of the interrupt signal except that, unless a receiving process has made other arrangements, it will not only be terminated but a core image file (called **core**) will be created in the current working directory.

ERASE (**<backspace>** or ASCII BS) erases the preceding character. It will not erase beyond the start of a line, as delimited by a newline or **EOF** character.

KILL (**<ctrl-U>** or ASCII NAK) deletes the entire line, as delimited by a newline or **EOF** character.

EOF (**<ctrl-D>** or ASCII EOT) generates an end-of-file character from a terminal. When received, all the characters waiting to be read are immediately passed to the program without waiting for a newline, and the **EOF** is discarded. Thus, if no characters are waiting, which is to say the **EOF** occurred at the beginning of a line. zero characters will be passed back, which is the standard end-of-file indication.

NL (ASCII LF) is the normal line delimiter. It cannot be changed or escaped.

STOP (**<ctrl-S>** or ASCII DC3) can be used to suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read. While output is suspended, STOP characters are ignored and not read.

START (**<ctrl-Q>** or ASCII DC1) resumes output that has been suspended by a STOP character. While output is not suspended, START characters are ignored and not read. The start/stop characters can be changed via command **stty**, or via special **ioctl()** calls described below.

The character values for INTR, QUIT, ERASE, **EOF**, and KILL may be changed to suit individual tastes. The ERASE, KILL, and **EOF** character may be escaped by a preceding \ character, in which case the system ignores its special meaning. See the Lexicon article on **stty** for information on how to change these settings dynamically.

When using a "modem control" serial line, loss of carrier from the data-set (modem) causes a *hangup* signal to be sent to all processes that have this terminal as the controlling terminal. Unless other arrangements have been made, this signal causes the process to terminate. If the hangup signal is ignored, any subsequent read returns with an end-of-file indication. Thus programs that read a terminal and test for end-of-file can terminate appropriately when hung up on.

When one or more characters are written, they are transmitted to the terminal as soon as previously written characters have finished typing. Input characters are echoed by putting them into the output queue as they arrive. If a process produces characters more rapidly than they can be printed, it will be suspended when its output queue exceeds some limit, known as the "high water mark". When the queue has "drained" down to some threshold, the program resumes.

The header file **<sgtty.h>** declares structures and manifest constants to control the **sgtty** interface. Of interest to users are the constants that define baud rates for terminal ports; these are as follows.

B50	50 baud
B75	75 baud
B110	110 baud
B134	134 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19,200 baud
B38400	38,400 baud

Terminal ioctl() Functions

Header file **<sgtty.h>** defines the following data structures used by the various device drivers to convey terminal specific information. These structures are used in conjunction with special terminal or device driver symbolic constants as part of **ioctl()** requests.

The **sgttyb** structure contains information related to line discipline, such as serial line speed, if appropriate, the "erase" and "kill" characters, and a series of flags which set the mode of the line.

```
/*
 * Structure for TIOCSETP/TIOCGETP
 */
struct sgttyb {
        char    sg_ispeed;      /* Input speed */
        char    sg_ospeed;      /* Output speed */
        char    sg_erase;       /* Character erase */
        char    sg_kill;        /* Line kill character */
        int     sg_flags;       /* Flags */
};
```

The following symbolic constants are used to access bit positions of member **sg_flags** in data structure **sgttyb**:

CBREAK	Each input character causes wakeup (i.e., forces a return from a **read()** system call).
CRMOD	Map the carriage return characters '\r' to the newline character '\n'.
CRT	Use CRT-style character erase.
ECHO	Echo input characters.
EVENP	Select even parity. If used in conjunction with **ODDP**, allow either parity.
LCASE	Lowercase mapping on input.
ODDP	Select odd parity. If used in conjunction with **EVENP**, allow either parity.
RAW	Raw mode. Same as **RAWIN** plus **RAWOUT**.
RAWIN	Input is treated as 8-bit characters and not interpreted.
RAWOUT	Output is treated as 8-bit characters and not interpreted.

LEXICON

TANDEM Use X-ON/X-OFF flow control protocol to remote device.

XTABS Expand tabs to spaces.

Data structure **tchars** specifies additional special terminal characters such as the "interrupt" and "quit" characters, the "start" and "stop" characters used for flow control, and the "end-of-file" character.

```
/*
 * Structure for TIOCSETC/TIOCGETC
 */
struct tchars {
        char    t_intrc;        /* Interrupt */
        char    t_quitc;        /* Quit */
        char    t_startc;       /* Start output */
        char    t_stopc;        /* Stop output */
        char    t_eofc;         /* End of file */
        char    t_brkc;         /* Input delimiter */
};
```

The following symbolic constants are used to access various device functions via **ioctl()** calls, as defined in header file **<sgtty.h>**. Note that not all functions are appropriate for all classes of devices.

TIOCCBRK Clear a BREAK condition on a serial line (i.e., "mark" the line). This request cancels a previously issued **TIOCSBRK** request.

TIOCCDTR Clear modem control signal Data Terminal Ready (DTR) on a serial line.

TIOCCHPCL Do not force a hangup on "last close" on a modem line. The normal mode of operation for serial lines is to drop modem signal Data Terminal Ready (DTR) when the last **close()** operation is performed, thus requesting the attached modem to drop the connection.

TIOCCRTS Clear the Request To Send (RTS) signal on a serial line. Modem control signal RTS is often used for hardware flow control.

TIOCEXCL Set device access as exclusive use. This request requires the process to have **root** privileges.

TIOCFLUSH Flush the input queue, discarding any pending input characters, and wait for the output queue to "drain".

TIOCGETC Get current values of the special terminal characters, as defined by data structure **tchars**.

TIOCGETF Get current console keyboard function key bindings. This request is specific to the **nkb** console keyboard device driver. See Lexicon article **nkb** for further details.

TIOCGETKBT Get current console keyboard key mapping table. This request is specific to the **nkb** console keyboard device driver. See Lexicon article **nkb** for further details.

TIOCGETP Get current terminal line settings, as defined by data structure **sgttyb**.

TIOCGETTF Get current value of the terminal flags, as defined by field **t_flags** in the TTY structure.

TIOCHPCL Set hangup on "last close". See **TIOCCHPCL** for further details.

TIOCRMSR Get the current value of the Modem Status Register (MSR) for the specified serial line. This request is device driver specific and is currently supported only in the **al** device driver. Symbolic constants **MSRCTS**, **MSRDSR**, **MSRRI**, and **MSRRLSD** correspond to the Clear To Send, Data Set Ready, Ring Indicator and Receive Line Status Detect (i.e. Carrier Detect) signals, respectively, in the MSR.

TIOCNXCL Set this device or port as non-exclusive use. See **TIOCEXCL** for further details.

TIOCQUERY Query the number of characters currently waiting in the input queue.

TIOCSBRK Assert BREAK (i.e., "space the line") on the given serial port. This is often used during login to signal a remote system to "hunt" to the next baud rate in a sequence. See **TIOCCBRK** for further details.

TIOCSDTR Assert modem control signal Data Terminal Ready (DTR) on a serial line.

TIOCSETC Wait for output to "drain", then set the terminal control characters for this device, as specified by data structure **tchars**.

TIOCSETF Set console keyboard function key mapping. This request is specific to the **nkb** console keyboard device driver. See Lexicon article **nkb** for further details.

TIOCSETKBT Set console keyboard key mapping table. This request is specific to the **nkb** console keyboard device driver. See Lexicon article **nkb** for further details.

TIOCSETN Set terminal line settings, as defined by data structure **sgttyb**. Do not flush the input queue prior to using the new settings.

TIOCSETP Same as request **TIOCSETN**, but also flush the input queue.

TIOCSRTS Assert the Request To Send (RTS) signal on a serial line. Modem control signal RTS is often used for hardware flow control.

Examples

The following code fragment gets the current terminal settings and turns off echo.

```
#include <sgtty.h>
static struct sgttyb new, orig;
. . .
/*
 * Get the existing terminal parameters for the terminal
 * device associated with file descriptor 0 (stdin),
 * turn off echo, turn on CBREAK (break on every input character)
 * and set the new parameters.
 */
ioctl(0, TIOCGETP, &orig);
new = orig;
new.sg_flags &= ~ECHO;           /* Turn off echo */
new.sg_flags |= CBREAK;          /* Turn on CBREAK mode */
ioctl(0, TIOCSETN, &new);
```

The following line uses the previously saved terminal mode to return the terminal mode to its prior state:

```
ioctl(0, TIOCSETN, &orig);
```

See Also

device drivers, gtty(), ioctl(), sgtty.h, stty, stty(), terminal, termio

sgtty.h — Header File

Definitions used to control terminal I/O
#include <sgtty.h>

sgtty.h defines structures, constants, and macros used by routines that use the **sgtty** method to control terminal I/O.

See Also

header files, sgtty

sh — Command

The Bourne shell
sh [-ceiknstuvx] *token ...*

The COHERENT system offers two command interpreters: **ksh**, the Korn shell; and **sh**, the Bourne shell. **sh** is the default COHERENT command interpreter. The tutorial included in this manual describes the Bourne shell in detail.

As you will see from the following description, a shell is both a command interpreter and a programming language in its own right. It would be worth your while to spend some time in learning the rudiment's of the shell's programming language; doing so will help you to use your COHERENT system to best advantage.

Commands

A *command* consists of one or more tokens. A *token* is a string of text characters (i.e., one or more alphabetic characters, punctuation marks, and numerals) delineated by spaces, tabs, or newlines.

LEXICON

A *simple command* consists of the command's name, followed by zero or more tokens that represent arguments to the command, names of files, or shell operators. A *complex command* uses shell constructs to execute one or more commands conditionally. In effect, a complex command is a mini-program that you write in the shell's programming language and that **sh** interprets.

Shell Operators

sh recognizes a number of operators that form pipes, that redirect input and output to commands, and that let you define the conditions under which a given command are executed.

command **|** *command*
The *pipe* operator: let the output of one command serve as the input to a second. You can combine commands with '|' to form *pipelines*. A pipeline passes the standard output of the first (leftmost) command to the standard input of the second command. For example, in the pipeline

```
sort customers | uniq | more
```

sh invokes **sort** to sort the contents of file **customers**. It then pipes the output of **sort** into the command **uniq**, which outputs one unique copy of the text that is input into it. **sh** then pipes the output of **uniq** to the command **more**, which displays it on your terminal one screenful at a time. Note that under COHERENT, unlike MS-DOS, pipes are executed concurrently: that is, **sort** does not have to finish its work before **uniq** and **more** can begin to receive input and go to work.

command **;** *command*
Execute commands on a command line sequentially. The command to the left of the ';' executes to completion; then the command to the right of it executes. For example, in the command line

```
a | b ; c | d
```

first executes the pipeline **a | b** then, when **a** and **b** are finished, executes the pipeline **c | d**.

command **&**
Execute a command in the background. This operator must follow the command, not precede it. It prints the process identifier of the command on the standard output, so you can use the **kill** command to kill that process should something go wrong. This operator lets you execute more than one command simultaneously. For example, the command

```
fdformat -v /dev/fha0 &
```

formats a high-density, 5.25-inch floppy disk in drive 0 (that is, drive A); but while the disk is being formatted, **sh** returns the command line prompt so you can immediately enter another command and begin to work. If you did not use the '&' in this command, you would have to wait until formatting was finished before you could enter another command.

command **&&** *command*
Execute a command upon success. **sh** executes the command that follows the token '&&' only if the command that precedes it returns a zero exit status, which signifies success. For example, the command

```
cd /etc
fdformat -v /dev/fha0 && badscan -o proto /dev/fha0 2400
```

formats a floppy disk, as described above. If the format was successful, it then invokes the command **badscan** to scan the disk for bad blocks; if it was not successful, however, it does nothing.

command **||** *command*
Execute a command upon failure. This is identical to operator '&&', except that the second command is executed if the first returns a non-zero status, which signifies failure. For example, the command

```
/etc/fdformat -v /dev/fha0 || echo "Format failed!"
```

formats a floppy disk. If formatting failed, it echoes the message **Format failed!** on your terminal; however, if formatting succeeds, it does nothing.

Note that the tokens newline, ';' and '&' bind less tightly than '&&' and '||'. **sh** parses command lines from left to right if separators bind equally.

>*file* Redirect into *file* all text written to the standard output, which normally is written onto your screen. For example, the command

```
sort customers >customers.sort
```

sorts file **customers** and writes the sorted output into file **customers.sort**. **sh** creates **customers.sort** if it does not exist, and destroys its previous contents if it does exist.

>>*file* Append onto *file* all text written to the standard output, which normally is written onto your screen. If *file* does not exist, **sh** creates it; however, if the file already exists, **sh** appends the output onto its contents rather than destroying them. For example, the command

```
sort customers.now | uniq >>customers.all
```

sorts file **customers.now**, pipes its output to command **uniq**, which throws away duplicate lines of input, and appends the results onto file **customers.all**.

<*file* Redirect input. Here, **sh** reads the contents of a file and processes them as if you had typed them from your keyboard. For example, the command

```
ed textfile <edit.script
```

invokes the line editor **ed** to edit **textfile**; however, instead of reading editing commands from your keyboard, **sh** passes to **ed** the contents of file **edit.script**. This command would let you prepare an editing script that you could execute repeatedly upon files rather than having to type the same commands over and over.

<< *token*
Prepare a "here document". This operator tells **sh** to accept standard input from the shell input until it reads the next line that contains only *token*. For example, the command

```
cat >FOO <<\!
      Here is some text.
!
```

redirects all text between '<<\!' and '!' to the **cat** command. The operator '>' in turn redirects the output of **cat** into file **FOO**. **sh** performs parameter substitution on the here document unless the leading *token* is quoted; parameter substitution and quoting are described below.

command **2>** *file*
Redirect into *file* all text written to the standard error, which normally is written onto your screen. For example, the command

```
nroff -ms textfile >textfile.p 2>textfile.err
```

invokes the command **nroff** to format the contents of **textfile**. It redirects the output of **nroff** (i.e., the standard output) into **textfile.p**; it also redirects any error messages that **nroff** may generate into file **textfile.err**.

Please note that a command may use up to 20 streams. By default, stream 0 is the standard input; stream 1 is the standard output; and stream 2 is the standard error. **sh** lets you redirect any of these streams individually into a file, or combine streams into each other.

<&*n* **sh** can redirect the standard input and output to duplicate other file descriptors. (See the Lexicon article **file descriptor** for details on what these are.) This operator duplicates the standard input from file descriptor *n*.

>&*n* Merge one output stream with another. For example,

```
2>&1
```

merges the output of file descriptor 2 (the standard error) with that file descriptor 1 (the standard output).

<&- Close the standard input.

>&- Close the standard output.

When you execute a command in the foreground, that command inherits the file descriptors and signal traps (described below) of the invoking shell, modified by any specified redirection. When you execute a command in the background, it receives its input from the null device **/dev/null** (unless you redirect its input and output), and ignores all interrupt and quit signals.

LEXICON

File-Name Patterns

If a token contains any of the characters '?', '*', or '[', **sh** interprets it as being a file-name *pattern*. **sh** "expands" a pattern into the names of zero or more files in the current directory. These characters are sometimes called "wildcards," because each can represent any of several values, depending upon how you use them:

? Match any single character except newline. For example, the command

```
ls name?
```

prints the name of any file that consists of the string **name** plus any one character. If B name is followed by no characters, or is followed by two or more characters, it will not be printed.

* Match a string of zero or more characters, other than newline. For example, the command

```
ls name*
```

prints the name of any file that begins with the string **name**, followed by zero or more other characters. Likewise, the command

```
ls name?*
```

prints the name of any file that consists of the string **name** followed by at least one character. Unlike **name***, the token **name?*** insists that be followed by at least one character before it will be printed.

[!*xyz***]**
Exclude characters *xyz* from the string search. For example, the command

```
ls [!abc]*
```

prints all files in the current directory except those that begin with **a**, **b**, or **c**.

[*C-c***]**
Enclose alternatives to match a single character. A hyphen '-' indicates a range of characters. For example, the command

```
ls name[ABC]
```

prints the names of files **nameA**, **nameB**, and **nameC** (assuming, of course, that those files exist in the current directory). The command

```
ls name[A-K]
```

prints the names of files **nameA** through **nameK** (again, assuming that they exist in the current directory).

When **sh** reads a token that contains one of the above characters, it replaces the token in the command line with an alphabetized list of file names that match the pattern. If it finds no matches, it passes the token unchanged to the command. For example, when you enter the command

```
ls name[ABC]
```

sh replaces the token **name[ABC]** with **nameA**, **nameB**, and **nameC** (again, if they exist in the current directory), so the command now reads:

```
ls nameA nameB nameC
```

It then passes this second, transformed version of the command line to the command **ls**.

Note that the slash '/' and leading period '.' must be matched explicitly in a pattern. The slash, of course, separates the elements of a path name; whereas a period at the begin of a file name usually (but not always) indicates that that file has special significance.

Pattern Matching in Prefixes and Suffices

sh recognizes special constructs that let you match patterns in the prefices and suffices of a string:

{#*parameter***}**
This operator gives the number of characters in *parameter*. For example, the command

```
foo=BAZZ ; echo ${#foo}
```

prints '4' on your screen, which is the number of characters that comprise variable **foo**.

{*string1*%*string2*}
This returns the longest string for which the beginning of *string1* matches *string2*. For example, if variable **xyzzy** is initialized to string **usr/bin/cpio**, then the command

```
echo ${xyzzy%/*}
```

echoes the string **usr/bin**.

{*string1*%%*string2*}
This returns the shortest string for which the beginning of *string1* matches *string2*. For example, if variable **xyzzy** is initialized to **usr/bin/cpio**, then the command

```
echo ${xyzzy%/*}
```

echoes the string **usr**.

{*string1*#*string2*}
This returns the longest string for which the end of *string1* matches *string2*. For example, if variable **plugh** is initialized to the string **usr/bin/cpio ,** the command

```
echo ${plugh#*/}
```

echoes **bin/cpio**.

{*string1*##*string2*}
This returns the shortest string for which the end of *string1* matches *string2*. For example, given that **plugh=usr/bin/cpio**, the command

```
echo ${plugh##*/}
```

echoes **cpio**.

The following shows how to use these expressions to implement the command **basename**:

```
basename () {
      set $(echo ${1##*/}) $2
      echo ${1%$2}
}
```

Quoting Text

From time to time, you will want to "turn off" the special meaning of characters. For example, you may wish to pass a token that contains a literal asterisk to a command; to do so, you need a way to tell **sh** not to expand the token into a list of file names. Therefore, **sh** recognizes the **quotation operators** '\', '"', and '"'. These "turn off" (or *quote*) the special meaning of operators.

The backslash '\' quotes the following character. For example, the command

```
ls name\*
```

lists a file named **name***, and no other.

The shell ignores a backslash immediately followed by a newline, called a *concealed newline*. This lets you give more arguments to a command than will fit on one line. For example, the command

```
cc -o output file1.c file2.c file3.c \
      file4.c file5.c file19.c
```

invokes the C compiler **cc** to compile a set of C source files, the names of which extend over more than one line of input. You will find this to be extremely helpful, especially when you write scripts and **makefile**s, to help you write neat, easily read commands.

A pair of apostrophes ' ' prevents interpretation of any enclosed special characters. For example, the command

```
find . -name '*.c' -print
```

finds and prints the name of any C-source file in the current directory and any subdirectory. The command **find** interprets the '*' internally; therefore, you want to suppress the shell's expansion of that operator, which is accomplished by enclosing that token between apostrophes.

A pair of quotation marks " " has the same effect. Unlike apostrophes, however, **sh** performs parameter substitution and command-output substitution (described below) between quotation marks.

LEXICON

Environmental Variables

Environmental variables are names that can be assigned string values on a command line, in the form

name=value

name must begin with a letter, and can contain letters, digits, and the underscore character '_'. In shell input, '$*name*' or '${*name*}' represents the value of the variable. Consider, for example, the commands:

```
TEXT=mytext
nroff -ms $TEXT >$TEXT.out
```

Here, **sh** expands **$TEXT** before it executes the command **fBnroff**. This technique is very useful in large, complex scripts: by using variables, you can change the behavior of the script by editing one line, rather than having to edit numerous variables throughout the script.

Note that if an assignment precedes a command on the same command line, the effect of the assignment is local to that command; otherwise, the effect is permanent. For example,

```
kp=one testproc
```

assigns variable **kp** the value **one** only for the execution of the script **testproc**.

sh sets the following environmental variables by default:

The number of actual positional parameters given to the current command.

@ The list of positional parameters "$1 $2 ...".

***** The list of positional parameters "$1" "$2" ... (the same as '$@' unless quoted).

- Options set in the invocation of the shell or by the **set** command.

? The exit status returned by the last command.

! The process number of the last command invoked with '&'.

$ The process number of the current shell.

sh also references the following variables:

CWD Current working directory: this is the name of the directory in which you are now working. Note that this variable is not common to other implementations of **sh**. Code that uses it may not be portable to other operating systems.

HOME Initial working directory; usually specified in the password file **/etc/passwd**.

IFS Delimiters for tokens; usually space, tab and newline.

LASTERROR Name of last command returning nonzero exit status.

MAIL Checked at the end of each command. If file specified in this variable is new since last command, the shell prints "You have mail." on the user's terminal.

PATH Colon-separated list of directories searched for commands.

PS1 First prompt string. By default, this is '$'.

PS2 Second prompt string. By default, this is '>'. **sh** prints it when it expects more input, such as when an open quotation-mark has been typed but a close quotation-mark has not been typed, or within a shell construct.

Beginning with release 4.2, the COHERENT implementation of **sh** performs word-expansion on the values of the variables **PS1** and **PS2**. For example, setting the variables

```
export SITE=$(uname -s)
PS1="$SITE $USER $(pwd) > "
```

create a prompt that consists of your site name, your login identifier, and your current working directory.

The special forms '${*nameCtoken*}' perform conditional parameter substition: *C* is one of the characters '-', '=', '+', or '?'. **sh** replaces the form '${*name-token*}' by the value of *name* if it is set, and by *token* otherwise. It handles the '=' form in the same way, but also sets the value of *name* to *token* if it was not set previously. **sh** replaces the '+' form

by *token* if the given *name* is set. It replaces the '?' form by the value of *name* if set, and otherwise prints *token* and exits from the shell.

To unset an environmental variable, use the command **unset**. The command **unset -f** undefines a shell function (described below).

Command Output Substitution

sh can use the output of a command as shell input (e.g., as command arguments) by enclosing the command between grave characters ` `. For example, to list the contents of the directories named in file **dirs**, use the command

```
ls -l `cat dirs`
```

Constructs

sh lets you control execution of commands by the constructs **break**, **case**, **continue**, **for**, **if**, **until**, and **while**. It recognizes each reserved word only if it occurs unquoted as the first token of a command. This implies that a separator must precede each reserved word in the following constructs; for example, newline or ';' must precede **do** in the **for** construct.

The following describes each shell construct:

break *[n]*
Exit from a **for**, **until**, or **while** loop. If *n* is given, exit from the preceding *n* levels of looping.

case *token* **in [** *pattern [| pattern] ...***)** *sequence***;;] ... esac**
Check *token* against each *pattern*, and execute *sequence* associated with the first matching *pattern*.

continue *[n]*
Branch to the end of the *n*th enclosing **for**, **until**, or **while** construct.

for *name [* **in** *token ...]* **do** *sequence* **done**
Execute *sequence* once for each *token*. On each iteration, *name* takes the value of the next *token*. If the **in** clause is omitted, **$@** is assumed. For example, to list all files ending with **.c:**

```
for i in *.c
do
      cat $i
done
```

if *seq1* **then** *seq2 [* **elif** *seq3* **then** *seq4] ... [* **else** *seq5]* **fi**
Execute *seq1*. If the exit status is zero, execute *seq2*; if not, execute the optional *seq3* if given. If the exit status of *seq3* is zero, then execute *seq4*, and so on. If the exit status of all tested sequences is nonzero, execute *seq5*.

until *sequence1 [* **do** *sequence2]* **done**
Execute *sequence2* until the execution of *sequence1* results in an exit status of zero.

while *sequence1 [* **do** *sequence2]* **done**
Execute *sequence2* as long as the execution of *sequence1* results in an exit status of zero.

(*sequence***)**
Execute *sequence* within a subshell. This allows *sequence* to change the current directory, for example, and not affect the enclosing environment.

$(( )) Perform arithmetic expansion, as described in the POSIX Standard. The expression syntax is that of C, but the only values are signed integers, and there are no side effects (i.e., no increment, decrement, or assignment operators). The expression given inside this form is first processed for further expansions, then evaluated according to the C rules for arithmetic; the result is placed on the output command line. This is most useful when used with **return** and **exit** to form return values from functions and scripts.

To use **$(())** with the shell's logical operators and statements, you must use some indirection. For example:

```
val () {
      return $((! ($*)))
}
```

Or:

LEXICON

```
val $(($# < 5)) && {
      echo Not enough arguments
      exit 1
}
```

Or:

```
val $((${#foo} > 8)) {
      echo $foo is too long; use 8 characters, maximum.
      exit 2
}
```

{*sequence*
} Braces simply enclose a *sequence*. Note that the closing '}' must appear on the line that follows *sequence*.

Special Commands

sh usually executes commands with the **fork** system call, which creates another process. However, **sh** executes the commands in this section either directly or with an **exec** system call. See the Lexicon articles on **fork()** and **exec** for details on these calls.

. *script* Read and execute commands from *script*. Positional parameters are not allowed. **sh** searches the directories named in the environmental variable **PATH** to find the given *script*.

: *[token ...]*
A colon ':' indicates a "partial comment". **sh** normally ignores all commands on a line that begins with a colon, except for redirection and such symbols as **$**, **{**, **?**, etc.

A complete comment: if **#** is the first character on a line, **sh** ignores all text that follows on that line.

cd *dir* Change the working directory to *dir*. If no argument is given, change to the home directory.

command *command* **[***arguments***]**
When you issue a command, **sh** by default looks for that command among its set of built-in functions; if it cannot find it there, it looks for the command in the directories set in the environmental variable **PATH**. Thus, if you have given a shell function the same name as an executable command, **sh** will never find the executable.

The command **command** tells **sh** to look for *command* in the directories named in your **PATH**, and ignore any shell function with that name.

dirs **sh** lets you maintain a "directory stack", or stack of names of directories. You can push, pop, and otherwise manipulate the contents of this stack, which you can use for any purpose for which you need to access a number of directory names quickly. The command **dirs** prints the contents of the directory stack. The commands **pushd** and **popd** also manipulate the directory stack.

Please note that these commands are unique to the COHERENT implementation of **sh**, and are not portable to other shells. *Caveat utilitor.*

eval *[token ...]*
Evaluate each *token* and treat the result as shell input.

exec *[command]*
Execute *command* directly rather than performing **fork**. This terminates the current shell.

exit *[status]*
Set the exit status to *status*, if given; otherwise, the previous status is unchanged. If the shell is not interactive, terminate it.

export *[name ...]*
sh executes each command in an *environment*, which is a set of shell-variable names and their corresponding values. When you invoke **sh**, it inherits all environmental variables that are currently set; and it, in turn, normally passes those variables to each command it invokes. **export** specifies that the shell should pass the modified value of each given *name* to the environment of subsequent commands. When no *name* is given, **sh** prints the name and value of each variable marked for export.

getopts *optstring name [arg ...]*
Parse the *arg*s to *command*. See the Lexicon entry for **getopts** for details.

popd *[N ...]*
Pop the directory stack. When used without an argument, it pops the stack once. When used with one or more numeric arguments, **popd** pops the specified items from the stack; item 0 is the top of the stack. (For information on the directory stack, see the entry for the command **dirs**, above.)

pushd *[dir0 ... dirN]*
Push *dir0* through *dirN* onto the directory stack, and change the current directory to the last directory pushed onto the stack. When called without an argument, **pushd** exchanges the two top stack elements. (For information on the directory stack, see the entry for the command **dirs**, above.)

read *name ...*
Read a line from the standard input and assign each token of the input to the corresponding shell variable *name*. If the input contains fewer tokens than the *name* list, assign the null string to extra variables. If the input contains more tokens, assign the last *name* the remainder of the input.

readonly *[name ...]*
Mark each shell variable *name* as a read only variable. Subsequent assignments to read only variables will not be permitted. With no arguments, print the name and value of each read only variable.

set [-ceiknstuvx *[name ...]]*
Set listed flag. If *name* list is provided, set shell variables *name* to values of positional parameters beginning with **$1**.

shift Reset positional parameter **1** to the value **$2**, reset positional parameter **2** to the value **$3**, and so on. The original value of positional parameter **1** is thrown away.

times Print the total user and system times for all executed processes.

trap *[command] [n ...]*
Execute *command* if **sh** receives signal *n*. If *command* is omitted, reset traps to original values. To ignore a signal, pass null string as *command*. With *n* zero, execute *command* when the shell exits. With no arguments, print a description of the traps that have already been set.

umask *[nnn]*
Set user file creation mask to *nnn*. If no argument is given, print the current file creation mask.

wait *[pid]*
Delay execution of further commands until the process that has process identifier *pid* terminates. If *pid* is omitted, delay until every child process has finished executing. If no child process is active, this command finishes immediately.

Shell Functions

Beginning with COHERENT release 4.2, **sh** lets you declare and use functions. In effect, a function is a script that you load permanently into memory.

A function takes the form

```
function() {
      command $1 $2
      other_function $3 $4
}
```

A function begins with its name. A pair of parentheses after the name tell **sh** that this is a function.

The body of the function is enclosed within braces. A function can call any command, plus any other function. Arguments are passed into the function using the syntax **$1**, **$2**, etc., just as with a shell script.

sh keeps an internal list of the functions that you have declared. When it reads an identifier, it first searches its list of functions; if the identifier is not a function, **sh** then assumes that the identifier names a command, and it attempts to find that command in the directories you have named in your environmental variable **PATH**. Thus, if you give a function the same name as that of an existing command, **sh** will always use the function and never find the command. To suppress this behavior, use the command **command**, described above.

The following example function copies one file into another:

```
copyfile(){
      if [ $# -eq 1 ]; then
            cat $1
      else
            cp $1 $2
      fi
}
```

Shell Library

The file **/usr/lib/shell_lib.sh** holds a library of shell functions. You can import these library with the '.' command.

This library holds the following functions:

basename "*pathname*" **[** "*suffix*" "*prefix*" **]**
: This function behaves the same as the command **basename**, except that you can include an option *prefix* to strip as well as a *suffix*.

file_exists "*filename*"
: Return **true** if file *filename* exists.

find_file "*filename*" "*path*" "**path**" **...**
: Seek *file* in every directory named in a *path*.

has_prefix "*string*" "*prefix*"
: Return **true** if *string* is prefixed with the string *prefix*.

is_empty "*arg*"
: Return true if *arg* is null.

is_equal "*arg1*" "*arg2*"
: Return true if *arg1* and *arg2* are equal.

is_numeric "*argument*"
: Return **true** if *argument* is numeric, or **false** if it is not.

is_yes "*arg*"
: Return true if *arg* matches 'Y', 'y', "YES", or "yes"; one if the argument matches 'N', 'n', "NO", or "no"; two if it matches anything else.

parent_of "*file*" **[** "*prefix*" "*suffix*" **]**
: By default, write the path name of *file*. *prefix* and *suffix* are the prefix and suffix of a command run with the output path name.

read_input "*prompt*" "*variable*" "*default*" "*validate*"
: Echo *prompt* onto the screen. Write what the user types into *variable*. If the user does not respond, set *variable* to *default*. The optional argument *validate* names a function that **read_input** calls to evaluate what the user types; often, this is the shell-library function **require_yes_or_no**.

require_yes_or_no "*arg*"
: This is the validation function for **read_input**. Check whether *arg* is properly affirmative or negative.

source_path "*script*" **[** "*command* " **]**
: Echo the name of the directory that contains *script*. Normally, this function is called with the **$0** of *script*. It pipes its output into *command* if this argument is set; if it is not, it writes to the standard output.

split_path "*string*" "*prefix*" "*suffix*"
: This function dissects *string*, which must consist of colon-separated elements, such as a **PATH** string. *prefix* and *suffix* give, respectively, the prefix and suffix of the command that is run for every component of *string*.

val "*expression*"
: Return the negated value of *expression*. You can use this function to turn shell arithmetic expressions into test results.

Scripts

Shell commands can be stored in a file, or *script.* The command

```
sh script [ parameter ... ]
```

executes the commands in *script* with a new subshell **sh**. Each *parameter* is a value for a positional parameter, as described below. If you have used the command **chmod** to make *script* executable, you may omit the **sh** command.

To ensure that a script is executed by **sh**, begin the script with the line:

```
#!/bin/sh
```

Parameters of the form '$*n*' represent command-line arguments within a script. *n* can range from zero through nine; **$0** always gives the name of the script. These parameters are also called *positional parameters*.

If no corresponding parameter is given on the command line, the shell substitutes the null string for that parameter. For example, if the script **format** contains the following line:

```
nroff -ms $1 >$1.out
```

then invoking **format** with the command

```
format mytext
```

invokes the command **nroff** to format the contents of **mytext**, and writes the output into file **mytext.out**. If, however, you invoke this command with the command line

```
format mytext yourtext
```

the script formats **mytext** but ignores **yourtext** altogether.

Reference **$*** represents all command-line arguments. If, for example, we change the contents of script **format** to read

```
nroff -ms $* >$1.out
```

then the command

```
format mytext yourtext
```

invoke **nroff** to format the contents of **mytext** and **yourtext**, and write the output into file **mytext.out**.

Commands in a script can also be executed with the . (dot) command. It resembles the **sh** command, but the current shell executes the script commands without creating a new subshell or a new environment; therefore, you cannot use command-line arguments.

Command-line Options

-c *string*
: Read shell commands from *string*.

-e Exit on any error (command not found or command returning nonzero status) if the shell is not interactive.

-i The shell is interactive, even if the terminal is not attached to it; print prompt strings. For a shell reading a script, ignore the signals **SIGTERM** and **SIGINT**.

-k Place all keyword arguments into the environment. Normally, **sh** places only assignments to variables preceding the command into the environment.

-n Read commands but do not execute them.

-s Read commands from the standard input and write shell output to the standard error.

-t Read and execute one command rather than the entire file.

-u If the actual value of a shell variable is blank, report an error rather than substituting the null string.

-v Print each line as it is read.

-x Print each command and its arguments as it is executed.

- Cancel the **-x** and **-v** options.

If the first character of argument 0 is '-', **sh** reads and executes the scripts **/etc/profile** and **$HOME/.profile** before reading the standard input. **/etc/profile** is a convenient place for initializing system-wide variables, such as **TIMEZONE**.

LEXICON

Files

/etc/profile — System-wide initial commands
$HOME/.profile — User-specific initial commands
/dev/null — For background input
/tmp/sh* — Temporary files
/usr/lib/shell_lib.sh — Library of shell functions

See Also

commands, dup(), environ, exec, fork(), getopts, ksh, login, newgrp, set, signal(), test, Using COHERENT, vsh
Introduction to sh, the Bourne Shell, tutorial

For a list of all commands associated with **sh**, see the section **Shell Commands** in the **commands** Lexicon article.

Diagnostics

sh notes on the standard error all syntax errors in commands, and all commands that it cannot find. Syntax errors cause a noninteractive shell to exit. It gives error messages if I/O redirection is incorrect. **sh** returns the exit status of the last command executed or the status specified by an **exit** command.

shadow — System Administration

File that holds restricted passwords
/etc/shadow

COHERENT stores information in file **/etc/passwd**. This file identifies each user, gives her home directory, default shell, and base group. It must be universally readable so that it can be used by programs like **ls**, which must translate user-identification numbers into login identifiers.

In general, this system works well; however, it does create a hole in system security. If users' encrypted passwords are kept in **/etc/passwd**, which is universally readable, a "cracker" can read the passwords, decypher some of them with brute-force methods, and then log in as the users whose passwords she cracked.

To plug that hole in system security, UNIX implemented the method of "shadow" passwords. In this scheme, a user's login information is still kept in **/etc/passwd**; however, the encrypted passwords (plus supplemental information) is kept in the file **/etc/shadow**, which can be read only by a process with root-level permissions.

The shadow password file contains one entry per user. Each user identified in **/etc/shadow** must have an entry in **/etc/passwd**. The opposite is not true, but a user described in **/etc/passwd** who does not have an entry in **/etc/shadow** cannot log into your system. Each entry in **/etc/shadow** is laid out exactly the same as file **/etc/passwd.** At present, the COHERENT implementation of **login** uses only the *name* and *password* fields. For details, see the Lexicon entry for **passwd**.

Reading /etc/shadow

COHERENT includes four functions with which a program can read the shadow-password file **/etc/shadow**:

endspent()
Close **/etc/shadow** after reading from it.

getspent()
Read the next record from **/etc/shadow**. If a process has not yet read **/etc/shadow**, it returns the first record.

getspnam()
Return the first record for the user with a given login identifier.

setspent()
Return the seek pointer for **/etc/shadow** to the beginning of the file.

Functions **getspent()** and **getspnam()** return a pointer to an object with structure **spwd**, which gives an analogue for each field in **/etc/shadow**. This structure is defined in header file **<shadow.h>**. For details on this structure, see the Lexicon entry for **shadow.h**.

See Also

Administering COHERENT, login, shadow.h

Notes

For details of how the program **login** uses shadow passwords, see its entry in the Lexicon.

shadow.h — Header File

Definitions used with shadow passwords
#include <shadow.h>

The header file **<shadow.h>** declares and defines the functions, macros, structures, and constants used to manipulate shadow passwords.

<shadow.h> holds defines the structure **spwd**, which describes the records that are stored in file **/etc/shadow**. **<shadow.h>** gives two definitions **spwd**: one implements the structure used by UNIX System V, release 4; and the other implements the structure used by UNIX System V, release 3.

The following gives the form of **spwd** that is used by some releases of UNIX System V, release 4:

```
struct spwd {
      char *sp_namp;    /* User Name */
      char *sp_pwdp;    /* Encrypted password */
      long sp_lstchg;   /* Last changed date */
      long sp_min;
      long sp_max;
      long sp_warn;
      long sp_inact;
      long sp_expire;   /* Acct expiration date. */
      unsigned long sp_flag;
};
```

The following gives the version of **spwd** used by UNIX System V, release 3:

```
struct      spwd {
      char *sp_name; /* User name */
      char *sp_passwd; /* Encrypted password - non-POSIX */
      int sp_uid; /* User id */
      int sp_gid; /* Group id */
      int sp_quota; /* File space quota  - non-POSIX*/
      char *sp_comment; /* Comments - non-POSIX */
      char *sp_gecos; /* Gecos box number - non-POSIX */
      char *sp_dir; /* Working directory */
      char *sp_shell; /* Shell */
};
```

By default, COHERENT uses the System V, release 3, version of **spwd**.

For information on how to select a given version of **spwd**, see the discussion of compilation environments in the Lexicon article **header files**.

See Also

header files, endspent(), getspent(), getspnam(), libc, setspent(), shadow

SHELL — Environmental Variable

Name the default shell
SHELL=*shell*

The environmental variable **SHELL** names the shell that COHERENT invokes when you log in. The default is **SHELL=/bin/sh**, which invokes the Bourne shell.

See Also

environmental variables, sh

shellsort() — General Function (libc)

Sort arrays in memory
void shellsort(*data***,** *n***,** *size***,** *comp***)**
char **data***data***; int** *n***,** *size***; int (******comp***)();**

shellsort() is a generalized algorithm for sorting arrays of data in memory, using D. L. Shell's sorting method. **shellsort()** works with a sequential array of memory called *data*, which is divided into *n* parts of *size* bytes each. In

practice, *data* is usually an array of pointers or structures, and *size* is the **sizeof** the pointer or structure.

Each routine compares pairs of items and exchanges them as required. The user-supplied routine to which *comp* points performs the comparison. It is called repeatedly, as follows:

```
(*comp)(p1, p2)
char *p1, *p2;
```

Here, *p1* and *p2* each point to a block of *size* bytes in the *data* array. In practice, they are usually pointers to pointers or pointers to structures. The comparison routine must return a negative, zero, or positive result, depending on whether *p1* is less than, equal to, or greater than *p2*, respectively.

Example

For an example of how to use this routine, see the entry for **string**.

See Also

libc, qsort()
The Art of Computer Programming, vol. 3, pp. 84*ff*, 114*ff*

Notes

For a discussion of how the **shellsort** algorithm differs from that used by **qsort()**, see the Lexicon entry for **qsort()**.

shift — Command

Shift positional parameters
shift

Commands to the shell can be stored in a file, or *script*. Positional parameters pass command-line variables to a script.

shift changes the values of positional parameters. The old parameter values **$2**, **$3**, ... become the new parameter values **$1**, **$2** .... **shift** also reduces the value of **$#**, which gives the number of positional parameters, by one.

The shell executes **shift** directly.

See Also

commands, ksh, sh

shm — Kernel Module

Kernel module for shared memory

The kernel module **shm** enables System V-style shared memory. It is called a *kernel module* because you can link it into your kernel or exclude it, as you wish, just like a device driver; yet it is not a true device driver because it does not perform I/O with a peripheral device.

See Also

device drivers, kernel, shmget()

shm.h — Header File

Definitions used with shared memory
#include <sys/shm.h>

shm.h defines constants and macros used by routines that implement the COHERENT shared-memory facility.

See Also

header files, shmget()

shmat() — General Function (libc)

Attach a shared-memory segment to a process
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
char ***shmat (***shmid***,** *shmaddr***,** *shmflg***)**
int *shmid***,** *shmflag***; char** **shmaddr***;*

shmat() attaches the shared-memory segment associated with the identifier *shmid* with the **.data** segment of the calling process.

shmat() selects the address at which to attach the shared-memory segment. If

shmflg & SHM_RDONLY

is true, the attached memory is read-only; otherwise, it is read-write.

shmat() fails if any of the following is true:

- *shmid* is not a valid shared-memory identifier. **shmat()** sets **errno** to **EINVAL**.
- The calling process lacks appropriate permission (**EACCES**).
- Not enough memory is available to hold the shared-memory segment (**ENOMEM**).
- The process already has the maximum number of shared-memory segments attached to it (**EMFILE**).

You can attach more than one shared-memory segment to a process, up to a maximum of six. COHERENT assigns each segment its own address.

If all went well, **shmat()** returns the address of the newly attached shared-memory segment; otherwise, it returns -1 and sets **errno** to an appropriate value.

Example

For an example of this function, see the Lexicon entry for **shmget()**.

See Also

libc, shmctl(), shmdt(), shmget()

Notes

The COHERENT implementation of shared memory does not yet support attaching a shared-memory segment to a user-defined address. Therefore, you should always set *shmaddr* to zero.

shmctl() — General Function (libc)

Manipulate shared memory
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
shmctl(*shmid***,** *command***,** *buf***)**
int *shmid*, *command*; **struct shmid_ds** **buf*;

shmctl() controls the COHERENT system's shared-memory facility. Note that shared memory consists of the segment of memory being shared, plus a copy of structure **shmid_ds**, which is defined in header file **<sys/shm.h>**. This structure describes the shared-memory segment and identifies who can manipulate it, and how.

command names the operation that you want **shmctl()** to perform, as follows:

IPC_RMID Remove the system identifier *shmid* and destroy its associated shared memory segment and **shmid_ds** structure. Only the superuser **root** or the user whose effective user ID matches the value of field **uid** can invoke this command.

IPC_SET Copy fields **shm_perm.uid**, **shm_perm.gid**, and **shm_perm.mode** (low nine bits only) from the **ipc_perm** associated with *buf* into *shmid*. Only the superuser **root** or the user who created this shared-memory segment can invoke this command.

IPC_STAT Copy every element of the **shmid_ds** associated with *shmid* into the one pointed to by *buf*.

SHM_LOCK Lock the shared-memory segment *shmid*, to keep it from being paged out of memory. Only the superuser **root** can invoke this command. Because COHERENT does not support paging, this command present does nothing.

SHM_UNLOCK Unlock the shared-memory segment *shmid*, to permit it to be paged out of memory. Only the superuser **root** can invoke this command. Because COHERENT does not support paging, this command present does nothing.

shmctl() fails if any of the following is true:

- *shmid* is not a valid shared-memory identifier. **shmget()** sets **errno** to **EINVAL**.
- *command* is not a valid command (**EINVAL**).
- *command* equals **IPC_STAT** but the owner of the calling process lacks permission (**EACCES**).
- *command* equals **IPC_RMID** or **IPC_SET** but the owner of the calling process lacks permission (**EPERM**).
- *buf* points to an illegal address (**EFAULT**).

shmctl() returns zero if all went well; otherwise, it returns -1 and sets **errno** to an appropriate value.

Example

For an example of this function, see the Lexicon entry for **shmget()**.

Files

/usr/include/sys/ipc.h
/usr/include/sys/shm.h

See Also

libc, shmat(), shmdt(), shmget()

Notes

For information on other methods of interprocess communication, see the Lexicon entries for **semctl()** and **msgctl()**.

shmdt() — General Function (libc)

Detach a shared-memory segment from a process
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
char *shmdt (*shmaddr***)**
char ****shmaddr*;**

shmdt() detaches the shared-memory segment at address *shmaddr* from the calling process. If all went well, **shmdt()** returns returns zero; otherwise, it returns -1 and sets **errno** to an appropriate value. In particular, it sets **errno** to **EINVAL** if *shmaddr* does not point to the beginning of a shared-memory segment.

Example

For an example of this function, see the Lexicon entry for **shmget()**.

See Also

libc, shmctl(), shmdt(), shmget()

Notes

The COHERENT implementation of shared memory does not yet support attaching a shared-memory segment to a user-defined address. Therefore, you should always set *shmaddr* to zero.

shmget() — General Function (libc)

Create or get shared-memory segment
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(*memkey***,** *size***,** *flag***)**
key_t *memkey***; int** *size***,** *flag***;**

shmget() creates a shared-memory identifier, associated data structure, and shared-memory segment, links them to the identifier *memkey*, and returns the shared-memory identifier that it has associated with *memkey*.

memkey is an identifier that your application generates to identify its shared-memory segments. To guarantee that each key is unique, you should use the function call **ftok()** to generate keys.

size gives the size, in bytes, of the shared-memory segment that you want **shmget()** to create.

flag can be bitwise OR'd to include the following constants:

IPC_ALLOC This process already has a shared-memory segment; please fetch it.

IPC_CREAT If this process does not already have a shared-memory segment, please create one.

IPC_EXCL Fail if a shared-memory segment already exists for this process.

IPC_NOWAIT Fail if the process must wait to obtain a shared-memory segment.

When it creates a shared-memory segment, **shmget()** also creates a copy of structure **shmid_ds**, which is defined in header file **<sys/shm.h>**, and which describes the shared-memory segment. It is defined as follows:

```
struct shmid_ds {
       struct ipc_perm                         shm_perm;/* operation permission struct */
       int shm_segsz;                          /* segment size */
       char *__unused;                         /* for binary compatibility */
       char __pad [4];                         /* for binary compatibility */
       pid_t shm_lpid;                         /* pid of last shmop */
       pid_t shm_cpid;                         /* pid of creator */
       unsigned short shm_nattch;              /* current # attached */
       unsigned short shm_cnattach;            /* for binary compatibility */
       time_t shm_atime;                       /* last shmat time */
       time_t shm_dtime;                       /* last shmdt time */
       time_t shm_ctime;                       /* last change time */
};
```

Field **shm_perm** is a structure of type **ipc_perm**, which header file **<sys/ipc.h>** defines as follows:

```
struct ipc_perm {
       unsigned short uid;                     /* owner's user id */
       unsigned short gid;                     /* owner's group id */
       unsigned short cuid;                    /* creator's user id */
       unsigned short cgid;                    /* creator's group id */
       unsigned short mode;                    /* access modes */
       unsigned short seq;                     /* slot usage sequence number */
       key_t key;                              /* key */
};
```

shmget() initializes **shm_id** as follows:

- It sets fields **shm_perm.guid**, **shm_perm.uid**, **shm_perm.cgid**, and **shm_perm.gid** to, respectively, the effective user ID and effective group ID of the calling process.
- It sets the low-order nine bits of field **shm_perm.mode** to the low-order nine bits of *flag*. These nine bits define access permissions: the top three bits give the owner's access permissions (read, write, execute), the middle three bits the owning group's access permissions, and the low three bits access permissions for others.
- It sets field **shm_segsz** equal to *size*.
- It sets fields **shm_atime**, **shm_dtime**, **shm_lpid**, and **shm_nattch** to zero, and field **shm_ctime** to the current time.

shmget() fails if any of the following is true:

- *size* is smaller than one byte, or larger than 0x10000 (the system-imposed maximum). **shmget()** sets **errno** to **EINVAL**.
- A shared-memory identifier exists for *memkey* but permission, as specified by *flag*'s low-order nine bits, is not granted (**EACCES**).
- A shared-memory identifier exists for *memkey*, but the size of its associated segment is less than *size*, and *size* does not equal zero (**EINVAL**).
- A shared-memory identifier does not exist for *memkey* and (*flag* & **IPC_CREAT**) is false (**ENOENT**).
- **shmget()** tried to create a shared-memory segment, but could not because 100 (the COHERENT-defined maximum) already exist (**ENOSPC**).

- **shmget()** tried to create a shared-memory identifier, but could not because not enough physical memory is available (**ENOMEM**).
- A shared-memory identifier already exists for *memkey*, but *flag* requests that **shmget()** create an exclusive segment it — i.e.

  ```
  ( (flag & IPC_CREAT) && (flag & IPC_EXCL) )
  ```

  is true (**EEXIST**).

If all goes well, **shmget()** returns a shared-memory identifier, which is always a non-negative integer. Otherwise, it returns -1 and sets **errno** to an appropriate value.

Example

The following demonstrates how to use COHERENT's shared-memory feature. Please note that this example will *not* work with versions of COHERENT prior to release 4.2.

The example consists of two programs: **writeshm**, which captures input from the keyboard and writes it into a shared-memory segment; and **readshm**, which reads and displays from the shared-memory segment the text that **writeshm** put there. Each program terminates when you type "end".

Note that this example is most effective if you run each program from its own virtual console.

The first program gives the source for **writeshm**:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

main()
{
      int iShmId; /* Segment id */
      char *cpShm; /* Pointer to the segment */
      key_t key; /* Segment key */

      key = ftok("/etc/passwd", 'S');     /* Get a key */

      /* if a shared-memory segment exists, get it; otherwise, create one */
      if ((iShmId = shmget(key, 256, 0644 | IPC_CREAT)) < 0) {
            perror("get");
            exit(1);
      }

      /* Attach segment to process. Use an attach address of zero to
       * let the system find a correct virtual address to attach.
       */
      if ((cpShm = shmat(iShmId, 0, 0644)) == (char *) -1) {
            perror("shmat");
            exit(1);
      }
      printf("Server is ready.\n");
      printf("Any message to continue, 'end' to exit\n");

      for (;;) {
            printf("Enter the message -> ");
            gets(cpShm);
            if (!strcmp(cpShm, "end")) {
                  puts("Bye");
                  shmdt(cpShm);     /* Detach segment */
                  break;
            }
      }
}
```

The next program gives the source for **readshm**:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

main()
{
      int iShmId; /* Segment id */
      char *cpShm; /* Pointer to the segment */
      key_t key; /* Segment key */
      char cBuf[16]; /* Read buffer */

      /* Get a key */
      key = ftok("/etc/passwd", 'S');

      /* Get shared memory id. If it does not exist, do *not* create it. */
      if ((iShmId = shmget(key, 256, 0644)) < 0) {
            perror("get");
            exit(1);
      }

      /* attach shared-memory segment to the process */
      if ((cpShm = shmat(iShmId, 0, 0644)) == (char *) -1) {
            perror("shmat");
            exit(1);
      }
      printf("Client is ready\n");

      for (;;) {
            printf("Press enter to read the message -> ");
            gets(cBuf);
            printf("Got: \"%s\"\n", cpShm);

            /* Exit on the 'end': detach and remove segment */
            if (!strcmp(cpShm, "end")) {
                  struct shmid_ds   stShmId;

                  puts("Bye");
                  shmdt(cpShm);
                  if (shmctl(iShmId, IPC_RMID, &stShmId)) {
                        perror("shmctl");
                        exit(1);
                  }
                  break;
            }
      }
}
```

Files

/usr/include/sys/ipc.h
/usr/include/sys/shm.h

See Also

ftok(), ipcrm, ipcs, libc, libsocket, shmat(), shmctl(), shmdt()

Notes

Prior to release 4.2, COHERENT implemented shared memory through the driver **shm**. In release 4.2, and subsequent releases, COHERENT implements shared memory as a set of functions that conform in large part to the UNIX System-V standard.

The kernel variables **SHMMAX** and **SHMMNI** set, respectively, the maximum size of a shared-memory segment and the number of shared-memory segments that can exist at any given time. Daredevil system operators who have large amounts of memory at their disposal may wish to change these variables to increase the system-defined limits. For details on how to do so, see the Lexicon entry **mtune**.

short — C Keyword

Data type

short is a numeric data type. The ANSI standard states that it cannot be longer than an **int**.

COHERENT defines a **short** to be two bytes long; thus, **sizeof short** equals two **char**s, or 15 bits plus a sign, and can hold any value from -32,768 to 32,767.

A **short** normally is sign extended when cast to a larger data type; however, an **unsigned short** will be zero extended when cast.

See Also

C keywords, data format, data type, int, long
ANSI Standard, §6.1.2.5

shutdown — Command

Shut down the COHERENT system
/etc/shutdown [reboot | halt | single | powerfail] *time*

shutdown shuts down the COHERENT system. You must use this command to shut COHERENT down. *Failure to shut down the system before rebooting or shutting off the computer could damage the* COHERENT *file system and destroy data.*

When you invoke **shutdown**, you must specify the "level" shutdown, and the time to shutdown. The level must be one of the following:

reboot Bring down the system, then reboot it automatically. Use this level if, for example, you are installing a new kernel.

halt Halt the system, but do not reboot it or enter single-user mode. Use this option when you intend to turn off your computer.

single Bring down the system to single-user mode.

powerfail
Bring down the system as quickly as possible. Normally, this level is invoked by a daemon that has received information of a power failure from your system's uninterruptable power supply (UPS).

time is the interval, in minutes, from the time you invoke the command to the time that **shutdown** shuts the system down. Setting *time* to zero shuts down the system immediately. Every minute, **shutdown** displays on every user's terminal the message

```
System going down in N minutes!
```

where *N* is the number of minutes left until shutdown. When time has expired, **shutdown** displays the message

```
System is going down now!
```

at which point users have ten seconds to save their files and exit. Users who have turned off system messages will not, of course, see these messages.

After the system has been halted, you do not need to type **sync**; **shutdown** does that automatically.

If users have not logged off from the system when it comes time to shut the system down, you will see the prompt:

```
Some file systems remain mounted.  Proceed with shutdown ? [y]
```

If you type 'n', the shutdown will be aborted. You should then make sure that the users have logged off, then invoke **/etc/shutdown** again. To lock out new users from logging in while you are trying to shut the system down, create the file **/etc/nologin**. Note that this file is removed automatically when you reboot your system.

If you type 'y', **shutdown** will continue as before. Users will be thrown off the system; any files they might have had opened at that time will not be updated.

See Also

commands, nologin, reboot

Notes

Only the superuser **root** can run **shutdown**.

shutdown can be run from any terminal. When the system reboots, however, control returns to the system console.

shutdown was written by Udo Munk (udo@umunk.GUN.de).

shutdown() — Sockets Function (libsocket)

Replace function to shut down system
int shutdown (*s***,** *how***)**
int *s*, *how*;

Function **shutdown()** does nothing under COHERENT. It is present in its sockets library to ensure that imported code will link.

See Also

libsocket

sigaction() — System Call (libc)

Perform detailed signal management
#include <signal.h>
int sigaction (*signal***,** *action***,** *old_action***)**
int *signal*; **const struct sigaction** **action*; **struct sigaction** **old_action*;

sigaction() lets the calling process specify the action it will take when it receives *signal*.

signal can be any of the signals named in the Lexicon entry for **signal()** except **SIGKILL** and **SIGSTOP**.

action points to a structure that specifies the action to take when *signal* is received. If *action* is set to NULL, the current disposition of the signal is unaffected.

old_action points to a structure that describes the action previously associated with *signal*, and that is to be restored upon return from **sigaction()**.

The structure **sigaction** has the following members:

```
void  (*sa_handler)();
sigset_t    sa_mask;
int   sa_flags;
```

sa_handler gives the disposition of the signal. You can set it to any of the actions described in the article for **signal()**.

sa_mask identifies the signals to be blocked while the signal handler is active. Upon entry to the signal handler, that set of signals is added to the set of signals already being blocked when *signal* was received. *signal* itself is also blocked. Note that you cannot block **SIGSTOP** and **SIGKILL**.

sa_flags specifies a set of flags used to modify the behavior of *signal*. As of this writing, **sigaction()** recognizes only the flag **SA_NOCLDSTOP**: If this is set and *signal* equals **SIGCHLD**, *signal* is not sent to the calling process when its child processes stop or continue.

sigaction() returns zero if all is well. It fails and returns -1 if either of the following is true:

- *signal* does not identify a valid signal. **sigaction()** sets **errno** to **EINVAL**.
- *action* or *old_action* points outside the process's allocated address space. **sigaction()** sets **errno** to **EFAULT**.

See Also

libc, sigaddset(), sigdelset(), sigemptyset(), sigfillset(), sigismember(), signal(), signal.h, sigpending(), sigprocmask(), sigset(), siglongjmp(), sigsetjmp(), sigsuspend()
POSIX Standard, §3.3.4

sigaddset() — Signal Function (libc)

Add a signal to a set of signals
#include <signal.h>
int sigaddset (*set***,** *signo***)**
sigset_t ******set***; int** *signo***;**

sigaddset() is one of a set of signalling functions that manipulate data objects addressable by the application, instead of a set of signals known to the system. It adds the signal *signo* to the set of signals to which *set* points.

If all goes well, **sigaddset()** returns zero. If *signo* is set to an invalid or unsupported value, it returns -1 and sets **errno** to **EINVAL**.

See Also

libc, sigaction(), sigdelset(), sigemptyset(), sigfillset(), sigismember()

Notes

If your program is compiled using the System V Release 4 compilation environment, this is a function that is linked in from **libc**. If not, a macro form is used.

sigdelset() — Signal Function (libc)

Delete a signal from a set
#include <signal.h>
int sigdelset (*set***,** *signo***)**
sigset_t ******set***;**
int *signo***;**

sigdelset() is one of a set of signalling functions that manipulate data objects addressable by the application, instead of a set of signals known to the system. It deletes the signal *signo* from the set of signals to which *set* points.

If all goes well, **sigdelset()** returns zero. If *signo* is set to an invalid or unsupported value, it returns -1 and sets **errno** to **EINVAL**.

See Also

libc, sigaction(), sigaddset(), sigemptyset(), sigfillset(), sigismember()

Notes

If your program is compiled using the System V Release 4 compilation environment, this is a function that is linked in from **libc**. If not, a macro form is used.

sigemptyset() — Signal Function (libc)

Initialize a set of signals
#include <signal.h>
int sigemptyset (*set***)**
sigset_t ******set***;**

sigemptyset() is one of a set of signalling functions that manipulate data objects addressable by the application, instead of a set of signals known to the system. It initializes the set of signals to which *set* points, such that all standard signals are excluded.

sigemptyset() returns zero if all goes well. If a problem occurs, it returns -1 and sets **errno** to an appropriate value.

An application must call either **sigemptyset()** or **sigfillset()** at least once for each object of type **sigset_t** prior to any other object. If such an object is not initialized in this way, but is supplied as an argument to any of the functions **sigaddset()**, **sigdelset()**, **sigismember()**, **sigaction()**, **sigprocmask()**, **sigpending()**, or **sigsuspend()**, the results are undefined.

See Also

libc, sigaction(), sigaddset(), sigdelset(), sigfillset(), sigismember()

Notes

If your program is compiled using the System V Release 4 compilation environment, this is a function that is linked in from **libc**. If not, a macro form is used.

sigfillset() — Signal Function (libc)

Initialize a set of signals
#include <signal.h>
int sigfillset (*set***)**
sigset_t **set**;

sigfillset() is one of a set of signalling functions that manipulate data objects addressable by the application, instead of a set of signals known to the system. It initializes the set of signals to which *set* points, such that all standard signals are included.

sigfillset() returns zero if all goes well. If a problem occurs, it returns -1 and sets **errno** to an appropriate value.

Applications must call either **sigemptyset()** or **sigfillset()** at least once for each object of type **sigset_t** prior to any other object. If such an object is not initialized in this way, but is supplied as an argument to any of the functions **sigaddset()**, **sigdelset()**, **sigismember()**, **sigaction()**, **sigprocmask()**, **sigpending()**, or **sigsuspend()**, the results are undefined.

See Also

libc, sigaction(), sigaddset(), sigdelset(), sigemptyset(), sigismember()

Notes

If your program is compiled using the System V Release 4 compilation environment, this is a function that is linked in from **libc**. If not, a macro form is used.

sighold() — System Call (libc)

Place a signal on hold
#include <signal.h>
int sighold (*sigtype***)**
int *sigtype*;

sighold() is a member of the **sigset()** family of signal-handling system calls. It is equivalent to the system call

```
sigset(sigtype, SIG_HOLD);
```

This, in effect, places the signal *sigtype* "on hold" until the system call **sigrelse()** is invoked for it. Only one "copy" of *sigtype* can be held at a time.

Thus, you can use **sighold()** and **sigrelse()** to defer processing of the signal *sigtype*. This permits you to protect a portion of your application from this signal until it is ready to process it.

See the Lexicon entry for **signal()** for a list of recognized signals. Note that signal **SIGKILL** cannot be held.

sighold() returns zero if all went well. If something went wrong, it returns -1 and sets **errno** to an appropriate value.

See Also

libc, sigignore(), signal(), sigpause(), sigrelse(), sigset()

Notes

For more information on the **sigset()** family of signal-handling system calls, see the Lexicon entry for **sigset()**.

sigignore() — System Call (libc)

Tell the system to ignore a signal
#include <signal.h>
int sigignore (*sigtype***)**
int *sigtype*;

sigignore() is a member of the **sigset()** family of signal-handling system calls. It is equivalent to the system call

```
sigset(sigtype, SIG_IGN);
```

This, in effect, tells the system to ignore all signals of type **sigtype**.

See the Lexicon entry for **signal()** for a list of recognized signals. Note that signal **SIGKILL** cannot be ignored.

sigignore() returns zero if all went well. If something went wrong, it returns -1 and sets **errno** to an appropriate value.

See Also

libc, sighold(), signal(), sigpause(), sigrelse(), sigset()

Notes

For more information on the **sigset()** family of signal-handling system calls, see the Lexicon entry for **sigset()**.

sigismember() — Signal Function (libc)

Check if a signal is a member of a set
#include <signal.h>
int sigismember (*set***,** *signo***)**
sigset_t **set***set***;**
int *signo***;**

sigismembeer() is one of a set of signalling functions that manipulate data objects addressable by the application, instead of a set of signals known to the system. It tests whether the signal *signo* is a member of the set of signals to which *set* points.

If *signo* is a member of *set*, **sigismember()** returns zero. If *signo* is set to an invalid or unsupported value, it returns -1 and sets **errno** to **EINVAL**.

See Also

libc, sigaction(), sigaddset(), sigdelset(), sigemptyset(), sigfillset()

Notes

If your program is compiled using the System V Release 4 compilation environment, this is a function that is linked in from **libc**. If not, a macro form is used.

siglongjmp() — General Function (libc)

Perform a non-local goto and restore signal mask
#include <setjmp.h>
void siglongjmp(*environ***,** *value***)**
sigjmp_buf *environ***; int** *val***;**

siglongjmp() behaves like the function **longjmp()**, except that it also restores the signal mask.

environ points to an array of type **sigjmp_buf**, which is declared in header file **setjmp.h**. It must have been initialized by a call to **sigsetjmp()**. *value* is the integer value to be returned to the function that called **sigsetjmp()**.

See Also

libc, sigaction(), sigprocmask(), sigsetjmp(), sigsuspend()
POSIX Standard, §8.3.1

signal() — System Call (libc)

Specify action to take upon receipt of a given signal
#include <signal.h>
int (*signal(*sigtype, function***))()**
int *sigtype***, (***function***)();**

A process can receive a *signal*, or interrupt, from a hardware exception, terminal input, or a **kill()** call made by another process. A hardware exception might be caused by an illegal instruction or a bad machine address. The terminal interrupt character (described in detail in the Lexicon entry **tty**) generates a process interrupt (and in one case a core dump file for debugging purposes).

signal() tells the signal handler what to do when the current process receives signal *sigtype*. *sigtype* is the signal to process, as defined below. *function* points to the routine to execute when *sigtype* is received. This can be a function of your own creation; or you can use one of the following macros, which expand into pointers to system-defined functions:

SIG_DFL
This is the default action. The process terminates just as if it called the function **exit()**. In addition, the system writes a core file in the current working directory if *sigtype* is any of the following: **SIGQUIT**, **SIGSYS**, **SIGTRAP**, or **SIGSEGV**. (Note that this behavior applies only to executables for which you have write permission. If you lack write permission on an executable, then no core file is written.) For more information on core files, see the Lexicon entry **core**.

SIG_IGN
Ignore *sigtype*. The system discards all signals of this type.

signal() returns a pointer to the previous action. If *sigtype* is not a recognized signal, **signal()** returns **(int (*)())-1**.

With the exception of **SIGKILL** and **SIGTRAP**, caught signals are reset to the default action **SIG_DFL**. To catch a signal again, the routine to which *function* points must reissue the call to **signal()**.

The following list names the signals that **signal()** can process, as defined in the header file **signal.h**. Note that the signal **SIGKILL**, which kills a process, can be neither caught nor ignored. Signals marked by an asterisk produce a core dump if the action is **SIG_DFL**:

SIGHUP . . . . . Hangup
SIGINT . . . . . Interrupt
SIGQUIT* . . . . . Quit
SIGILL*. . . . . Illegal instruction
SIGTRAP* . . . . . Trace trap
SIGIOT . . . . . IOT instruction
SIGABRT* . . . . . To be replaced by SIGIOT
SIGEMT . . . . . Emulator trap
SIGFPE* . . . . . Floating-point exception
SIGKILL . . . . . Kill
SIGBUS. . . . . Bus error
SIGSEGV* . . . . . Segmentation violation
SIGSYS* . . . . . Bad argument to system call
SIGPIPE . . . . . Write to pipe with no readers
SIGALRM . . . . . Alarm
SIGTERM . . . . . Software termination signal
SIGUSR1. . . . . User-defined signal
SIGUSR2. . . . . User-defined signal
SIGCLD. . . . . Death of a child
SIGCHLD. . . . . Death of a child
SIGPWR . . . . . Restart
SIGWINCH. . . . . Window change
SIGPOLL. . . . . Polled event in stream

A signal may be caught during a system call that has not yet returned. In this case, the system call appears to fail, with **errno** set to **EINTR.** If desired, such an interrupted system call may be reissued. System calls which may be interrupted in this way include **pause()**, **read()** on a device such as a terminal, **write()** on a pipe, and **wait()**.

Example

The following program demonstrates **signal()**, **kill()**, **getpid()**, and **fork()**.

```
#include <signal.h>

int got_it; /* Each side gets its own copy of all data at the fork */
int errset;

/*
 * Control comes here on SIGTRAP.  Do no I/O in signal function.
 * Reset the signal if you ever want another.
 */

void
sig_ser()
{
      got_it = 1; /* tell the child we got it */
```

```
      if (0 > signal(SIGTRAP, sig_ser))   /* reset the signal */
            errset = 1;
}

main()
{
      int count;
      int child, parent;

      parent = getpid();      /* Both sides will get a copy */

      if (signal(SIGTRAP, sig_ser) < 0) { /* sets for both sides */
            perror("signal set failed");
            exit(0);
      }

      if (child = fork()) {   /* parent gets the child's id */
            for (count = 0; count < 3; count++) {
                  kill(child, SIGTRAP);   /* signal the child */

                  while(!got_it)          /* wait for signal */
                        sleep(1);
                  if (errset)
                        perror("parent: signal reset failed");

                  printf("parent got signal %d\n", count);
                  got_it = errset = 0;
            }
            exit(0);
      }

      for (count = 0; count < 3; count++) {
            while(!got_it)                /* wait for signal */
                  sleep(1);
            if (errset)
                  perror("child: signal reset failed");
            printf("child got signal %d\n", count);   /* show we got it */

            kill(parent, SIGTRAP);        /* signal the parent */
            got_it = errset = 0;
      }
      exit(0);
}
```

See Also

kill, kill(), libc, ptrace(), sh, sigaction(), signame, sigset()
ANSI Standard, §7.7.1.1

Notes

The function **signal()** predates the **sigset()** and **sigaction()** sets of signal-handling functions. *Never* combine **signal()** with any of the **sigset()** or **sigaction()** families of functions: use one or the other, but not both. For a description of how **signal()** differs from **sigset()** and **sigaction()**, see their Lexicon entries.

signal.h — Header Files

Define signals
#include <signal.h>

The header file **signal.h** defines manifest constants that name all of the machine-independent signals that the COHERENT system uses to communicate with its processes.

See Also

header files, kill, signal()
ANSI Standard, §7.7
POSIX Standard, §3.3.1

signame — Global Variable

Array of names of signals
#include <signal.h>
extern char *signame[_SIGNAL_MAX];

When a program terminates abnormally, its parent process receives a byte of termination information from the system call **wait()**. This byte contains a signal number, as defined in the header file **signal.h**. For example, **SIGINT** indicates an interrupt from the terminal.

The array **signame**, indexed by signal number, contains strings that give the meaning of each signal. Thus, **signame[SIGINT]** points to the string "interrupt". For portability reasons, all programs which wait on child processes (such as the shell **sh**) should use **signame**.

Files

<signal.h>

See Also

Programming COHERENT, sh, signal(), wait

Notes

Please note that through revision 10 of the COHERENT manual, the signal numbers in **signame[]** were offset by one. That is erroneous: the signal numbers are not offset at all.

In standard implementations of UNIX, the manifest constant **NSIG** was one larger than the number of signals. Prior to release 4.2, however, COHERENT defined **NSIG** as being equal to the number of signals. Beginning with release 4.2, COHERENT defines **NSIG** to conform to the UNIX usage, and introduces the manifest constant **_SIGNAL_MAX**, which is equal to the number of signals. If your code depends upon the old definition of **NSIG**, you should replace it with **_SIGNAL_MAX**.

Please note that **signame[]** is obsolete, and will be removed from future releases of COHERENT. Please do not incorporate it into new code, and you should try to remove it from existing code.

sigpause() — System Call (libc)

Pause until a given signal is received
#include <signal.h>
int sigpause (*sigtype***)**
int *sigtype***;**

sigpause() is a member of the **sigset()** family of signal-handling system calls. It pauses until the signal *sigtype* is received. If, however, a signal of type *sigtype* had previously been "placed on hold" by a call to **sighold()**, **sigpause()** releases the signal for processing, just as if you had invoked the system call **sigrelse()**.

See the Lexicon entry for **signal()** for a list of recognized signals.

sigpause() returns zero if all went well. If something went wrong, it returns -1 and sets **errno** to an appropriate value.

See Also

libc, sighold(), sigignore(), signal(), sigrelse(), sigset()

Notes

For more information on the **sigset()** family of signal-handling system calls, see the Lexicon entry for **sigset()**.

Note that invoking

```
sigpause(SIGCHLD)
```

with no children pauses forever.

sigpending() — System Call (libc)

Examine signals that are blocked and pending
#include <signal.h>
int sigpending(*stash***)**
sigset_t **stash**;

sigpending() retrieves the signals that have been sent to the calling process but have been blocked by the calling process's signal mask. *stash* points to the area of memory where the retrieved signals are to be stored.

sigpending() returns zero if all goes well. It returns -1 and sets **errno** to **EFAULT** if *stash* points outside the process's allocated address space.

See Also

libc, sigaction(), signal()
POSIX Standard, §3.3.6

sigprocmask() — System Call (libc)

Examine or change the signal mask
#include <signal.h>
int sigprocmask(*how***,** *set***,** *old_set***)**
int *how***; const sigset_t** ***set***; sigset_t** ***old_set***;**

sigprocmask() examines or changes the calling process's signal mask.

how defines how to modify the mask, as follows:

SIG_BLOCK
Add to the signal mask the set of signals to which *set* points.

SIG_UNBLOCK
Remove from the signal mask the set of signals to which *set* points.

SIG_SETMASK
Replace the current signal mask with the set of signals to which *set* points. If *old_set* is not NULL, **sigprocmask()** stores the old mask in the space to which it points.

If *set* is NULL, **sigprocmask()** ignores the value of *how*; thus, you can use the call to find which signals are in the signal mask.

If any unblocked unblocked signals are pending after you call **sigprocmask()**, at least one of those signals will be delivered before **sigprocmask()** returns.

If all goes well, **sigprocmask()** returns zero. **sigprocmask()** returns -1 if either of the following conditions is true:

- *how* is not set to a recognized value. **sigprocmask()** sets **errno** to **EINVAL**.
- *set* or *old_set* points outside the process's allocated address space. **sigprocmask()** sets **errno** to **EFAULT**.

In either error condition, **sigprocmask()** does not change the signal mask.

See Also

libc, signal(), siglongjmp(), sigsetjmp()
POSIX Standard, §3.3.5

sigrelse() — System Call (libc)

Release a signal for processing
#include <signal.h>
int sigrelse (*sigtype***)**
int *sigtype***;**

sigrelse() is a member of the **sigset()** family of signal-handling system calls. It releases the signal *sigtype*, which had previously been "placed on hold" by the system call **sighold()**. Only one "copy" of *sigtype* can be held at a time. Thus, you can use **sighold()** and **sigrelse()** to defer processing of the signal *sigtype*. This permits you to protect a portion of your application from this signal until it is ready to process it.

When *sigtype* is released, it is processed by the function that had set for it by the system call **sigset()**. If **sigset()**

has not been invoked for *sigtype*, then the system uses the function to which **SIG_DFL** points. **SIG_DFL** terminates the process, just as if it called the function **exit()**. In addition, it dumps core if *sigtype* is any of the following: **SIGQUIT**, **SIGRESET**, **SIGTRAP**, **SIGSEGV**, or **SIGSYS**.

Note that signal **SIGKILL** cannot be held. See the Lexicon entry for **signal()** for a list of recognized signals.

sigrelse() returns zero if all went well. If something went wrong, it returns -1 and sets **errno** to an appropriate value.

See Also

libc, sighold(), sigignore(), signal(), sigpause(), sigset()

Notes

For more information on the **sigset()** family of signal-handling system calls, see the Lexicon entry for **sigset()**.

sigset() — System Call (libc)

Specify action to take upon receipt of a given signal
#include <signal.h>
void (*sigset (*sigtype***,** *function***))()**
int *sigtype***;**
void (**function***)**();**

sigset() tells the signal handler what to do when the current process receives signal *sigtype*.

sigtype identifies the signal being sought. For a list of recognized signals, see the Lexicon entry for **signal()**. Note that the signal **SIGKILL**, which kills a process, can be neither caught nor ignored.

function points to the function to be executed when *sigtype* is received. This can be a function of your own creation; or you can use one of the following macros, which expand into pointers to system-defined functions:

SIG_DFL
: This is the default action. The process terminates just as if it called the function **exit()**. In addition, the system writes a core file in the current working directory if *sigtype* is any of the following: **SIGQUIT**, **SIGSYS**, **SIGTRAP**, **SIGSEGV**, or **SIGSYS**. For more information on core files, see the Lexicon entry **core**.

SIG_IGN
: Ignore *sigtype*. The system discards all signals of this type.

SIG_HOLD
: Hold *sigtype*. The signal is held until the process calls **sigrelse()** to release it. Once the signal is released, it is processed as defined by **sigset()**. Only one "copy" of *sigtype* can be held at any given time.

If all goes well, **sigset()** returns a pointer to the routine that had previously been in place to process *sigtype*. If something goes wrong (e.g., *sigtype* is not defined in **signal.h**), **sigset()** returns **SIG_ERR** and sets **errno** to an appropriate value.

sigset() Versus signal()

The COHERENT system also include the system call **signal()**, which also handles signals. **signal()** predates **sigset()** and its related functions **sighold()**, **sigignore()**, **sigpause()**, and **sigrelse()**. You should *never* combine **signal()** with the **sigset()** family of functions: use one or the other, but not both.

The **sigset()** functions differ from **signal()** in the way they handle signals while a signal is being processed: **signal()** automatically invokes **SIG_DFL** for *sigtype* while its *function* is executing; whereas **sigset()** and its related functions invoke **SIG_HOLD**.

Thus, with **signal()**, sending signal *sigtype* to a program while that signal's *function* is already executing will trigger the default action, which in most instances is to exit from the program. The signal-handling function itself can call **signal()** to reset the signal-handler to point to itself or another function; however, there remains a brief interval of vulnerability between the time the signal-processing function is called and the time it calls **signal()** to program the signal handler. With **sigset()**, however, if another *sigtype* is received while its *function* processing, the signal handler holds it, and releases it automatically after *function* returns.

sigset() also differs from **signal()** in the way in which the signal-handler is reset once *sigtype* has been processed. With **signal()**, *function* is automatically reset to **SIG_DFL** just before a signal of type *sigtype* is processed. If you wish *sigtype* always to be processed by *function*, you must explicitly re-invoke **signal()** for *sigtype* within *function*. However, the **sigset()** family of routines always process *sigtype* by the routine to which *function* points until you

explicitly change it.

See Also

libc, sighold(), sigignore(), signal(), sigpause(), sigrelse()

Notes

Functions called from within a signal handler should be re-entrant; this includes the standard I/O library. Thus, in general, it is not a good idea to call **printf()** from inside a signal handler. The risk is that a signal will arrive while the main program is updating a static structure, or calling **malloc()**; then the signal handler will run when something is not in a consistent state, with unpredictable results.

sigsetjmp() — General Function (libc)

Save machine state and signal mask for non-local jump
#include <setjmp.h>
int sigsetjmp(*environ***,** *savemask***)**
sigjmp_buf *environ***;**
int *savemask***;**

sigsetjmp() performs the same action as the function **setjmp()**, except that if the value of *savemask* is not zero, it saves the process's signal mask as well as the machine state into the array to which *environ* points.

See Also

libc, sigaction(), siglongjmp(), sigprocmask(), sigsuspend()
POSIX Standard, §8.3.1

sigsuspend() — System Call (libc)

Install a signal mask and suspend process
#include <signal.h>
int sigsuspend(*set***)**
const sigset_t **set***;*

sigsuspend() replaces the process's signal mask with the set of signals to which *set* points, then suspends the current process until it receives a signal that either terminates the process or invokes a signal-handling function.

If the received signal terminates the process, **sigsuspend()** does not return. If, however, the received signal invokes a signal-handling function, **sigsuspend()** restores the original signal mask.

Because **sigsuspend()** indefinitely suspends execution of the process, there is no return value that indicates successful completion. If something goes wrong, it returns -1 and sets **errno** to an appropriate value. **sigsuspend()** fails if either of the following is true:

- The calling process catches a signal and grabs control from the signal-catching function. **sigsuspend()** sets **errno** to **EINTR**.
- *set* points outside the process's allocated address space. **sigsuspend()** sets **errno** to **EFAULT**.

See Also

libc, siglongjmp(), signal(), sigsetjmp()
POSIX Standard, §3.3.7

sin() — Mathematics function (libm)

Calculate sine
#include <math.h>
double sin(*radian***) double** *radian***;**

sin() calculates the sine of its argument *radian*, which must be in radian measure.

Example

The following example uses the functions **sin()** and **cos()** to paint sine and cosine on the screen. It is by Dmitry Gringauz (dmitry@golem.com).

```
#include <math.h>
#include <stdio.h>
```

```
#define MAX_X 79 /* X dimension of screen */
#define MAX_Y 23 /* Y dimension of screen */
char screen[MAX_X][MAX_Y]; /* the screen matrix */

main()
{
      double pi = 3.14159, i, result;
      int x = 0, y = 0, mid_x = (MAX_X-1)/2, mid_y = (MAX_Y-1)/2;

      /* blank (dot) out the screen */
      for (y = 0; y < MAX_Y; y++)
            for (x = 0; x < MAX_X; x++)
                  screen[x][y] = '.';

      /* build the "axis" */
      for (x=0; x < MAX_X; x++)
            screen[x][mid_y] = '-';
      for (y = 0; y < MAX_Y; y++)
            screen[mid_x][y] = '|';

      /* make center and arrows */
      screen[mid_x][mid_y] = '+';
      screen[mid_x][0] = '^';
      screen[MAX_X-1][mid_y] = '>';

      /* do the sin() and cos() thing */
      for (i = -pi; i <= pi; i = i + 2.0 / (MAX_X)) {
            result = sin(i) ;

            x = i*mid_x/pi + mid_x;
            y = mid_y*(-1.0*result) + mid_y;

            if (x >= MAX_X)
                  x = MAX_X - 1;

            if (y >= MAX_Y)
                  y = MAX_Y - 1;

            screen[x][y] = '*';
            result = cos(i) ;

            x = i*mid_x/pi + mid_x;
            y = mid_y*(-1.0*result) + mid_y;

            if (x >= MAX_X)
                  x = MAX_X - 1;

            if (y >= MAX_Y)
                  y = MAX_Y - 1;
            screen[x][y] = '*';
      } /* i */

      /* print the screen */
      for (y = 0; y < MAX_Y; y++) {
            for (x = 0; x < MAX_X; x++)
                  printf("%c", screen[x][y]);
            printf("\n");
      } /* y */
}
```

See Also

cos(), cosh(), libm, sinh()
ANSI Standard, §7.5.2.6
POSIX Standard, §8.1

sinh() — Mathematics Function (libm)

Calculate hyperbolic sine
#include <math.h>
double sinh(*radian***) double** *radian***;**

sinh() calculates the hyperbolic sine of *radian*, which is in radian measure.

See Also

libm
ANSI Standard, §7.5.3.2
POSIX Standard, §8.1

size — Command

Print size of an object file
size [*file ...***]**

size prints the sizes, in bytes, of the segments of each *file* (in decimal) and also prints the total size of all the segments (in both decimal and octal). Each *file* must be an object file.

size outputs one line for each file, listing the following segments:

.text
.data
.bss

See Also

coff.h, commands, l.out.h

Notes

size makes no concessions to machines that use hexadecimal.

sizeof — C Keyword

Return size of a data element

sizeof is a C operator that returns a constant **int** that gives the size of any given data element. The element examined can be a data object, a portion of a data object, or a type cast. **sizeof** returns the size of the element in **char**s; for example

```
long foo;
sizeof foo;
```

returns four, because a **long** is as long as four **char**s.

sizeof can also tell you the size of an array. This is especially helpful for use with external arrays, whose size can be set when they are initialized. For example:

```
char *arrayname[] = {
      "COHERENT",
      "COHware volume I",
      "COHERENT Device Driver Kit",
        "GNU C/C++"
};

main()
{
      printf("\"arrayname\" has %d entries\n",
            sizeof(arrayname)/sizeof char*);
}
```

sizeof is especially useful in **malloc()** routines, and when you need to specify byte counts to I/O routines. Using it to set the size of data types instead of using a predetermined value will increase the portability of your code.

See Also

C keywords, data types, operators
ANSI Standard, §6.3.3.4

sleep — Command

Stop executing for a specified time
sleep *seconds*

The command **sleep** suspends execution for a specified number of *seconds*. This routine is especially useful with other commands to the shell. For example, typing

```
(sleep 3600; echo coffee break time) &
```

executes the **echo** command in one hour (3,600 seconds) to indicate an important appointment.

See Also

alarm(), commands, ksh, pause(), sh

sleep() — General Function (libc)

Suspend execution for interval
#include <unistd.h>
sleep(*seconds***)**
unsigned *seconds***;**

sleep() suspends execution for not less than *seconds*.

Example

The following example demonstrates how to use **sleep()**:

```
#include <unistd.h>
main()
{
      printf("Waiting for Godot ...\n");

      for ( ; ; ) {
            sleep(5);   /* sleep for five seconds */
            printf("... still waiting ...\n");
      }
}
```

See Also

libc, nap(), unistd.h
POSIX Standard, §3.4.3

Notes

To make a program sleep for less than one second, use the system calls **nap()** or **poll()**. For an example, see the Lexicon article for **poll()**.

smail — Command

Mail delivery system
smail [*flags* **]** *address ...*

smail is the program that receives and delivers mail. It accepts mail from a source either on your local host or on a remote host, and delivers that mail to its destination — again, either on your local host or another remote host. **smail** does not provide a user interface for typing mail or reading it; to do so, you must use a "mailer" program, such as **mail** or **elm**.

You will rarely, if ever, need to invoke **smail** directly. You may modify one of its configuration files from time to time, but **smail** normally is invoked only by other programs. The rest of this article gives **smail**'s command-line options and describes how it works. You will find this information useful should you wish to reconfigure your mail system, or chase down a bug.

smail can be invoked under a variety of names. Each name indicates the major use to which **smail** will be put, e.g., receiving local mail, receiving remote mail, attempting to deliver undelivered mail, or displaying information about undelivered mail. These names are described below; each also has its own Lexicon entry.

Command-line Options

smail recognizes the following command-line options:

-bc Display the contents of file **COPYING**, which is distributed with the source code for **smail**. This file details what your rights and restrictions the authors of **smail** have set upon their work.

-bd Listen for connection requests on a socket bound in the Internet domain. When a connection occurs, conduct an Simple Mail Transfer Protocol (SMTP) conversation with the peer process on the other system. This option currently is not implemented under COHERENT, as COHERENT does not yet support networking.

-bi Initialize the **aliases** file. The file that it builds depends upon whether you also use option **-oA** on the command line.

By default, **smail** under COHERENT is compiled with the **GDBM** package. **GDBM** is a set of functions that permit a program to build and read a simple hashed data base; for details on how it works, see the Lexicon entry for **libgdbm**. Thus, when you also use option **-oA***file* to name an aliases file, **smail** invokes the command **/usr/lib/mail/newaliases** to compile the contents of *file* into a DBM data base.

-bm *address ...*
Deliver mail to each *address*.

-bP *address*
Assume that each *address* on the command line is a configuration-file variable, and write its value onto the standard output. For example, the command

```
smail -bP hostnames max_message_size
```

produces output of the form:

```
lepanto.com
102400
```

If you also use the flags **-d** or **-v** on the command line, **smail** also displays the variable names. Thus, the command

```
smail -bP -v max_message_size
```

prints something like the following:

```
max_message_size=102400
```

The command

```
smail -bP primary_name
```

prints the primary (or "canonical") name for the local host that **smail** uses, and command

```
smail -bP config_file
```

prints the name of the primary configuration file. The command

```
smail -bP help
```

prints a verbose listing of every variable plus its type, one variable per line. Finally, command

```
smail -bP all
```

prints a verbose listing of every variable and its value. It is equivalent to the command **smail -bP -v** followed by a list of the name of every configuration variable.

-bp List information about the messages that currently reside in **smail**'s input spool directories. This is **smail**'s default mode of operation when you invoke it under the name **mailq**. When you also use the flags **-v** or **-d**, **smail** displays the transaction-log entries for each message, to show what has happened to the message so far.

-bS Read SMTP commands from the standard input, but do not write SMTP replies onto the standard output. Report failures via mail rather than through reply codes.

This option is suitable for setting up a batched form of SMTP between machines over a remote execution service like UUCP. This is the default mode of operation if you invoke **smail** under the name **rsmtp**.

-bs Read SMTP commands from standard input, and write SMTP replies onto the standard output. The following SMTP commands are implemented:

HELO	**MAIL**	**FROM**	**RCPT**
TO	**DATA**	**RSET**	**NOOP**
VRFY	**EXPN**	**QUIT**	

This is the default mode of operation if you invoke **smail** under the name **smtpd**.

-bt Run **smail** in test-address mode: **smail** reads addresses from standard input, parses them, and writes its result onto the standard output. This is primarily useful for debugging **smail** or debugging new **smail** routers.

-bV See option **-V**, below.

-bv Verify an address. **smail** reads each address you list on its command line, subjects it to aliasing and forwarding expansions, then subjects it to host routing or resolving, and finally prints the resolved address onto the standard output. You can then check whether the resolved address matches what you expect. If **smail** cannot resolve an address, it prints an explanation of why it cannot.

-C *file* Use *file* as the primary configuration file — i.e., the file that holds global attributes. **smail** resets the effective user identifier and group identifier to those of the real user and group, to avoid problems should **smail** be **setuid** to the superuser.

If *file* is '-', then **smail** does not use a primary configuration file. You should use this only for debugging.

-d[*number***]**
Turn on debugging. *number* sets the level of debugging; the default level is one. No white space must separate the option and *number*. Please note that **-d** and **-v** are identical; **smail** recognizes both for historical reasons.

-D *file* Write debugging information into *file*. Normally, using option **-v** or **-d** to generate debugging output also disables background delivery of mail, because programs should not continue to write to the standard error after the mail process exits; however, if you name a debugging-output file, background delivery can continue.

-ee
-oee These options refer to a "berkenet" style of error-processing that **smail** does not support. If used, **smail** mails an error message back to you.

-em
-oem Mail error messages to the sender. This is the default.

-ep Write error messages onto the standard-error device.

-eq If an error occurs, do not notify the sender of it. This only works for mail being delivered locally: an error that occurs on a remote host's mail system still generates a mail message to the sender. To set this behavior on both the local host and a remote host, supply a header that reads:

```
Precedence: junk
```

-ew
-oew Mail errors to the sender, just as with option **-m**. With some mail-delivery programs, this option asks the program to invoke the command **write** to write errors onto the sender's screen, should she be logged in.

-F *fullname*
Set to fullname the full name of the sender for incoming mail. Use this option only if you wish to use smail to receive a single mail message from the standard input.

-f *sender*
Set to *sender* the address for incoming mail. Use this option only if you want **smail** to receive a single mail message from the standard input.

-h *number*
Set to *number* the hop count for a message. If this command-line option is not used, **smail** computes the hop count from the number of **Received:** fields in the message's header. **smail** uses the hop count as a primitive method of detecting an infinite loop: if the hop count is too large, **smail** rejects the mail.

NB, an infinite loop occurs when two sites each think that a given user resides on the other. A message mailed to that user will ping-pong between the sites; unless the message is stopped somehow, its header can grow infinitely large.

-I Use the "hidden-dot" algorithm when reading a message. If a message contains a line that begins with one or more periods, **smail** removes that leading period; a line that consists of a single period terminates the message. This option is always set for messages received via SMTP.

-i A line that consists of a single period does not terminate an incoming message. This is the default if you invoke smail under the name **rmail**.

-m If a user mails a message to an alias list or mailing list that includes her name, send a copy of the message to that user. By default, if the user mails a message to a list that includes her name, **smail** does *not* send a copy of a message back to her.

-N Disable delivery of a message. **smail** performs all other processing, and the transport programs are expected to go through most of the steps involved in delivery. Use this option when you wish to debug smail but do not want to have the messages delivered.

-n Do not process aliases. With this option, **smail** will not expand entries in alias files; however, it will still expand entries in mailing-list files and forwarding files.

-oC *file* See option **-C**, above.

-odb If mail is to be delivered, deliver it in the background. Note that background delivery is not currently supported in the SMTP modes: mail is delivered in the foreground.

-oD *file* Use *file* as the **directors** file, instead of the default **/usr/lib/mail/directors**. **smail** resets the effective user and group identifiers to those of the real user and group, to avoid problems should an installation setuid **smail** to the superuser. If file is '-', **smail** does not read a **directors** file. Use this option only when you are debugging **smail**.

-odf If mail is to be delivered, deliver it in the foreground.

-oE *file* Use *file* as the delivery-retry control file, instead of the default **/usr/lib/mail/retry**. **smail** resets the effective user and group identifiers to those of the real user and group, to avoid problems should an installation setuid **smail** to the superuser. If file is '-', **smail** does not read a retry file. Use this option only when you are debugging **smail**.

-oep See option **-ep**, above.

-oeq See option **-eq**, above.

-oI See option **-I**, above.

-oi See option **-i**, above.

-oL *directory*
Use *directory* as the library directory — that is, the directory that holds configuration files and mailing-list directories. This overrides the default value compiled into smail through its option **smail_lib_dir** (under COHERENT **/usr/lib/smail**), as well as any name set in a configuration file.

-oMr *sender_proto*
Use *sender_proto* as the protocol by which sending host delivers the mail message. You can include this value in expansion strings via the variable **$sender_proto**.

-oMs *sender_host*
Set to sender_host the system that can send the mail message. You can include this value in expansion strings via the variable **$***sender_host*.

-om See option **-m**, above.

-oQ *file* Set the path name of the host-name qualification file to *file*, instead of the default **/usr/lib/mail/qualify**. **smail** resets the effective user and group identifiers to those of the real user and group, to avoid problems should an installation setuid **smail** to the superuser. If *file* is '-', **smail** does not read a qualify file. Use this option only when you are debugging **smail**.

-oR *file* Use *file* as the router file, instead of the default **/usr/lib/mail/routers**. **smail** resets the effective user and group identifiers to the real user and group identifiers, to avoid problems should an installation setuid **smail** to the superuser. If file is '-', **smail** does not read a router file. Use this option only when you are debugging **smail**.

-oT *file* Use *file* as the transport file, instead of the default **/usr/lib/mail/transports**. **smail** resets the effective user and group identifiers to those of the real user and group, to avoid problems should an installation setuid **smail** to the superuser. If file is '-', **smail** does not read a transport file. Use this option only when you are debugging **smail**.

-oU Tell **smail** to report memory usage when it exits.

-oX *mail-service*
Tell **smail** to listen for SMTP requests on the TCP/IP service or port *mail-service*. You can use this option with **-bd** mode to define alternate debugging versions of **smail**'s SMTP listening daemon; this can be useful when you test a new installation.

Please note that because COHERENT does not yet support networking, this option does nothing.

-oQ
-odq Spool incoming messages, but do not deliver them until later queue. This mode of operation is somewhat more efficient in terms of CPU usage, but slows down the flow of mail.

-q[*interval***]**
Force **smail** to process its input spool directory. If you set *interval*, **smail** continually checks its input-spool directory, and sleeps for *interval* between checks. *interval* is a string that consists of a number followed by one of the following letters to indicate unit of time:

s	seconds
m	minutes
h	hours
d	days
w	weeks
y	years

For example, option **-q2h30m** tells **smail** to check its input spool directory every two hours and 30 minutes. This flag is useful with the **-bd** mode of operation, as it awakens the daemon process after each interval to process the queue. This is **smail**'s default mode of operation when you invoke it under the name **runq**.

-r *sender*
See option **-f**, above.

-t Extract addresses from the **To:**, **Cc:**, and **Bcc:** fields of the message header. This is useful for mailers that do not compute the recipient addresses themselves. In this mode, the addresses given on the command line will not receive mail, even as a result of expanding aliases or forwarding addresses. **smail** ignores this option unless it is in the mode set by command-line option **-bm** (which is the default mode).

-V Print the version smail onto the standard output.

Normal Use

A user agent can submit new mail message by invoking **smail** and passing it a message via the standard input. For example, mailers such as **mail** and **elm** submit mail by invoking **smail** with a command such as

```
smail -em -i address ...
```

Because **smail** also works correctly if invoked as **sendmail**, it is common to install **smail** as **/usr/lib/sendmail**, so that existing binaries on BSD systems, or other systems that currently run **sendmail**, need not be modified to run **smail** instead. This also lets you run applications that have been configured to send mail via **sendmail** without modifying their sources or recompiling.

Some user agents, such as GNU Emacs, may wish to have **smail** decipher the recipient list from the header. These programs can invoke **smail** with a command, such as:

```
smail -em -t -i
```

To receive mail over UUCP, **uuxqt** invokes the command **rmail**, which is a link to **smail**. **rmail** can also be another program that invokes **smail** directly as:

```
smail -em -i -fsender-address recipient address ...
```

An alternative method of receiving mail over UUCP is through the command **rsmtp**, which receives batched SMTP requests. This can be used between two sites running **smail** to gain many of the benefits of the SMTP protocol, such as the ability to use recipient addresses that **uux** cannot correctly pass to a remote **rmail** program. For

example, an address that contains quotations marks or spaces cannot be expected to pass correctly over an **uux-rmail** link, but will pass correctly over a **uux-rsmtp** link.

Addressing Under smail

The following describes how **smail** interprets an E-mail address.

smail understands domain-style addresses (e.g., **henry@mwc.com**) UUCP-style path names, (e.g., **mwc!lepanto!henry**), and local addresses (e.g., **henry**). It assumes that an address of the form *user@domain* is a domain address, that an address of the form *host***!***address* is a UUCP path, and anything else is a local address.

When it parses a mixed address (that is, an address that contains both a '!' and a '@'), **smail** gives precedence to '@' over '!'. Thus, it parses the address **a!b@c** as **(a!b)@c**, rather than **a!(b@c)**, which means that mail addressed to **a!b@c** is forwarded to system **c** instead of to system **a**.

Resolving Addresses

An E-mail address has two forms: internal and external. The internal form of an address is what appears on the **To:** line in the message's header. This is the recipient's address as typed by the person who mailed the message. This is regardless of whether the sender typed the recipient's full address, or typed an alias for the recipient. (For details on how to use aliases to address mail messages, see the Lexicon entry for **aliases**.) The external form of an address (also called the message's *envelope*), is the address that smail passes to the mail-delivery agent (either **uux** or **lmail**).

Resolving is the act of transforming an internal address into an envelope. It has two stages: host resolution and alias resolution. *Host resolution* (also called *routing*) is how **smail** figures out the identity of the computer to which it must send the message. If **smail** determines that the message must be delivered on your local machine, it then applies *alias resolution* (also called *alias expansion*) to the address. If the address proves to be an alias, **smail** expands the alias and again performs host resolution to find the machine to which it should deliver the message. If, however, the address names a user on your local machine, then **smail** hands the message to the local mailer **lmail** for delivery.

Although **smail** understands domain-style addresses (i.e., addresses that contain a '@' and are read from right to left), it can deliver mail only to UUCP paths (i.e., addresses that contain '!' characters and are read from left to right) and local addresses. Thus, it must resolve a domain address into a UUCP path or local address.

To resolve a domain-style address, **smail** must find the route to the most specific part of the domain, as specified in the routing file **/usr/lib/mail/paths**. Two degrees of resolution can occur:

Full Resolution
: **smail** finds the full route to the machine. In this case, **smail** either tacks the user specification onto the end of the machine's UUCP path, or resolves it into a local address, whichever is appropriate.

Partial Resolution
: **smail** finds a route for only the right portion of the domain specification; e.g., for

```
henry@lepanto.mwc.com
```

it finds **mwc.com** but cannot identify **lepanto**. Here, **smail** tacks the complete address (in the form **domain!user**) onto the end of the UUCP path. For example, if **smail** finds that the route to **mwc.com** is via systems **foo**, **bar**, and **baz**, it constructs the path:

```
foo!bar!baz!lepanto.mwc.com!henry
```

This assumes that routing program on system **baz** (perhaps **smail**, perhaps some other program) will recognize the token **lepanto.mwc.com** as being a domain rather than a host.

It is an error to route a partially resolved address to the local host (a null UUCP path), because the local host is responsible for resolving the address more fully.

The command-line option **-r** tells **smail** to attempt to route the first (leftmost) component of a UUCP path, regardless of whether it knows how to send mail directly to a site named further to the right in the path. This is called *always routing*. For example, if a mail message is address to

```
foo!bar!baz!mwc!lepanto!fred
```

option **-r** tells **smail** always to route the mail to **foo**, even if also knows how to route mail to **mwc**.

The command-line option **-R** tells **smail** to route mail to the rightmost host named on a UUCP path. This is called *reroute routing*. Use it if you have a very up-to-date routing table, and wish to bypass some obsolete routing

information in the current path.

If file **/usr/lib/mail/paths** does not contain a path to the remote system, **smail** forwards mail to the the host named in the entry **smart_path** in file **/usr/lib/mail/config**. This lets your system depend on another, better informed, system to deliver your mail. Note that before you name another system as your system's **smart_path**, you should get the permission of the person who administers that system. Please note that if you start to forward mail to a system without permission, that system's administrator may forward your mail to the bit bucket.

After **smail** resolves an address, it reparses the address to see if it is now a UUCP path or local address. If the new address turns out to be another domain address, **smail** complains. This error occurs when an address partially resolves to the local host.

By default, **smail** does not alter an explicit UUCP path of any mail message. If the stated path is unusable (i.e., the next host is unknown), then **smail** applies always-routing and attempts to deliver the message to first (leftmost) system named in the UUCP path. If this fails, **smail** then uses reroute-routing and again attempts to deliver the message. If this too fails, **smail** finally attempts to to find a path to a smart-host and passes the mail to it. And if that finally fails, **smail** mails an error message to user who mailed the message, and abandons any further attempt to deliver the message.

Headers

Document RFC822, which governs Internet mail, demands that a mail message contain certain entries in its header. These entries include one that begins with the string **To:**, one that begins with the string **From:**, and one that begins with the string **Date**. If a message's header does not contain one or more of these entries, **smail** inserts it.

Build the From: Line of a Message

The header of a mail message includes a line that begins **From:**. This line names the user who originated the message. This line is extremely important, as it will read by users and programs on the recipient's system to identify the sender, and possibly reply to the message.

smail collapses the **From_** and **>From_** lines within a mail message to generate a simple "from" argument, which it then uses to create its own **From:** line. This processing sometimes is called *from-ming* a message. The following gives the rules for from-ming:

- First, it concatenates all hosts named on remote from lines, separating them from each other by '!'s.
- It appends onto that concatenated address, the address from the last **From_** line.
- If that address is in domain format (i.e., the form *user*@*domain*), **smail** rewrites it in bang-path format (i.e., the form *domain*!*user*). If a host or domain names the local system, **smail** ignores it.
- Finally, **smail** removes redundant information from the **From_** line.

smail generates its own **From_** line. For mail that is to be forwarded via UUCP, **smail** generates a line of the form remote-from host; *host* is the UUCP host name (not the domain name), so that **From_** can indicate a valid UUCP path, thus leaving the sender's domain address in **From:**.

Undeliverable Mail

smail returns to sender all mail that is undeliverable. A message is declared to be undeliverable if the user is unknown, or if the user resides on an unknown host.

Logging

smail uses two log files:

/usr/spool/smail/log/logfile
: The log of every mail message that your system receives. Please note that if your system is busy, this file will quickly become very large. You should embed the command **/usr/lib/mail/savelog** in **root**'s **cron** file to ensure that this file is truncated and saved regularly. For details on **savelog** or **cron**, see their articles in the Lexicon.

/usr/spool/smail/log/paniclog
: The log of every mail that created a panic situation. If your system is configured properly, this file will never become large.

Registered Domains and Subdomains

You may wish to register your domain with the NIC. This will give you an internationally recognized e-mail address. For more information, send E-mail to **postmaster@internic.net**.

Once you have registered your domain, you can also set up subdomains for other systems, so they can receive information from the Internet via your system. The rest of this section discusses how to describe subdomains to your system, and related topics.

Let's say that you have registered your domain, and you have named it **mydomain**. To route mail systems subordinate to mydomain, do the following:

1. Insert the following entry into **/usr/lib/mail/paths**:

```
.mydomain.com  %s 50
```

This tells **smail** that the local host (i.e., your machine) must resolve that any address that ends in the suffix **.mydomain.com**, or it is an error.

2. For each site in **mydomain**, create two entries in **/usr/lib/mail/paths**. For example, for the site **foo.mydomain.com**, create the entries:

```
foo foo!%s  200
foo.mydomain.com  foo!%s  200
```

If the site bar.mydomain.com is fed by the route **froboz!florp!bar**, insert these lines into **paths**:

```
bar froboz!florp!bar!%s  200
bar.mydomain.com  froboz!florp!bar!%s  200
```

Note that you do not have to register subdomains with the NIC. Once you register the top-level domain with the NIC, you control the entire name space — you can subdomain to your heart's content.

The only restrictions on subdomaining may be with your Internet Nameserver. Most nameservers for UUCP domains publish a "wildcard" mail exchanger (MX) record, that essentially says, "send everything for ***.mydomain.com** to this Internet gateway." However, some nameserver managers require you to register every site in your domain, for which they provide a separate MX record. The advantage of this scheme is that anybody on the Internet who sends mail to your domain immediately receives an error message if the message is addressed to a non-existent site. For details, check with the person who manages your nameserver.

To route for an entire subdomain (e.g., **.subd.mydomain.com**), you must choose a gateway for that domain (e.g., **gateway.subd.mydomain.com**), and then use a line like this:

```
.subd.mydomain.com gateway!%s  200
```

smail automatically chooses the longest subdomain match it can find, so this rule applies before the **.mydomain.com %s** rule.

Note that the gateway need not be in the subdomain itself. You could have a line elsewhere in the **paths** file on **mydomain** that says:

```
gateway.mydomain.com  gateway!%s  200
```

Your main gateway may also have information about machines in subdomains, although this is not necessary. For instance, if your main machine is directly connected to a machine in a subdomain, you may want to put this information into **paths**, so the mail will not go through the gateway for that domain.

For example, the machine **smith.subd.mydomain.com** might be directly connected to your master gateway, **mydomain.com**:

```
smith  smith!%s  200
smith.subd.mydomain.com smith!%s  200
```

Without this rule, mail for **smith** would be queued for forwarding through **gateway.subd.mydomain.com**.

Compatibility With sendmail

smail was designed to be a plug-in replacement for the BSD program **sendmail**, in that external programs can call **smail** in the same way that they call **sendmail** and expect similar results. However, **smail** is completely different internally and has entirely different configuration files. As a result, the option **-o** to **smail** only sets a few configuration parameters that were believed to be commonly used by other programs. Also, for convenience, some new (upper-case only) parameters are defined only in **smail**. **smail** ignores attempts to use this flag to set other

options. For a complete list of the **-o** options that **smail** recognizes, see the section on command-line options, above.

For compatibility with other software systems (in particular, the Taylor UUCP package), COHERENT links **smail** to command **/usr/lib/sendmail**. Thus, a program that expects to use **sendmail** can use **smail** without being recompiled or reconfigured.

Configuration Files

For most sites, the configuration compiled into **smail** is sufficient, and thus no configuration files are needed. However, you can use any or all of the following optional configuration files to restructure how **smail** behaves on your system:

/usr/lib/mail/config
General configuration. This file can override compiled-in configuration, including the names of any of the following configuration files.

/usr/lib/mail/directors
Configuration file for **smail** directors, i.e., configured methods for resolving local addresses.

/usr/lib/mail/routers
Configuration file for **smail** routers, i.e., configured methods for resolving or routing to remote hosts.

/usr/lib/mail/transports
Configuration for **smail** transports, i.e., configured methods of mail delivery.

The contents of file **config** dictate how **smail** configures its internal workings — where it looks to find other configuration files, where it should send error messages, and so on. The contents of **routers**, **directors**, and **transports** together tell **smail** how to deliver mail both on your local system and on remote systems. The following describes how these files work together.

When **smail** is given a list of addresses to which a message is to be delivered, it processes the list iteratively until it produces a list of resolved addresses. When an address is resolved, **smail** will know which transport it must use to deliver the message to the person or persons to whom it is addressed, and all data that this transport requires. To accomplish this, smail goes through the following steps:

A. **smail** first parses each address to find a host name, called the *target*, and the remaining part of the address, called the *remainder*.

As a simple example, in the address **tron@uts.amdahl.com**, the host part **uts.amdahl.com** is the target and **tron** is the remainder. Likewise, in the address **sun!amdahl!tron**, the target is **sun** and the remainder is **amdahl!tron**.

B. **smail** then shows each address to directors, in the order given in file **/usr/lib/mail/directors**, until one of the directors says that it knows what to do with that address. That director can either return a new list of addresses, or put the address onto a list of resolved addresses. If new addresses are produced, **smail** places them onto the input list, to be processed from step **A**.

C. When an address has passed through step **B**, **smail** shows it to various routers, in the order given in file **/usr/lib/mail/routers**, until a router can match the target name for the address. If no router can match the complete target, then **smail** selects the router that matches the longest portion of the target. The router names the transport to be used to deliver the message to that address, plus some other information that the transport requires (e.g., the next host and next address values). The information as to which transport to use can come either from the definition of the router, from a method file, or may be specified by the router specifically.

When all addresses have been resolved, **smail** sorts them and passes them to transports. The transport then delivers the message to the addresses it is given.

Host names and local user-names are matched independent of case; for example, “Postmaster”, “POSTMASTER”, and “postmaster” are in effect all the same. In addition, **smail** keeps an internal hash table of all regular recipient addresses, that is, all addresses that do not specify files or shell commands. This table is used to discard duplicate regular recipient addresses. This hash table is independent of case, as well, so that the address **Postmaster@SRI-NIC.ARPA** is considered a duplicate of **postmaster@sri-nic.arpa**.

Other Files and Directories

smail also uses the following configuration files:

LEXICON

/usr/lib/mail/qualify
Configuration file for host-name qualification.
/usr/lib/mail/retry
Optional delivery retry configuration file, i.e., minimum time between retries and maximum time to retry before giving up.

smail reads the following files to learn how to redirect mail locally and to give paths to remote sites:

/usr/lib/mail/aliases
Aliases for mail addresses.
/usr/lib/mail/paths
Paths to remote hosts.
/usr/lib/mail/lists
Mailing-list files.
/usr/spool/mail
The directory that holds each user's mailbox file.
$HOME/.forward
Forwarding address or addresses for a given user.

smail uses the following directories to hold incoming mail messages and its work files:

/usr/spool/smail
Directory that holds spool directories and work files.
/usr/spool/smail/input
Spool directory for incoming messages.
/usr/spool/smail/error
Directory that holds mail messages that failed for a reason that the site administrator should investigate.
/usr/spool/smail/msglog
Directory that holds transaction logs for individual messages.
/usr/spool/smail/lock
This directory holds **smail**'s input lock files.

The following files log **smail**'s activity. Please note that these files will grow without end. Your system's system administrator should check and truncate these files from time to time. She can also use the script **/usr/lib/mail/savelog** to manage these files; for details, see the Lexicon entry for this command:

/usr/spool/smail/log/logfile
A log of **smail**'s transactions.
/usr/spool/smail/log/paniclog
A log of configuration or system errors encountered by **smail**.

See Also

commands, mail [command], mail [overview], mailq, rmail, rsmtp, runq, savelog, smtpd

Diagnostics

If all goes well, **smail** returns zero to the shell when it exits. If an error occurs, it returns a value other than zero. The meaning of each code is set in **smail**'s source file **exitcodes.h**, as follows:

EX_USAGE. . . . . Error in command-line usage
EX_DATAERR. . . . . Data-format error
EX_NOINPUT . . . . . Cannot open input file
EX_NOUSER. . . . . Addressee unknown
EX_NOHOST . . . . . Host unknown
EX_UNAVAILABLE. . . Service unavailable
EX_SOFTWARE. . . . . Internal software error
EX_OSERR . . . . . System error
EX_OSFILE . . . . . Critical OS file missing
EX_CANTCREAT . . . . Cannot create (user) output file
EX_IOERR. . . . . Error in file I/O
EX_TEMPFAIL . . . . . Temporary failure; user can retry
EX_PROTOCOL. . . . . Remote error in protocol
EX_NOPERM . . . . . Permission denied

If you invoke **smail** with its option **-bd**, then the message

```
bind() failed: address already in use
```

means that another process is already listening to the SMTP socket.

Notes

Many mail bugs are not **smail** bugs. **smail** cannot help it if remote sites trash your mail messages.

Setting the input spool directory processing interval to a period of more than 2,147,483,647 seconds will result in an incorrectly calculated processing interval — and is a rather silly thing to do at any event.

smtpd — Command

SMTP daemon
/bin/smtpd

The daemon **smtpd** reads SMTP commands from standard input, and writes SMTP replies onto the standard output. The following SMTP commands are implemented:

HELO	MAIL	FROM	RCPT
TO	DATA	RSET	NOOP
VRFY	EXPN	QUIT	DEBUG

See Also

commands, mail [overview], rsmtp, smail

Notes

smtpd is a link to command **smail**.

smult() — Multiple-Precision Mathematics (libmp)

Multiply multiple-precision integers
#include <mprec.h>
void smult(*a, n, c***)**
mint **a*, *c*; **int** *n*;**

smult() multiplies the multiple-precision integer (or **mint**) pointed to by *a* by the integer *n*, which is <= 127. It writes the product into the **mint** pointed to by *c*.

See Also

libmp

SOCKADDRLEN() — Sockets Function (libsocket)

Return length of an address
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/un.h>
int SOCKADDRLEN(*address***)**
struct sockaddr **address*;**

Function **SOCKADDRLEN()** returns the size, in bytes, of *address***->sa_family**. This helps a program distinguish between a UNIX and an Internet address.

See Also

libsocket

Notes

COHERENT implements **SOCKADDRLEN()** as a function rather than as a macro.

socket() — Sockets Function (libsocket)

Create a socket
#include <sys/types.h>
#include <sys/socket.h>
int socket(*domain***,** *type***,** *protocol***)**
int *domain***,** *type***,** *protocol***;**

socket() creates a "socket" — that is, an endpoint for communication. It returns a descriptor that uniquely identifies the socket.

domain specifies the domain within which communication will take place. This selects the protocol family to be used. These families are defined in **<sys/socket.h>** Currently, **socket()** recognizes the following domains:

AF_UNIX UNIX internal protocols.
AF_INET ARPA Internet protocols.

The socket has the indicated *type*, which specifies the semantics of communication. **socket()** recogizes the following types:

SOCK_STREAM
This type provides a byte stream that is sequenced, reliable, two-way, and connection-based.

SOCK_DGRAM This type supports "datagrams" — that is, connectionless, unreliable messages of a fixed maximum length.

protocol identifies the protocol to be used with the newly created socket. In most instances, a given type of socket supports only one protocol. However, a socket type may support many different protocols, in which case you must specify the one to use. The protocol number to use is particular to the "communication domain" in which communication is to take place.

Sockets of type **SOCK_STREAM** are full-duplex byte streams, similar to pipes. A stream socket must be in a connected to another socket (through a call to function **connect()**) before any data can be sent to it or received on it. Once connected, data can be transferred using the system calls **read()** and **write()**. When a session has been completed, invoke the system call **close()** to close the socket.

If all goes well, **socket()** returns the descriptor of the newly created socket; this is always a positive integer. If something goes wrong, it returns -1 and sets **errno** to an appropriate value. The following lists the possible errors, by the value to which **socket()** sets **errno**:

EPROTONOSUPPORT
type or *protocol* is not supported within this domain.

EMFILE
The per-process descriptor table is full.

ENFILE
The system file table is full.

EACCESS
You do not have permission to create a socket of a given *type* or *protocol*.

ENOBUFS
Not enough buffer space is available. The socket cannot be created until sufficient resources are freed.

See Also

accept(), connect(), libsocket, listen(), read(), write()

socket.h — Header File

Define constants and structures with sockets
#include <sys/socket.h>

Header file **<socket.h>** defines constants, structures, and prototypes used with **sockets**.

See Also

header files, libsocket

socketpair() — Sockets Function (libsocket)

Create a pair of sockets
int socketpair (*family***,** *type***,** *protocol***,** *fds***)**
int *family***,** *type***,** *protocol***,** *fds***[2];**

Function **socketpair()** creates a pair of sockets. *family*, *type*, and *protocol* give the family, type, and protocol of the sockets to be created. At present, *family* must be set to **AF_UNIX**. *fds* gives the address of an array of two integers, into which **socketpair()** writes the file descriptors of the sockets it creates.

If all goes well, **socketpair()** returns zero. If an error occurs, it returns -1 and sets **errno** to an appropriate value.

See Also

libsocket

Notes

socketpair() does not connect the pair of sockets that it creates, so a call to **getpeername()** on one of them will not return the name of the other.

sort — Command

Sort lines of text
sort [-bcdfimnru] [-t *c***] [-o** *outfile***] [-T** *dir***] [+***beg***[-***end***]][***file ...***]**

sort reads lines from each *file*, or from the standard input if no file is specified. It sorts what it reads, and writes the sorted material to the standard output.

sort sorts lines by comparing a *key* from each line. By default, the key is the entire input line (or *record*) and ordering is in ASCII order. The key, however, can be one or more *fields* within the input record; by using the appropriate options, you can select which fields are used as the key, and dictate the character that is used to separate the fields.

The following options affect how the key is constructed or how the output is ordered.

-b Ignore leading white space (blanks or tabs) in key comparisons.

-d Dictionary ordering: use only letters, blanks, and digits when comparing keys. This is essentially the ordering used to sort telephone directories.

-f Fold upper-case letters to lower case for comparison purposes.

-i Ignore all characters outside of the printable ASCII range (octal 040-0176).

-n The key is a numeric string that consists of optional leading blanks and optional minus sign followed by any number of digits with an optional decimal point. Ordering is by the numeric, as opposed to alphabetic, value of the string.

-r Reverse the ordering, i.e., **sort** from largest to smallest.

As noted above, the key compared from each line need not be the entire input line. The option **+***beg* indicates the beginning position of the key field in the input line, and the optional **-***end* indicates that the key field ends just before the *end* position. If no **-***end* is given, the key field ends at the end of the line. Each of these positional indicators has the form **+***m.nf* or **-***m.nf*, where *m* is the number of fields to skip in the input line and *n* is the number of characters to skip after skipping fields. Optional flags *f* are chosen from the above key flags (**bdfinr**) and are local to the specified field.

The following additional options control how **sort** works.

-c Check the input to see if it is sorted. Print the first out-of-order line found.

-m Merge the input files. **sort** assumes each *file* to be sorted already. With large files, **sort** runs much faster with this option.

-o *outfile*
Put the output into *outfile* rather than on the standard output. This allows **sort** to work correctly if the output file is one of the input files.

-t*c* Use the character *c* to separate fields rather than the default blanks and tabs. For example, **-t/** uses the slash instead of white space to separate fields; this is useful when sorting file names and directory names.

-T *dir*
Create temporary files in directory *dir* rather than the standard place.

-u Suppress multiple copies of lines with key fields that compare equally.

The following example sorts the password file **/etc/passwd,** first by group number (field 4) and then by user name (field 1):

```
sort -t: +3n -4 +0 -1 /etc/passwd
```

Limits

The COHERENT implementation of **sort** sets the following limits on input and output:

Characters per input record	399
Characters per output record	399
Characters per field	399

Files

/usr/tmp/sort* — First attempt at temporary files
/tmp/sort* — Second attempt at temporary files

See Also

ASCII, commands, ctype.h, qsort(), shellsort(), tsort, uniq

Diagnostics

sort returns a nonzero exit status if internal problems occurred, or if the file was not correctly sorted in the case of the **-c** option.

spac — Command

Sort a file system
spac *raw_device*

The command **spac** uses the default **dpac** sorting algorithm to re-organize file system *raw_device*.

See Also

commands, dpac, fmap, fsck, qpac, upac

Notes

spac is a link to the command **dpac**. **spac** was written by Randy Wright (rw@rwsys.wimsey.bc.ca).

spell — Command

Find spelling errors
spell [-a][-b][*file ...***]**

spell builds a set of unique words from a document contained in each input *file*, or the standard input if none. It writes a list of words believed to be misspelled onto the standard output.

spell should normally be invoked with the document in the form of the input to the text formatter **nroff** rather than the output. **spell** deletes control information to the formatter by invoking **deroff**.

The default dictionary is for American spelling of English. The **-a** option specifies this dictionary explicitly. Under the **-b** option, British spelling is checked. This accepts *favour*, *fibre*, and *travelled* rather than the American spellings *favor*, *fiber*, and *traveled* for the same words. Words ending in *ize* are also accepted when ending in *ise* (e.g., *digitize*, *digitise*).

The dictionary has a reasonably complete coverage of proper names as well as technical terms in certain fields. However, it covers some fields (e.g., computer science) better than others (e.g., medicine).

Looking up a Word

The COHERENT command **look** reads **spell**'s dictionaries to find words that resemble a fraction of a word that you type. For example, the command

```
look consider
```

returns the following to the standard output:

```
consider#
considerable
considerably
considerate
considerately
consideration#
considered
considering
```

The '#' indicates a possible plural form by adding 's' to the end of the word. This lets you check the spelling of a word without having to enter the word into a file and run **spell** on it.

Files

/usr/dict/clista — Compressed American dictionary
/usr/dict/clistb — Compressed British dictionary
/usr/dict/spellhist — History file for dictionary maintainer
/usr/lib/spell

See Also

commands, deroff, look, nroff, sort, typo

Notes

Dictionaries are not provided for languages other than English.

No dictionary can be complete. You must add new words to the dictionary to ensure that it fully meets your needs.

Obscure words (such as opcodes, variable names, etc.) are flagged as spelling errors.

Because the data files required for **spell** are quite large, they might not be installed onto systems with limited disk space. As a result, the command might not work as expected on all systems.

split — Command

Split a text file into smaller files
split [-*lines***][-c***count***][***infile* **[***outfile***]]**

split divides a file into a number of smaller files. This is especially useful for dividing text files into chunks that can be managed by MicroEMACS or similar editors, or for dividing binary files into chunks that can be easily transmitted via UUCP.

split uses *infile* as its input file if given; otherwise, it uses the standard input. If *infile* is '-', **split** uses the standard input.

split puts its output into files with names prefixed by *outfile* and suffixed consecutively with **aa**, **ab**, **ac**, and so on. If no *outfile* is specified, file names are prefixed with **x**.

Normally, **split** puts 1,000 lines in each output file. This default may be changed for text files by the option *-lines*, where *lines* gives the desired number of lines per file. When using **split** on binary files, the argument *count* to the **-c** option lets you specify the number of characters to place into each output file.

See Also

commands

spow() — Multiple-Precision Mathematics (libmp)

Raise multiple-precision integer to power
#include <mprec.h>
void spow(*a, n, b***)**
mint **a, *b; int** *n***;**

spow() raises the multiple-precision integer (or **mint**) pointed to by *a* to the power of *n*, and writes the result into the **mint** pointed to by *b*. In no case may the exponent be negative.

See Also

libmp

sprintf() — STDIO Function (libc)

Format output
#include <stdio.h>
int sprintf(*string***,** *format* **[,** *arg* **] ...)**
char **string***, ***format***;**

sprintf() formats and prints a string. It resembles the function **printf()**, except that it writes its output into the memory location pointed to by *string*, instead of to the standard output.

sprintf() reads the string pointed to by *format* to specify an output format for each *arg*; it then writes every *arg* into *string*, which it ends with a null character. For a detailed discussion of **sprintf()**'s formatting codes, see **printf()**.

If it wrote the formatted string correctly, **sprintf()** returns the number of characters written. Otherwise, it returns a negative number.

Example

For an example of this function, see the entry for **sscanf()**.

See Also

printf(), fprintf(), libc, vsprintf()
ANSI Standard, §7.9.6.5
POSIX Standard, §8.1

Notes

The output *string* passed to **sprintf()** must be large enough to hold all output characters.

Because C does not perform type checking, it is essential that each argument match its format specification.

sqrt() — Mathematics Function (libm)

Compute square root
#include <math.h>
double sqrt(*z***) double** *z***;**

sqrt() returns the square root of *z*.

Example

The following program prints all prime numbers between one and a positive integer that the user enters. It was written by Michael B. Young (myoung@mcs.csuhayward.edu).

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
      int i, userinput;

      /* get user input */
      fprintf(stderr, "Enter an integer value greater than 2:  ");
      scanf("%d", &userinput);
```

```
      if (userinput < 3) {
            fprintf(stderr, "Error:  enter a positive integer > 2\n");
            exit(EXIT_FAILURE);
      }

      /* test for all numbers between one and "userinput". */
      /* for efficiency's sake, even numbers are not tested. */
      /* two is the only even prime number */

      printf("%d\n", 2);
      for (i = 3; i < userinput; i += 2)
            if (prime(i))
                  printf("%d\n", i);

      exit(EXIT_SUCCESS);
}

/*
 *  function prime() - tests the passed integer testvalue for "prime-ness"
 *  by testing whether each integer between 1 and the square root of
 *  testvalue divides evenly into testvalue.  Returns 1 if prime, 0 if not.
 */
int prime(testvalue)
int testvalue;
{
      int end, j, result;

      end = (int) sqrt ( (double) testvalue );
      for (j = 2, result = 1; result == 1 && j <= end; j++) {
            if ((testvalue % j) == 0)
                  result = 0;
      }
      return result;
}
```

See Also

cos(), cosh(), libm, sin()
ANSI Standard, §7.5.5.2
POSIX Standard, §8.1

Diagnostics

When a domain error occurs (i.e., when *z* is negative), **sqrt()** sets **errno** to **EDOM** and returns zero.

srand() — Random-Number Function (libc)

Seed random number generator
#include <stdlib.h>
void srand(*seed***) int** *seed***;**

srand() uses *seed* to initialize the sequence of pseudo-random numbers returned by **rand()**. Different values of *seed* initialize different sequences.

Example

For an example of this function, see the entry for **rand()**.

See Also

libc, rand(), stdlib.h
The Art of Computer Programming, vol. 2
ANSI Standard, §7.10.2.2
POSIX Standard, §8.1

Notes

For a superior but non-standard random-number generator, see the function **randl()**, described in the Lexicon article **libmisc**.

srand() cannot be used with any of the "rand48" functions. For an overview of these functions, see the entry for **srand48()**.

srand48() — Random-Number Function (libc)

Seed the 48-bit pseudo-random number routines
void srand48(*seedval***)**
long seedval;

Computation of 48-bit pseudo-random numbers uses two 48-bit integers and one 16-bit integer. One of the 48-bit values holds the "seed" value from which the 48-bit pseudo-random value is computed. This seed can be set explicitly, or is the previously computed pseudo-random number. The other 48-bit integer holds the multiplier from which the pseudo-random number is computed; and the 16-bit integer gives holds the addend.

Function **srand48()** builds the 48-bit "seed" value from a long integer. The 32 bits of the long integer comprise the high 32 bits of the seed; the low 16 bits are filled with the value 0x33E.

Functions **lcong48()** and **seed48()** can also be used to seed the routines that generate 48-bit pseudo-random numbers. **srand48()** returns nothing.

See Also

drand48(), erand48(), jrand48(), libc, lcong48(), lrand48(), mrand48(), nrand48(), seed48()

srandom() — Sockets Function (libsocket)

Seed the random-number generator
int srandom(*seed***)**
int *seed***;**

The function **srandom()** "seeds" the random-number generator with value *seed*. It is a synonym for **srand()**.

srandom() does not return a meaningful value.

See Also

libsocket, srand()

srcpath — Command

Find source files
srcpath [-aw] [-p *path***]** *filename pattern* **...**

The command **srcpath** expands the environmental variable **SRCPATH**, applies it to each argument, and prints the full path of each unique result.

An argument can either be a file name or a pattern. For example, the command

```
srcpath "*.[ch]"
```

finds all **.c** and **.h** files on **SRCPATH**. By default. **srcpath** keeps only the first file that it finds with a given name. **srcpath** automatically appends '.' to the beginning of **SRCPATH** so files in the current directory have precedence.

srcpath recognizes the following command-line options:

-p *path*
Use *path* as its path instead of **SRCPATH**. For example,

```
srcpath -p ".:/usr/src/cmd" "*.c"
```

tells **srcpath** to search '.' and **/usr/src/cmd** instead of **SRCPATH**. Note that with this option, **srcpath** does not automatically place '.' at the beginning of the list.

-a Disable shadowing. Normally, if **srcpath** finds a file is found in more than one directory on the path, it prints only the first. The **-a** option forces **srcpath** to print all instances of the file name.

-w By default, **srcpath** silently bypasses directories and matching files for which it has no read permission. The **-w** option causes it to print a warning message when this happens.

See Also

commands, find, make, PATH

sscanf() — STDIO Function (libc)

Format a string
#include <stdio.h>
int sscanf(*string***,** *format* **[,** *arg* **] ...)**
char ******string***; char** ******format***;**

sscanf() reads the argument *string*, and uses *format* to specify a format for each *arg*, each of which must be a pointer. For more information on **sscanf()**'s conversion codes, see **scanf()**.

Example

This example uses **sprintf()** to create a string, and then reads it with **sscanf()**. It also illustrates a common problem with this routine.

```
#include <stdio.h>

main()
{
      char string[80];
      char s1[10], s2[10];

      sprintf(string, "12345678901234567890123456789O");
      sscanf(string, "%9c", s1);
      sscanf(string, "%10c", s2);

      printf("\n%s is the string\n", string);
      printf("%s: first 9 characters in string\n", s1);
      printf("%s: first 19 characters in string\n", s2);
}
```

See Also

fscanf(), libc, scanf()
ANSI Standard, §7.9.6.6
POSIX Standard, §8.1

Diagnostics

sscanf() returns the number of arguments filled. It returns zero if no arguments can be filled or if an error occurs.

Notes

Because C does not perform type checking, an argument must match its format specification. **sscanf()** is best used only to process data that you are certain are in the correct data format, such as data that were written with **sprintf()**.

sscanf() is difficult to use correctly, and incorrect usage can create serious bugs in programs. It is recommended that you use **strtok()** instead.

stack — Definition

The **stack** is the segment of memory that holds function arguments, local variables, function return addresses, and stack frame linkage information.

If your program uses recursive algorithms, or declares large amounts of automatic data, or simply contains many levels of functions calls, the stack may "overflow", and overwrite the program data. Note that this is unlikely with COHERENT, because the 80386 has implemented dynamic stack allocation.

See Also

Programming COHERENT

standard error — Definition

The **standard error** is the peripheral device or file where programs write error messages by default. It is defined in the header file **stdio.h** under the abbreviation **stderr**, and by default is the computer's monitor.

The shell lets you redirect into a file all text written to the standard error device. To do so, use the shell operator **2>**. For example

```
make 2>errorfile
```

redirects all error messages generated by **make** into file **errorfile**.

See Also

Programming COHERENT, stderr, stdio.h

standard input — Definition

The **standard input** is the device or file from which data are accepted by default. It is defined in the header file **stdio.h** under the abbreviation **stdin**, and will be the computer's keyboard unless redirected by the operating system, a shell, or **freopen**.

The shell lets you redirect the standard input device. To do so, use the shell operator **<**. For example

```
mail fwb <textfile
```

the standard input device from your terminal to file **textfile**; in effect, this commands mails the contents of **textfile** to user **fwb**.

See Also

Programming COHERENT, stdin, stdio.h

standard output — Definition

The **standard output** is the device or file where programs write output by default. It is defined in the header file **stdio.h** under the abbreviation **stdout**, and in most instances is defined to be the computer's monitor.

The shell lets you redirect into a file all text written to the standard output device. To do so, use the shell operator **>**. For example

```
sort myfile >sortfile
```

redirects the text output by **sort** into file **sortfile**.

See Also

Programming COHERENT, stdio.h, stdout

stat() — System Call

Find file attributes
#include <sys/stat.h>
int stat(*file, statptr***)**
char ******file***; struct stat** ******statptr***;**

stat() returns a structure that contains the attributes of a file, including protection information, file type, and file size.

file points to the path name of file. *statptr* points to a structure of the type **stat**, as defined in the header file **stat.h**. For information on **stat**, see the Lexicon entry for **stat.h**.

Example

The following example uses **stat()** to print a file's status.

```
#include <sys/stat.h>
main()
{
      struct stat sbuf;
      int status;

      if (status = stat("/usr/include", &sbuf)) {
            printf("Can't find\n");
            exit(EXIT_FAILURE);
      }

      printf("uid = %d gid = %d\n", sbuf.st_uid, sbuf.st_gid);
}
```

See Also

chmod(), chown(), libc, ls, open(), stat.h
POSIX Standard, §5.6.2

Diagnostics

stat() returns -1 if an error occurs, e.g., the file cannot be found. Otherwise, it returns zero.

Notes

stat() differs from the related function **fstat()** mainly in that **fstat()** accesses the file through its descriptor, which was returned by a successful call to **open()**, whereas **stat()** takes the file's path name and opens it before checking its status.

The call

```
stat("", &s)
```

is identical to

```
stat(".", &s)
```

Both calls succeed. The POSIX Standard forbids the former call — in fact, the POSIX Standard forbids the NULL string as a path name under any circumstances; therefore you should never use the former call.

stat.h — Header File

Definitions and declarations used to obtain file status
#include <sys/stat.h>

stat.h is a header file that declares the structure **stat** plus constants used by the routines that manipulate files, directories, and named pipes. It holds the prototypes for the routines **chmod()**, **fstat()**, **mkdir()**, **stat()**, and **umask()**.

The following summarizes the structure **stat**:

```
struct stat {
        dev_t                st_dev;             /* Device */
        ino_t                st_ino;             /* Inode number */
        mode_t               st_mode;            /* Mode */
        nlink_t              st_nlink;           /* Link count */
        uid_t                st_uid;             /* User id */
        gid_t                st_gid;             /* Group id */
        dev_t                st_rdev;            /* Real device; NB, this is non-POSIX */
        off_t                st_size;            /* Size */
        time_t               st_atime;           /* Access time */
        time_t               st_mtime;           /* Modify time */
        time_t               st_ctime;           /* Change time */
};
```

st_dev and **st_ino** together form a unique description of the file. The former is the device on which the file and its i-node reside, whereas the latter is the index number of the file. **st_mode** gives the permission bits, as outlined below. **st_nlink** gives the number of links to the file. **st_uid** and **st_gid,** respectively given the user id and group id of the owner. **st_rdev**, valid only for special files, holds the major and minor numbers for the file. **st_size** gives the size of the file, in bytes. For a pipe, the size is the number of bytes waiting to be read from the pipe.

Three entries for each file give the last occurrences of various events in the file's history. **st_atime** gives time the file was last read or written to. **st_mtime** gives the time of the last modification, write for files, create or delete entry for directories. **st_ctime** gives the last change to the attributes, not including times and size.

The following manifest constants define file types:

S_IFMT	Type
S_IFDIR	Directory
S_IFCHR	Character-special file
S_IFPIP	Pipe
S_IFIFO	Pipe
S_IFBLK	Block-special file
S_IFREG	Regular file

The following manifest constants define file modes:

S_IREAD	Read permission, owner
S_IWRITE	Write permission, owner
S_IEXEC	Execute/search permission, owner
S_IRWXU	RWX permission, owner
S_IRUSR	Read permission, owner
S_IWUSR	Write permission, owner
S_IXUSR	Execute/search permission, owner
S_IRWXG	RWX permission, group
S_IRGRP	Read permission, group
S_IWGRP	Write permission, group
S_IXGRP	Execute/search permission, group
S_IRWXO	RWX permission, other
S_IROTH	Read permission, other
S_IWOTH	Write permission, other
S_IXOTH	Execute/search permission, other

See Also

chmod(), fstat(), header file, stat()
POSIX Standard, §5.6.1

statfs() — System Call (libc)

Get information about a file system
#include <sys/types.h>
#include <sys/statfs.h>
int statfs (*path***,** *buffer***,** *length***,** *fstype***)**
char ******path***;**
struct statfs ******buffer***;**
int *length***,** *fstype***;**

The COHERENT system call **statfs()** returns information about a file system, either mounted or unmounted.

buffer points to a structure of type **statfs**, which contains the following members:

```
short          f_fstyp;              /* type of the file system */
short          f_bsize;              /* block size */
short          f_frsize;             /* fragment size */
long           f_blocks;             /* number of blocks in the file system */
long           f_bfree;              /* number of free blocks */
long           f_files;              /* number of file nodes */
long           f_ffree;              /* number of free file nodes */
char           f_fname[6];           /* name of the volume */
char           f_fpack[6];           /* name of the pack */
```

length is the length of the area into which **statfs()** can write its output. This should always be set to **sizeof(struct statfs)**.

path and *fstype* identify the file system. If the file system is unmounted, then *path* should name the device by which the file system is accessed, and *fstype* should contain the type of the file system. If the file system is mounted, then *path* should give the full path name of a file on the file system in question, and *fstype* must be set to zero.

statfs() returns zero if all went well. If something went wrong, it returns -1 and sets **errno** to an appropriate value.

See Also

fstatfs(), libc, mkfs, statfs.h, ustat()

static — C Keyword

Declare storage class

static is a C storage class. It has two entirely different meanings, depending upon whether it appears inside or outside a function.

Outside a function, **static** means that the function or variable it preceeds may not be seen outside the module.

Inside a function, static may only precede a variable. It means that that variable is permanently allocated, rather

than allocated on the stack when the function is entered and discarded when the function exits. If a **static** variable is initialized, that occurs before the program starts rather than every time the function is entered. If a function returns a pointer to a variable, often that variable is declared **static** within the function. If a pointer to a **non-static** local variable is returned, that variable is freed when the function returns and the pointer points to an unprotected location.

Example

The following example demonstrates the uses of the **static** keyword. It returns the next integer in a sequence as a string.

```
/* static to keep function hidden outside of this module */
static char *nextInt()
{
      /* static to protect value between calls */
      static int next = 0;
      /* static to allow the return of a pointer to s */
      static char s[5];

      sprintf(s, "%d", next++);
      return(s);
}
```

See Also

auto, C keywords, extern, register variable, storage class
ANSI Standard, §6.5.1

stdarg.h — Header File

Header for variable numbers of arguments
#include <stdarg.h>

stdarg.h is the header file that ANSI C uses to declare and define the routines that traverse a variable-length argument list. It declares the type **va_list** and defines the macros **va_arg()**, **va_start()**, and **va_end()**.

Example

The following example concatenates multiple strings into a common allocated string and returns the string's address. method of handling variable arguments:

```
#include <stdarg.h>
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>

char *
multcat(numargs)
int numargs;
{
      va_list argptr;
      char *result;
      int i, siz;

      /* get size required */
      va_start(argptr, numargs);
      for(siz = i = 0; i < numargs; i++)
            siz += strlen(va_arg(argptr, char *));

      if ((result = calloc(siz + 1, 1)) == NULL) {
            fprintf(stderr, "Out of space\n");
            exit(EXIT_FAILURE);
      }
      va_end(argptr);

      va_start(argptr, numargs);
      for(i = 0; i < numargs; i++)
            strcat(result, va_arg(argptr, char *));
      va_end(argptr);
      return(result);
}
```

```
int
main()
{
      printf(multcat(5, "One ", "two ", "three ",
            "testing", ".\n"));
}
```

See Also

header files, varargs.h
ANSI Standard, §7.8

Notes

The routines defined in **<stdarg.h>** were first implemented under UNIX System V, where they are declared in the header file **<varargs.h>**. The ANSI C committee recognized the usefulness of **<varargs.h>**, but decided that it had semantic problems. In particular, **<varargs.h>** introduced the notion of declaring "..." for the variable-arguments argument list in the function prototype. This, unfortunately, left them with declarations of the form

```
void error(...)
{
      whatever
}
```

and no obvious hook for accessing the parameter list within the body of the function. So, the ANSI committee changed the header declaration: it insisted on one or more formal parameters, followed by the list of variables.

The committee had the wisdom to change the name of its header file, hence **<stdarg.h>** came into being. Unfortunately, the committee kept the same macro names, but in one macro (**va_start()**) changed the number of arguments it takes.

COHERENT includes both **<varargs.h>** and **<stdarg.h>**, to support both ANSI and System-V code.

stddef.h — Header File

Header for standard definitions
#include <stddef.h>

stddef.h defines types and macros that are used through the library.

See Also

header files, offsetof()
ANSI Standard, §7.1.6

stderr — Definition

stderr is the name of the **FILE** pointer assigned to the standard error device. It is set in the header file **stdio.h**.

See Also

Programming COHERENT, stdin, stdio.h, stdout, standard error
ANSI Standard, §4.9.1, §4.9.3

stdin — Definition

stdin is the name of the **FILE** pointer that is assigned to the standard input device. It is set in the header file **stdio.h**.

See Also

Programming COHERENT, standard input, stderr, stdio.h, stdout
ANSI Standard, §7.9.1

STDIO — Definition

STDIO is an abbreviation for *standard input and output*. It refers to a set of standard library functions that accompany all C compilers and that govern input and output with peripheral devices. For details on the STDIO routines, see the Lexicon entries for **libc** and **stdio.h**.

See Also

libc, Programming COHERENT, stdio.h
ANSI Standard, §4.9

stdio.h — Header File

Declarations and definitions for I/O

stdio.h is a header file that defines manifest constants used in standard I/O, prototypes the STDIO functions, and defines numerous I/O macros, as follows:

Types

FILE . . . . . Descriptor of file used by STDIO routines
stderr . . . . . Standard error device (by default, the screen)
stdin . . . . . Standard input device (by default, the keyboard)
stdout . . . . . Standard output device (by default, the screen)

Manifest Constants

BUFSIZ . . . . . Default buffer size
EOF . . . . . End of file
FILENAME_MAX . . . . Maximum length of a file name
FOPEN_MAX . . . . . Maximum number of of open files
L_ctermid . . . . . Length of **ctermid()**
L_tmpnam . . . . . Length of a temporary file name
P_tmpdir . . . . . Default directory for temporary files
TMP_MAX . . . . . Maximum number of temporary file names

Functions and Macros

clearerr() . . . . . Present status stream
fclose() . . . . . Close a file stream
fdopen() . . . . . Open a file stream for I/O
feof() . . . . . Discover a file stream's status
ferror() . . . . . Discover a file stream's status
fflush() . . . . . Flush an output buffer
fgetc() . . . . . Get a character
fgetpos() . . . . . Read the file-position indicator
fgets() . . . . . Get a string
fgetw() . . . . . Get a word
fileno() . . . . . Get a file descriptor from a **FILE** structure
fopen() . . . . . Open a file stream
fprintf() . . . . . Format and print to a file stream
fputc() . . . . . Output a character
fputs() . . . . . Output a string
fputw() . . . . . Output a word
fread() . . . . . Read a file stream
freopen() . . . . . Open a file stream
fscanf() . . . . . Format and read from a file stream
fseek() . . . . . Seek in a file stream
fsetpos() . . . . . Set the file-position indicator
ftell() . . . . . Return file pointer position
fwrite() . . . . . Write to a file stream
getc() . . . . . Get a character
getchar() . . . . . Get a character
gets() . . . . . Get a string
getw() . . . . . Get a word
pclose() . . . . . Close a pipe
popen() . . . . . Open a pipe
printf() . . . . . Print a formatted string
putc() . . . . . Output a character
putchar() . . . . . Output a character
puts() . . . . . Output a string
putw() . . . . . Output a word
rewind() . . . . . Reset a file pointer
scanf() . . . . . Format and input from standard input

setbuf() . . . . . Set alternative file-stream buffer
setvbuf() . . . . . Set alternative file-stream buffer
sprintf() . . . . . Format and print to a string
sscanf() . . . . . Format and read from a string
tmpfile() . . . . . Create a temporary file
ungetc() . . . . . Return character to file stream
vfprintf() . . . . . Format and print to a file stream
vprintf() . . . . . Print a formatted string
vsprintf() . . . . . Format and print to a string

See Also

header file, libc, STDIO
ANSI Standard, §7.9

Notes

COHERENT release 4.2 has rewritten its version of **stdio.h** so that it conforms to the ANSI Standard. For this reason, program that use STDIO and are compiled under COHERENT release 4.2 (or subsequent releases) will not run correctly under versions of COHERENT prior to release 4.2.

stdlib.h — Header File

Declare/define general functions
#include <stdlib.h>

stdlib.h is a header file that is defined in the ANSI Standard. It declares a set of general utilities and defines attending macros and data types, as follows.

Types

div_t . . . . . Type of object returned by **div**
ldiv_t . . . . . Type of object returned by **ldiv**

Manifest Constants

EXIT_FAILURE . . . . . Value to indicate that program failed to execute properly
EXIT_SUCCESS. . . . . Value to indicate that program executed properly
MB_CUR_MAX . . . . . Largest size of multibyte character in current locale
MB_LEN_MAX. . . . . Largest overall size of multibyte character in any locale
RAND_MAX . . . . . Largest size of pseudo-random number

Functions

abort() . . . . . End program immediately
abs(). . . . . Compute the absolute value of an integer
atof() . . . . . Convert string to floating-point number
atoi() . . . . . Convert string to integer
atol() . . . . . Convert string to long integer
bsearch(). . . . . Search an array
calloc() . . . . . Allocate dynamic memory
div(). . . . . Perform integer division
exit() . . . . . Terminate a program gracefully
free() . . . . . De-allocate dynamic memory to free memory pool
getenv() . . . . . Read environmental variable
labs() . . . . . Compute the absolute value of a long integer
ldiv() . . . . . Perform long integer division
malloc() . . . . . Allocate dynamic memory
qsort() . . . . . Sort an array
rand(). . . . . Generate pseudo-random numbers
realloc() . . . . . Reallocate dynamic memory
srand() . . . . . Seed the random-number generator
strtod(). . . . . Convert string to floating-point number
strtol() . . . . . Convert string to long integer
strtoul() . . . . . Convert string to unsigned long integer
system() . . . . . Suspend a program and execute another

See Also

header files
ANSI Standard, §7.10

stdout — Definition

stdout is the name of the **FILE** pointer that is assigned to the standard output device. It is set in the header file **stdio.h**.

See Also

Programming COHERENT, standard output, stderr, stdin, stdio.h
ANSI Standard, §7.9.1

sticky bit — Definition

The *sticky bit* is one of the mode bits associated with a file. If the sticky bit is set for an executable file and swapping is enabled, COHERENT behaves in a special way when it executes that file.

When the COHERENT system executes the file the first time, all proceeds normally. When the program exits, however, the pure segments are left on the swap device; when the program is re-invoked, COHERENT reads "pure" code (text) areas from the swap device and all other (impure) segments from the file system. This speeds execution of large programs that are executed frequently.

This strategy works well on systems that have large swap devices. Because overuse of the sticky bit would quickly swamp the swap device, only the superuser can set the sticky bit.

See Also

chmod, Using COHERENT

stime() — System Call (libc)

Set the time
#include
int stime(*timep***)**
time_t **timep***;*

stime() sets the system time. *timep* points to a variable of type **time_t**, which contains the number of seconds since midnight GMT of January 1, 1970.

If all goes well, **stime()** zero. If a problem occurs, it returns -1.

stime() is restricted to the superuser.

Files

<sys/types.h>

See Also

ctime(), date, ftime(), libc, stat(), utime()

storage class — Definition

Storage class refers to the part of a declaration that indicates how data are to be stored. The C language recognizes the following storage clases:

auto
extern
register
static

typedef is technically defined as a storage class as well, but it does not actually indicate how data are stored. The default class is **auto**.

See Also

auto, extern, Programming COHERENT, register, static, typedef

store() — DBM Function (libgdbm)

Write a record into a DBM data base
#include <dbm.h>
int store(*key***,** *datum***)**
datum *key***,** *datum***;**

Function **store()** writes a record into the currently open DBM data base. The data base must first have been opened by a call to **dbminit()**.

key points to the key by which the datum is identified. *datum* points to the datum itself. If the data base already contains a record with *key*, **store()** overwrites it.

The sizes of *key* and *datum* together must not exceed **BSIZE** bytes — that is, the size of one file-system block. (**BSIZE** is defined in header file **<sys/buf.h>**.)

If all goes well, **store()** returns zero. If an error occurs, it returns a negative value.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

strcasecmp() — Sockets Function (libsocket)

Case-insensitive string comparison
int strcasecmp (*left***,** *right***)**
char **left***,* **right***;*

Function **strcasecmp()** compares strings *left* and *right*. It returns zero if the strings are identical; -1 if *left* is lexigraphically less than (that is, occurs earlier in the alphabet) than *right*; or one if if *left* is lexicographically greater than *right*. Unlike the function **strcmp()**, **strcasecmp()** ignores case when it compares the strings.

See Also

libsocket, strcmp(), string.h

strcasencmp() — Sockets Function (libsocket)

Case-insensitive string comparison
int strcasencmp (*left***,** *right***,** *n***)**
char **left***,* **right***;*
int *n***;**

Function **strcasencmp()** compares the first *n* bytes of strings *left* and *right*. It returns zero if the first *n* bytes of the strings are identical; -1 if *left* is lexigraphically less than (that is, occurs earlier in the alphabet) than *right*; or one if if *left* is lexicographically greater than *right*. Unlike the function **strncmp()**, **strcasencmp()** ignores case when it compares the strings.

See Also

libsocket, strncmp(), string.h

strcat() — String Function (libc)

Concatenate two strings
#include <string.h>
char ***strcat(***string1***,** *string2***)**
char **string1***,* **string2***;*

strcat() appends all characters in *string2* onto the end of *string1*. It returns the modified *string1*.

Example

For an example of this function, see the entry for **string.h**.

See Also

libc, string.h, strncat()
ANSI Standard, §7.11.3.2
POSIX Standard, §8.1

Notes

string1 must point to enough space to hold itself and *string2*; otherwise, another portion of the program may be overwritten.

strchr() — String Function (libc)

Find a character in a string
#include <string.h>
char ***strchr(***string***,** *character***)**
char **string***; **int** *character***;**

strchr() searches for the first occurrence of *character* within *string*. The null character at the end of *string* is included within the search. It is equivalent to the COHERENT function **index()**.

strchr() returns a pointer to the first occurrence of *character* within *string*. If *character* is not found, it returns NULL.

Having **strchr()** search for a null character will always produce a pointer to the end of a string. For example,

```
char *string;
assert(strchr(string, '\0') == string + strlen(string));
```

never fails.

See Also

libc, string.h
ANSI Standard, §7.11.5.2
POSIX Standard, §8.1

strcmp() — String Function (libc)

Compare two strings
#include <string.h>
int strcmp(*string1***,** *string2***)**
char **string1***, **string2***;**

strcmp() compares *string1* with *string2* lexicographically. It returns zero if the strings are identical, returns a number less than zero if *string1* occurs earlier alphabetically than *string2*, and returns a number greater than zero if it occurs later. This routine is compatible with the ordering routine needed by **qsort()**.

Example

For examples of this function, see the entries for **string.h** and **malloc()**.

See Also

libc, qsort(), shellsort(), string.h, strncmp()
ANSI Standard, §7.11.4.2
POSIX Standard, §8.1

strcoll() — String Function (libc)

Compare two strings, using locale-specific information
#include <string.h>
int strcoll(*string1***,** *string2***)**
char **string1***; **char** **string2***;**

strcoll() lexicographically compares the string pointed to by *string1* with one pointed to by *string2*. Comparison ends when a null character is read.

strcoll() compares the two strings character by character until it finds a pair of characters that are not identical. It returns a number less than zero if the character in *string1* is less (i.e., occurs earlier in the character table) than its counterpart in *string2*. It returns a number greater than zero if the character in *string1* is greater (i.e., occurs later in the character table) than its counterpart in *string2*. If no characters are found to differ, then the strings are identical and **strcoll()** returns zero.

See Also

libc, localization, string.h
ANSI Standard, §7.11.4.3

Notes

The string-comparison routines **strcoll()**, **strcmp()**, and **strncmp()** differ from the memory-comparison routine **memcmp()** in that they compare strings rather than regions of memory. They stop when they encounter a null character, but **memcmp()** does not.

strcoll() differs from **strcmp()** and similar functions in that it reads the user's locale, as set by a call to function **setlocale()**, to determine the lexicographic value of each character. For details, see the Lexicon entry for **localization**.

strcpy() — String Function (libc)

Copy one string into another
#include <string.h>
char *strcpy(*string1***,** *string2***)**
char ****string1*,** ******string2***;**

strcpy() copies the contents of *string2*, up to the NUL, into the memory to which *string1* points. It returns *string1*.

Example

See **string**.

See Also

libc, memcpy(), string.h, strncpy()
ANSI Standard, §7.11.2.3
POSIX Standard, §8.1

Notes

string1 must point to enough space to hold *string2*, or another portion of the program or operating system may be overwritten.

strcspn() — String Function (libc)

Return length a string excludes characters in another
#include <string.h>
unsigned int strcspn(*string1***,** *string2***)**
char ****string1*,** ******string2***;**

strcspn() compares *string1* with *string2*. It then returns the length, in characters, for which *string1* consists of characters *not* found in *string2*.

See Also

libc, string.h
ANSI Standard, §7.11.5.3
POSIX Standard, §8.1

strdup() — String Function (libc)

Duplicate a string
#include <string.h>
char *strdup(*string***)**
char ****string*;**

The string function **strdup()** duplicates the text to which *string* points. It calls **malloc()** to allocate memory for the duplicate, copies the string, and returns a pointer to the memory that holds the copy. If something goes wrong, it returns NULL.

See Also

libc, string.h

Notes

strdup() is not part of the ANSI Standard. Using it in your programs may limit their portability.

stream — Definition

The term **stream** is a metaphor for any entity that can be named and from which bits can flow, such as a device or a file. The name "stream" reflects the fact that the C programming environment does not depend upon record descriptors and other devices that predetermine what form data can assume; instead, data from whatever source are conceived as being a flow of bytes whose significance is set entirely by the program that reads them.

For example, whether 16 bits forms an **int**, two **char**s, and should be used as an absolute value or a bit map, is entirely up to the program that receives it. It is also irrelevant to the program that processes these 16 bits whether they come from the keyboard, from a file on disk, or from a peripheral device.

The **FILE** structure holds all of the information needed to manipulate a stream. The STDIO functions can be used to open, close, or reopen a stream; read data from it; or write data to it.

See Also

bit, byte, data formats, file, FILE, Programming COHERENT, stdio.h

stream.h — Header File

Definitions for message facility
#include <stream.h>

stream.h definitions constants and structures used by the routines that implement the COHERENT version of STREAMS.

See Also

header files, STREAMS

STREAMS — Definition

COHERENT implementation of STREAMS

Beginning with release 4.2, COHERENT supports STREAMS. This is a system that helps programmers create system-independent device-drivers. STREAMS replaces most of the kernel-accessible routines that are unique to COHERENT.

For details on the COHERENT implementation of STREAMS, and for summaries of the STREAMS routines, see the manual that comes with release 2.2 of the COHERENT Device-Driver Kit.

To add the STREAMS driver to your kernel (should it not already have it), log in as the superuser **root** and then enter the following commands:

```
cd /etc/conf
streams/mkdev
bin/idmkcoh -o /kernel_name
```

where *kernel_name* names the new kernel to build. Then reboot to invoke the newly built *kernel_name*.

See Also

device drivers, getmsg(), Programming COHERENT, putmsg(), stropts.h

strerror() — String Function (libc)

Translate an error number into a string
#include <string.h>
char *strerror(*error***)**
int *error***;**

strerror() helps to generate an error message. It takes the argument *error*, which presumably is an error code generated by an error condition in a program, and may return a pointer to the corresponding error message.

The error numbers recognized and the texts of the corresponding error messages are set by COHERENT.

See Also

libc, perror(), string.h
ANSI Standard, §7.11.6.2

Notes

strerror() returns a pointer to a static array that may be overwritten by a subsequent call to **strerror()**.

strerror() differs from the related function **perror()** in the following ways: **strerror()** receives the error number through its argument *error*, whereas **perror()** reads the global constant **errno**. Also, **strerror()** returns a pointer to the error message, whereas **perror()** writes the message directly into the standard error stream.

The error numbers recognized and the texts of the messages associated with each error number are set by COHERENT. However, **strerror()** and **perror()** return the same error message when handed the same error number.

strftime() — Time Function (libc)

Format locale-specific time
#include <time.h>
size_t strftime(*string***,** *maximum***,** *format***,** *brokentime***)**
char **string***; **size_t** *maximum***; const char** **format***;*
const struct tm **brokentime***;*

The function **strftime()** provides a locale-specific way to print the current time and date. It also gives you an easy way to shuffle the elements of date and time into a string that suits your preferences.

strftime() references the portion of the locale that is affected by the calls

```
setlocale(LC_TIME, locale);
```

or

```
setlocale(LC_ALL, locale);
```

For more information on setting locales, see the entry for **localization**.

string points to the region of memory into which **strftime()** writes the date and time string it generates. *maximum* is the maximum number of characters that can be written into *string*. *string* should point to an area of allocated memory at least *maximum*+1 bytes long; if it does not, reserved portions of memory may be overwritten.

brokentime points to a structure of type **tm**, which contains the broken-down time. This structure must first be initialized by either of the functions **localtime()** or **gmtime()**.

Finally, *format* points to a string that contains one or more conversion specifications, which guide **strftime()** in building its output string. Each conversion specification is introduced by the percent sign '%'. When the output string is built, each conversion specification is replaced by the appropriate time element. Characters within *format* that are not part of a conversion specification are copied into *string*; to write a literal percent sign, use "%%".

strftime() recognizes the following conversion specifiers:

a The locale's abbreviated name for the day of the week.
A The locale's full name for the day of the week.
b The locale's abbreviated name for the month.
B The locale's full name for the month.
c The locale's default representation for the date and time.
d The day of the month as an integer (01 through 31).
H The hour as an integer (00 through 23).
I The hour as an integer (01 through 12).
j The day of the year as an integer (001 through 366).
m The month as an integer (01 through 12).
M The minute as an integer (00 through 59).
p The locale's way of indicating morning or afternoon (e.g, in the United States, "AM" or "PM").
S The second as an integer (00 through 59).
U The week of the year as an integer (00 through 53); regard Sunday as the first day of the week.
w The day of the week as an integer (0 through 6); regard Sunday as the first day of the week.
W The day of the week as an integer (0 through 6); regard Monday as the first day of the week.
x The locale's default representation of the date.
X The locale's default representation of the time.
y The year within the century (00 through 99).
Y The full year, including century.

z The name of the locale's time zone. If no time zone can be determined, print a null string.

Use of any conversion specifier other than the ones listed above will result in undefined behavior.

If the number of characters written into *string* is less than or equal to *maximum*, then **strftime()** returns the number of characters written. If, however, the number of characters to be written exceeds *maximum*, then **strftime()** returns zero and the contents of the area pointed to by *string* are indeterminate.

See Also

asctime(), ctime(), gmtime(), libc, localtime(), time [overview]
ANSI Standard, §7.12.3.5
POSIX Standard, §8.1

string.h — Header File

Declarations for string library
#include <string.h>

string.h is the header that holds the prototypes of all ANSI routines that handle strings and buffers. It declares the following routines:

fnmatch() . . . . . Match a string with a normal expression
index() . . . . . Search string for a character; use **strchr()** instead
memccpy(). . . . . Copy a region of memory up to a set character
memchr() . . . . . Search a region of memory for a character
memcmp(). . . . . Compare two regions of memory
memcpy() . . . . . Copy one region of memory into another
memmove() . . . . . Copy one region of memory into another with which it overlaps
memset(). . . . . Fill a region of memory with a character
pnmatch() . . . . . Match string pattern
strcat() . . . . . Concatenate two strings
strcmp() . . . . . Compare two strings
strncat() . . . . . Append one string onto another
strncmp() . . . . . Compare two lengths for a set number of bytes
strcpy(). . . . . Copy a string
strncpy(). . . . . Copy a portion of a string
strcoll(). . . . . Compare two strings, using locale information
strcspn() . . . . . Return length one string excludes characters in another
strdup(). . . . . Duplicate a string
strerror(). . . . . Translate an error number into a string
strlen() . . . . . Measure a string
strpbrk() . . . . . Find first occurrence in string of character from another string
strchr(). . . . . Find leftmost occurrence of character in a string
strrchr() . . . . . Find rightmost occurrence of character in a string
strspn(). . . . . Return length one string includes character in another
strstr() . . . . . Find one string within another string
strtok(). . . . . Break a string into tokens
strxfrm(). . . . . Transform a string, using locale information

Example

This example reads from **stdin** up to **NNAMES** names, each of which is no more than **MAXLEN** characters long. It then removes duplicate names, sorts the names, and writes the sorted list to the standard output. It demonstrates the functions **shellsort()**, **strcat()**, **strcmp()**, **strcpy()**, and **strlen()**.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NNAMES 512
#define MAXLEN 60

char *array[NNAMES];
char first[MAXLEN], mid[MAXLEN], last[MAXLEN];
char *space = " ";

int compare();
```

```
main()
{
      register int index, count, inflag;
      register char *name;

      count = 0;
      while (scanf("%s %s %s\n", first, mid, last) == 3) {
            strcat(first, space);
            strcat(mid, space);
            name = strcat(first, (strcat(mid, last)));
            inflag = 0;

            for (index=0; index < count; index++)
                  if (strcmp(array[index], name) == 0)
                        inflag = 1;

            if (inflag == 0) {
                  if ((array[count] =
                        malloc(strlen(name) + 1)) == NULL) {
                        fprintf(stderr, "Insufficient memory\n");
                        exit(EXIT_FAILURE);
                  }
                  strcpy(array[count], name);
                  count++;
            }
      }

      shellsort(array, count, sizeof(char *), compare);
      for (index=0; index < count; index++)
            printf("%s\n", array[index]);
      exit(EXIT_SUCCESS);
}

compare(s1, s2)
register char **s1, **s2;
{
      return(strcmp(*s1, *s2));
}
```

See Also

header files, libc, strcasecmp(), strcasencmp()
ANSI Standard, §7.1.1

Notes

Some implementations of UNIX call this header file **strings.h**. If you are porting code to COHERENT, you may have to modify the **#include** directives that invoke this header file.

The ANSI standard allows adjacent string literals, e.g.:

```
"hello" "world"
```

Adjacent string literals are automatically concatenated. Thus, the compiler will automatically concatenate the above example into:

```
"helloworld"
```

Because this departs from the Kernighan and Ritchie description of C, it will generate a warning message if you use the compiler's **-VSBOOK** option.

strings — Command

Print all character strings from a file
strings [-dopx] [-*length***] [***file ...* **]**

strings looks for ASCII strings in a binary file. A "string" is defined as any sequence of four or more printable characters. **strings** is useful for identifying unknown object files, or for looking at the messages printed by commands. You can also use it as a filter if *file* is not specified.

strings recognizes the following command-line options:

-d Precede each string by its offset in the file in decimal.

-o Precede each string by its offset in the file in octal.

-p Strip the parity bits of all characters in the string prior to comparison.

-x Precede each string by its offset in the file in hexadecimal.

Finally, the option **-***length* forces **strings** to use *length* as the minimum length for a printable string.

See Also

commands, isprint, od

strip — Command

Strip tables from executable file
strip *file [...]*

strip removes the symbol table, relocation information, and debug tables from a file. It makes the executable file noticeably smaller.

See Also

cc, commands, ld, nm, size

strlen() — String Function (libc)

Measure a string
#include <string.h>
int strlen(*string***)**
char **string***;*

strlen() measures *string*, and returns its length in bytes, *not* including the null terminator. This is useful in determining how much storage to allocate for a string.

Example

For an example of how to use this function, see the entry for **string**.

See Also

libc, string.h
ANSI Standard, §7.11.6.3
POSIX Standard, §8.1

strncat() — String Function (libc)

Append one string onto another
#include <string.h>
char ***strncat(***string1***,** *string2***,** *n***)**
char ***string1***,** ***string2***; unsigned** *n***;**

strncat() copies up to *n* characters from *string2* onto the end of *string1*. It stops when *n* characters have been copied or it encounters a null character in *string2*, whichever occurs first, and returns the modified *string1*.

Example

For an example of this function, see the entry for **strncpy**.

See Also

libc, strcat(), string.h
ANSI Standard, §7.11.3.2
POSIX Standard, §8.1

Notes

string1 should point to enough space to hold itself and *n* characters of *string2*. If it does not, a portion of the program or operating system may be overwritten.

strncmp() — String Function (libc)

Compare two strings
#include <string.h>
int strncmp(*string1***,** *string2***,** *n***)**
char **string1***, ***string2***; unsigned** *n***;**

strncmp() compares lexicographically the first *n* bytes of *string1* with *string2*. Comparison ends when *n* bytes have been compared, or a null character encountered, whichever occurs first. **strncmp()** returns zero if the strings are identical, returns a number less than zero if *string1* occurs earlier alphabetically than *string2*, and returns a number greater than zero if it occurs later. This routine is compatible with the ordering routine needed by **qsort()**.

Example

For an example of this function, see the entry for **strncpy()**.

See Also

libc, strcmp(), string.h
ANSI Standard, §7.11.4.4
POSIX Standard, §8.1

strncpy() — String Function (libc)

Copy one string into another
#include <string.h>
char *strncpy(*string1***,** *string2***,** *n***)**
char **string1***, ***string2***; unsigned** *n***;**

strncpy() copies up to *n* bytes of *string2* into *string1*, and returns *string1*. Copying ends when **strncpy()** has copied *n* bytes or has encountered a null character, whichever comes first. If *string2* is less than *n* characters in length, **strncpy()** pads *string1* to length *n* with one or more null bytes.

Example

This example, called **swap.c**, reads a file of names, and changes them from the format

```
first_name  [middle_initial] last_name
```

to the format

```
last_name, first_name [middle_initial]
```

It demonstrates **strncpy()**, **strncat()**, **strncmp()**, and **index()**.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define NNAMES 512
#define MAXLEN 60

char *array[NNAMES];
char gname[MAXLEN], lname[MAXLEN];

main(argc, argv)
int argc; char *argv[];
{
      FILE *fp;
      register int count, num;
      register char  *name, string[60], *cptr, *eptr;
      unsigned glength, length;

      if (--argc != 1) {
            fprintf (stderr, "Usage: swap filename\n");
            exit(EXIT_FAILURE);
      }

      if ((fp = fopen(argv[1], "r")) == NULL)
            printf("Cannot open %s\n", argv[1]);
      count = 0;
```

```
      while (fgets(string, 60, fp) != NULL) {
            if ((cptr = index(string, '.')) != NULL) {
                  cptr++;
                  cptr++;
            } else if ((cptr = index(string,' ')) != NULL)
                  cptr++;

            strcpy(lname, cptr);
            eptr = index(lname, '\n');
            *eptr = ',';

            strcat(lname," ");
            glength = (unsigned)(strlen(string) - strlen(cptr));
            strncpy(gname, string, glength);

            name = strncat(lname, gname, glength);
            length = (unsigned)strlen(name);
            array[count] = malloc(length + 1);

            strcpy(array[count],name);
            count++;
      }

      for (num = 0; num < count; num++)
            printf("%s\n", array[num]);
      exit(EXIT_SUCCESS);
}
```

See Also

libc, strcpy(), string.h
ANSI Standard, §7.11.2.4
POSIX Standard, §8.1

Notes

string1 must point to enough space to *n* bytes; otherwise, a portion of the program or operating system may be overwritten.

stropts.h — Header File

User-level STREAMS routines
#include <stropts.h>

The header file **stropts.h** gives user-level information about STREAMS system calls and calls to **ioctl()**.

See Also

header files, STREAMS

strpbrk() — String Function (libc)

Find first occurrence of a character from another string
#include <string.h>
char *strpbrk(*string1***,** *string2***)**
char ****string1*, ****string2***;**

strpbrk() returns a pointer to the first character in *string1* that matches any character in *string2*. It returns NULL if no character in *string1* matches a character in *string2*.

The set of characters that *string2* points to is sometimes called the "break string". For example,

```
char *string = "To be, or not to be: that is the question.";
char *brkset = ",;";
strpbrk(string, brkset);
```

returns the value of the pointer **string** plus five. This points to the comma, which is the first character in the area pointed to by **string** that matches any character in the string pointed to by **brkset**.

See Also

libc, string.h
ANSI Standard, §7.11.5.4

POSIX Standard, §8.1

Notes

strpbrk() resembles the function **strtok()** in functionality, but unlike **strtok()**, it preserves the contents of the strings being compared. It also resembles the function **strchr()**, but lets you search for any one of a group of characters, rather than for one character alone.

strrchr() — String Function (libc)

Search for rightmost occurrence of a character in a string
#include <string.h>
char *strrchr(*string***,** *character***)**
char ****string*; int** *character***;**

strrchr() looks for the last, or rightmost, occurrence of *character* within *string*. *character* is declared to be an **int**, but is handled within the function as a **char**. Another way to describe this function is to say that it performs a reverse search for a character in a string. It is equivalent to the COHERENT function **rindex()**.

strrchr() returns a pointer to the rightmost occurrence of *character*, or NULL if *character* could not be found within *string*.

See Also

libc, rindex(), string.h
ANSI Standard, §7.11.5.5
POSIX Standard, §8.1

strspn() — String Function (libc)

Return length a string includes characters in another
#include <string.h>
unsigned int strspn(*string1***,** *string2***)**
char ****string1*; char ****string2***;**

strspn() returns the length for which *string1* initially consists only of characters that are found in *string2*. For example,

```
char *s1 = "hello, world";
char *s2 = "kernighan & ritchie";
strcspn(s1, s2);
```

returns two, which is the length for which the first string initially consists of characters found in the second.

See Also

libc, string.h
ANSI Standard, §7.11.5.6
POSIX Standard, §8.1

strstr() — String Function (libc)

Find one string within another
#include <string.h>
char *strstr(*string1***,** *string2***)**
char ****string1*, ****string2***;**

The string function **strstr()** looks for *string2* within *string1*. The terminating NUL is not considered part of *string2*.

strstr() returns a pointer to where *string2* begins within *string1*, or NULL if *string2* does not occur within *string1*.

For example,

```
char *string1 = "Hello, world";
char *string2 = "world";
strstr(string1, string2);
```

returns **string1** plus seven, which points to the beginning of **world** within **Hello, world**. On the other hand,

```
char *string1 = "Hello, world";
char *string2 = "worlds";
strstr(string1, string2);
```

returns NULL because **worlds** does not occur within **Hello, world**.

See Also

libc, string.h
ANSI Standard, §7.11.5.7
POSIX Standard, §8.1

Notes

Neither *string1* nor *string2* can be more than 2,147,483,647 characters long.

strtod() — General Function (libc)

Convert string to floating-point number
#include <stdlib.h>
double strtod(*string, tailptr***)**
char ***string*; **char** ****tailptr*;**

strtod() converts the number given in *string* to a double-precision floating-point number and returns its value. It is a more general version of the function **atof()**. **strtod()** also stores a pointer to the first character following the number through *tailptr*, provided *tailptr* is not NULL.

strtod() parses the input *string* into three portions: beginning, subject sequence, and tail.

The *beginning* consists of zero or more white-space characters that begin the string.

The *subject sequence* is the portion of the input *string* that **strtod()** converts into a floating-point number. It consists of an optional sign character, a nonempty sequence of decimal digits optionally including a decimal-point character, and an optional exponent. If present, the exponent consists of either 'e' or 'E' followed by an optional sign and a nonempty sequence of decimal digits. **strtod()** reads characters until it encounters either a second decimal-point character or exponent marker, or any other non-numeral.

The *tail* continues from the end of the subject sequence to the null character that ends the string.

strtod() ignores the beginning portion of the string. It converts the subject sequence to a double-precision number. Finally, it sets the pointer pointed to by *tailptr* to the address of the first character of the string's tail.

strtod() returns the **double** generated from the subject sequence. If no subject sequence could be recognized, it returns zero and stores the initial value of *string* through *tailptr*. If the number represented by the subject sequence is too large or too small to fit into a **double**, then **strtod()** sets the global constant **errno** to **ERANGE** and returns **HUGE_VAL** or zero, respectively. If, however, the number given in the subject sequence has more digits to the right of the decimal point than can be encoded within an IEEE **double** (which has a fraction of 53 bits), **strtod** trims the excess digits before it converts the string.

Example

The following gives an example for **strtod()**.

```
#include <stdlib.h>

main()
{
      static char st[] = " 123.4 567.8";
      char *head, *tail;

      for (head = st;; head = tail) {
            double amt = strtod(head, &tail);

             /* No token found is end of string */
            if (head == tail)
                  break;
            printf("%f\n", amt);
      }
      exit(EXIT_SUCCESS);
}
```

See Also

atof(), double, errno, libc, limits.h, stdlib.h, strtol(), strtoul()
ANSI Standard, §7.10.1.4

Notes

strtod() ignores initial white space in the string pointed to by *string*; white space is defined as being all characters so recognized by the function **isspace()**.

strtok() — String Function (libc)

Break a string into tokens
#include <string.h>
char *strtok(*string1***,** *string2***)**
char ****string1*,** ******string2***;**

strtok() divides a string into a set of tokens. *string1* points to the string to be divided, and *string2* points to the character or characters that delimit the tokens.

strtok() divides a string into tokens by being called repeatedly.

On the first call to **strtok()**, *string1* should point to the string being divided. **strtok()** searches for a character that is *not* included within *string2*. If it finds one, then **strtok()** regards it as the beginning of the first token within the string. If one cannot be found, then **strtok()** returns NULL to signal that the string could not be divided into tokens. When it finds the beginning of the first token, **strtok()** then looks for a character that *is* included within *string2*. When it finds one, **strtok()** replaces it with NUL to mark the end of the first token, stores a pointer to the remainder of *string1* within a static buffer, and returns the address of the beginning of the first token.

On subsequent calls to **strtok()**, pass it NULL instead of *string1*. **strtok()** then looks for subsequent tokens using the address that it saved from the first time you called it.

Note that with each call to **strtok()**, *string2* may point to a different delimiter or set of delimiters.

Example

The following example breaks **command_string** into individual tokens and puts pointers to the tokens into the array **tokenlist[]**. It then returns the number of tokens created. No more than **maxtoken** tokens will be created. **command_string** is modified to place '\0' over token separators. The token list points into **command_string**. Tokens are separated by spaces, tabs, commas, semicolons, and newlines.

```
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>

tokenize(command_string, tokenlist, maxtoken)
char *command_string, *tokenlist[]; size_t maxtoken;
{
      static char tokensep[]="\t\n ,;";
      int tokencount;
      char *thistoken;

      if(command_string == NULL || !maxtoken)
            return 0;

      thistoken = strtok(command_string, tokensep);

      for(tokencount = 0; tokencount < maxtoken &&
                  thistoken != NULL;) {
            tokenlist[tokencount++] = thistoken;
            thistoken = strtok(NULL, tokensep);
      }

      tokenlist[tokencount] = NULL;
      return tokencount;
}

#define MAXTOKEN 100
char *tokens[MAXTOKEN];
char buf[80];

main()
{
      for(;;) {
            int i, j;
```

```
            printf("Enter string ");
            fflush(stdout);
            if(gets(buf) == NULL)
                  exit(EXIT_SUCCESS);

            i = tokenize(buf, tokens, MAXTOKEN);
            for (j = 0; j < i; j++)
                  printf("%s\n", tokens[j]);
      }
}
```

See Also

libc, string.h
ANSI Standard, §7.11.5.8
POSIX Standard, §8.1

strtol() — General Function (libc)

Convert string to long integer
#include <stdlib.h>
long strtol(*string, tailptr, base***)**
char ***string*; char ****tailptr*; int *base*;**

strtol() converts the number given in *string* to a **long** and returns its value; it is a more general version of the function **atol()**. **strtol()** also stores a pointer to the first character following the number through *tailptr*, provided *tailptr* does not equal NULL.

base gives the base of the number being read, either zero or a value from two to 36. If the given *base* is zero, **strtol()** determines an implicit base for the number: hexadecimal if the number starts with **0x** or **0X**, octal if the number starts with **0**, or decimal otherwise. Alternatively, you can specify a *base* between 2 and 36.

strtol() parses *string* into three portions: beginning, subject sequence, and tail.

The *beginning* consists of zero or more white-space characters that begin the string.

The *subject sequence* is the portion of the string that **strtol()** converts into a **long**. It consists of an optional sign character, an optional prefix **0x** or **0X** if the *base* is 16, and a nonempty sequence of *digits* in the specified base. For example, if the *base* is 16, then **strtol()** recognizes numeric characters '0' to '9' and alphabetic characters 'A' through 'F' and 'a' to 'f' as digits. It continues to scan until it encounters a nondigit.

The *tail* continues from the end of the subject sequence to the null character that ends the string.

strtol() ignores the beginning portion of the string. It converts the subject sequence to a **long**. Finally, if *tailptr* is not NULL, it sets the pointer pointed to by *tailptr* to the address of the first character of the string's tail.

strtol() returns a **long** representing the value of the subject sequence. If the input *string* does not specify a valid number, it returns zero and stores the initial value of *string* through *tailptr*. If the number it builds is too large or too small to fit into a **long**, it sets the global variable **errno** to the value of the macro **ERANGE** and returns **LONG_MAX** or **LONG_MIN**, respectively.

See Also

libc
ANSI Standard, §7.10.1.5

Notes

strtol() ignores initial white space in the input *string*. White space is defined as being all characters so recognized by the function **isspace()**.

strtoul() — General Function (libc)

Convert string to unsigned long integer
#include <stdlib.h>
unsigned long strtoul(*string, tailptr, base***)**
char ***string*; char ****tailptr*; int *base*;**

strtoul() converts the number given in *string* to a **unsigned long** and returns its value. It is the **unsigned long** counterpart of **strtol()** and a more general version of the function **atol()**. **strtoul()** also stores a pointer to the first character following the number through *tailptr*, provided *tailptr* does not equal NULL.

base gives the base of the number being read, either zero or a value from two to 36. If the given *base* is zero, **strtoul()** determines an implicit base for the number: hexadecimal if the number starts with **0x** or **0X**, octal if the number starts with **0**, or decimal otherwise. Alternatively, the user can specify an explicit *base* between two and 36.

strtoul() parses the *string* into three portions: beginning, subject sequence, and tail.

The *beginning* consists of zero or more white-space characters that begin the string.

The *subject sequence* is the portion of the string that **strtoul()** converts into an **unsigned long**. It consists of an optional sign character, an optional prefix **0x** or **0X** if the *base* is 16, and a nonempty sequence of *digits* in the specified base. For example, if the *base* is 16, then **strtoul()** recognizes numeric characters '0' to '9' and alphabetic characters 'A' through 'F' and 'a' to 'f' as digits. It continues to scan until it encounters a nondigit.

The *tail* continues from the end of the subject sequence to the null character that ends the string.

strtoul() ignores the beginning portion of the string. It converts the subject sequence to an **unsigned long**. Finally, if *tailptr* does not equal NULL , it sets the pointer pointed to by *tailptr* to the address of the first character of the string's tail.

strtoul() returns an **unsigned long** representing the value of the subject sequence. If the input *string* does not specify a valid number, it returns zero and stores the initial value of *string* through *tailptr*. If the number it builds is too large to fit into an **unsigned long**, it sets the global variable **errno** to the value of the macro **ERANGE** and returns **ULONG_MAX**.

Example

This example uses **strtoul()** as a hash function for table lookup. It demonstrates both hashing and linked lists. Hash-table lookup is the most efficient when used to look up entries in large tables; this is an example only.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*
 * For fastest results, use a prime about 15% bigger
 * than the table. If short of space, use a smaller prime.
 */
#define HASHP 11
struct symbol {
      struct symbol *next;
      char *name;
      char *descr;
} *hasht[HASHP], codes[] = {

      NULL,        "a286",            "frogs togs",
      NULL,        "xy7800",          "doughnut holes",
      NULL,        "z678abc",         "used bits",
      NULL,        "xj781",           "black-hole varnish",
      NULL,        "h778a",           "table hash",
      NULL,        "q167",            "log(-5.2)",
      NULL,        "18888",           "quid pro quo",
      NULL,        NULL,              NULL    /* end marker */
};

void
buildTable()
{
      long h;
      register struct symbol *sym, **symp;

      for(symp = hasht; symp != (hasht + HASHP); symp++)
            *symp = NULL;
```

```
      for(sym = codes; sym->descr != NULL; sym++) {
            /*
             * hash by converting to base 36. There are
             * many ways to hash, but use all the data.
             */

            h = strtoul(sym->name, NULL, 36) % HASHP;
            sym->next = hasht[h];
            hasht[h] = sym;
      }
}

struct symbol *
lookup(s)
char *s;
{
      long h;
      register struct symbol *sym;

      h = strtoul(s, NULL, 36) % HASHP;
      for(sym = hasht[h]; sym != NULL; sym = sym->next)
            if(!strcmp(sym->name, s))
                  return(sym);
      return(NULL);
}

main()
{
      char buf[80];
      struct symbol *sym;

      buildTable();
      for(;;) {
            printf("Enter name ");
            fflush(stdout);

            if(gets(buf) == NULL)
                  exit(EXIT_SUCCESS);

            if((sym = lookup(buf)) == NULL)
                  printf("%s not found\n", buf);

            else
                  printf("%s is %s\n", buf, sym->descr);
      }
}
```

See Also

errno, libc, limits.h, stdlib.h, strtol()
ANSI Standard, §7.10.1.6

Notes

strtoul() ignores initial white space in the input *string*. White space is defined as being all characters so recognized by the function **isspace()**.

struct — C Keyword

Data type

struct is a C keyword that introduces a structure. The following is an example of how **struct** can be used in the description of a name and address file:

```
struct address {
      char firstname[10];
      char lastname[15];
      char street[25];
      char city[10];
      char state[2];
      char zip[5];
      int  salescode;
};
```

The C Programming Language, second edition prohibits the assignment of structures, the passing of structures to functions, and the returning of structures by functions. COHERENT, however, lifts these restrictions. It allows one structure to be assigned to another, provided the two structures are of the same type. It also allows structures to be passed by and returned by functions. These features are supported by most compilers, but users should be aware that their use can cause problems in porting code to some compilers.

See Also

array, C keywords, field, initialization, structure
ANSI Standard, §3.1.2.5, §3.5.2.1

structure — Definition

A *structure* is a set of variables that has been given a name and can be manipulated as a single entity. The variables may be of different data types. Structures are a convenient way to deal with data elements that belong together, such as names and addresses, employee descriptions, or sales and inventory information.

See Also

Programming COHERENT, struct

structure assignment — Definition

The C Programming Language, second edition forbids structure assignment, the passing of structures to functions, and returning structures from functions (as opposed to the passing or returning of *pointers* to structures). The COHERENT C compiler lifts these restrictions.

Some C compilers transform structure arguments and structure returns into structure pointers. Note that the use of structure assignment, structure arguments, or structure returns may create problems when porting the code to another C compiler.

See Also

portability, Programming COHERENT, struct, structure

Notes

Because this feature deviates from the description of the C language found in the first edition of *The C Programming Language* compiling with the **-VSBOOK** option will flag all points where it occurs in your program.

strxfrm() — String Function (libc)

Transform a string using locale information
#include <string.h>
unsigned int strxfrm(*string1***,** *string2***,** *n***)**
char ******string1***,** ******string2***;** **unsigned int** *n***;**

strxfrm() transforms *string2* using information concerning the program's locale, as set by the function **setlocale()**. It then writes up to *n* bytes of the transformed result into the area to which *string1* points. It returns the length of the transformed string, not including the terminating null character. The transformation incorporates locale-specific material into *string1*.

If *n* is set to zero, **strxfrm()** returns the length of the transformed string.

If two strings return a given result when compared by **strcoll()** before transformation, they will return the same result when compared by **strcmp()** after transformation.

Example

The following simple program demonstrates **strxfrm()**:

```
#include <stdio.h>
#include <string.h>

main()
{
      char string1[20], string2[20];

      strcpy (string1, "This is string 1");
      strcpy (string2, "This is string 2");

      printf ("String 1 before transformation: %s\n", string1);
      printf ("String 2 before transformation: %s\n", string2);

      strxfrm (string1, string2, 18);

      printf ("String 1 after transformation: %s\n", string1);
      printf ("String 2 after transformation: %s\n", string2);
}
```

See Also

libc, string.h
ANSI Standard, §7.11.4.5

Notes

If **strxfrm()** returns a value equal to or greater than *n*, the contents of the area to which *string1* points are indeterminate.

stty — Command

Set/print terminal modes
stty
stty -a
stty -g
stty *x*:*x*: **...** :*x*
stty *arglist ...*

The command **stty** lets you change or display the settings of the standard input device. The device is usually a terminal, although tapes, disks and other special files may be applicable.

Default Settings

The following describes how COHERENT sets up a terminal device by default. This normal processing is often called "cooked" mode. Note that on some machines, the default characters differ from those given below.

The *erase* and *kill* characters (normally **<crtl-H>** and **<ctrl-U>**) erase, respectively, one typed character and an entire line of typing.

The *stop-output* and *start-output* characters (normally **<ctrl-S>** and **<ctrl-Q>**) respectively stop and restart output.

The *interrupt* character (normally **<ctrl-C>**) sends the signal **SIGINT**, which usually terminates program execution.

The *quit* character (normally **<ctrl-\>**) sends the signal **SIGQUIT**, which usually terminates program execution with a core dump.

The *end of file* character (normally **<ctrl-D>**) generates an end-of-file signal from the terminal.

You can change the setting of each special character by invoking **stty** with the appropriate option.

Options

When called without any arguments, **stty** gives a brief listing of settings for the standard-input device.

stty can read the settings of devices other than the standard-input device by redirecting that device to it. For example, the command

```
stty < /dev/com1l
```

prints a brief summary of the settings for serial device **com1l**.

stty's command-line arguments can take a number, as indicated below by *n*; or they can take a character, as indicated below by *c*. Argument *c* can be one of the following:

- A single character.
- A caret '^' followed by a single character (to indicate a control character, e.g., **^X** for **<ctrl-X>**).
- An **^?**, which denotes the **<del>** character.
- An '0x' followed by two hexadecimal digits.
- An **^-**, which indicates that that option is not used.

stty recognizes the following command-line arguments:

-a Give a complete listing of settings for the standard-input device.

-g Give a complete list of settings for the standard-input device, but in hexadecimal. This is a dump of the **termio** structure in effect at the moment. For more information on the **termio** structure, see the Lexicon entry for **termio**.

x:*x*:...:*x* Establish new settings for the standard-input device. The settings are hexadecimal values that are separated by colons. This form can be combined with **-g** option to copy **stty** settings from one device to another. For example, to set device **com2l** so that it mimics device **com1l**, use the following command:

```
stty `stty -g < /dev/com1l` < /dev/com2l
```

0 Hang up the telephone.
50 Set line speed to 50 bps.
75 Set line speed to 75 bps.
110 Set line speed to 110 bps.
134 Set line speed to 110 bps.
150 Set line speed to 150 bps.
200 Set line speed to 200 bps.
300 Set line speed to 300 bps.
600 Set line speed to 600 bps.
1200 Set line speed to 1200 bps.
1800 Set line speed to 1800 bps.
2400 Set line speed to 2400 bps.
4800 Set line speed to 4800 bps.
9600 Set line speed to 9600 bps.
19200 Set line speed to 19200 bps.
38400 Set line speed to 38400 bps.
brkint Send interrupt on break.
-brkint Do not send interrupt on break.
bs0 No delay on backspace.
bs1 Delay briefly on backspace.
clocal Turn on modem control.
-clocal Turn off modem control.
cooked Set the device into cooked mode. This is a composite of options **parenb**, **-parodd**, **cs7**, **brkint**, **ignpar**, **istrip**, **icrnl**, **ixon**, **opost**, **onlcr**, **isig**, and **icanon**.
cr0 No delay on carriage returns.
cr1 Carriage-return delay depends upon column position.
cr2 Delay approximately 0.10 seconds on carriage return.
cr3 Delay appoximately 0.15 seconds on carriage return.
cread Enable the receiver.
-cread Disable the receiver.
cs5 Character size is five bits.
cs6 Character size is six bits.
cs7 Character size is seven bits.
cs8 Character size is eight bits.
cstopb Use two stop bits per character.
-cstopb Use one stop bit per character.
echo Echo every character.
-echo Do not echo characters.
echoe Echo the erase character as backspace-space-backspace.
-echoe Do not echo the erase character as backspace-space-backspace.
echok Echo newline after the kill character.
-echok Do not echo newline after the kill character.

echonl Echo newline.
-echonl Do not echo newline.
ek Set the kill and erase characters to printable characters. A composite of **erase '#'** and **kill '@'**.
eof *c* Set the end-of-file character to *c*.
eol *c* Set the end-of-line character to *c*.
erase *c* Set the erase character to *c*.
evenp Set the port to even parity. This is a composite of the options **parenb**, **-parodd**, and **cs7**.
-evenp Turn off even parity — in effect, turn off parity altogether. This is a composite of the options **-parenb** and **cs8**.
ff0 No delay on formfeeds.
ff1 Delay approximately two seconds on formfeeds.
hup Hang up the telephone on logging out.
-hup Do not hang up the telephone on logging out.
hupcl Same as **hup**.
-hupcl Same as **-hup**.
icanon Enable canonical input.
-icanon Disable canonical input.
icrnl Map carriage-return to newline on input.
-icrnl Do not map carriage-return to newline on input.
ignbrk Ignore break on input.
-ignbrk Do not ignore break on input.
igncr Ignore carriage return on input.
-igncr Do not ignore carriage return on input.
ignpar Ignore parity errors on input.
-ignpar Do not ignore parity errors on input.
inlcr Map newline to carriage return on input.
-inlcr Do not map newline to carriage return on input.
inpck Enable parity checking on input.
-inpck Do not enable parity checking on input.
intr *c* Set the interrupt character to *c*.
isig Check input against interrupt and quit characters.
-isig Do not check input against interrupt and quit characters.
iuclc Map input's upper-case characters to lower case.
-iuclc Do not map input's upper-case characters to lower case.
istrip Strip input to seven bits.
-istrip Do not strip input to seven bits.
ixany Allow any on input character to restart output.
-ixany Do not allow any input character to restart output.
ixoff Request that system send start or stop characters when the input queue is, respectively, nearly full or nearly empty.
-ixoff Do not request that system send start or stop characters to manage input queue.
ixon Use start/stop characters to control output queue.
-ixon Do not use start/stop characters to control output queue
kill *c* Set the kill character to *c*.
lcase Map upper-case characters to lower case. A composite of options **xcase**, **iuclc**, and **olcuc**.
-lcase Turn off mapping of upper-case character to lower case. A composite of options **-xcase**, **-iuclc**, and **-olcuc**.
LCASE A synonym for **lcase**.
-LCASE A synonym for **-lcase**.
min *n* Set the constant **VMIN** to decimal value *n*. For more about **VMIN**, see the Lexicon entry for **termio**.
nl A composite of options **-icrnl** and **-onlcr**.
-nl A composite of options **icrnl**, **-inlcr**, **-igncr**, **onlcr**, **-ocrnl**, and **-onlret**.
nl0 No delay on newline.
nl1 Delay approximately 0.10 seconds on newline.
noflsh Flush buffer on interrupt or quit.
-noflsh Do not flush buffer on interrupt or quit.
ocrnl In output, map carriage return to newline.
-ocrnl In output, do not map carriage return to newline.
oddp Set device to odd parity. This option is a composite of the options **parenb**, **parodd**, and **cs7**.
-oddp Turn off odd parity — in effect, turn off parity altogether. This is a composite of the options **-parenb** and **cs8**.

LEXICON

ofdel Use delete characters as fill characters.
-ofdel Do not use delete characters as fill characters.
ofill Use fill characters for delays.
-ofill Do not use fill characters for delays.
olcuc Map lower-case characters to upper case on output.
-olcuc Do not map lower-case characters to upper case on output.
onlcr Map newline to carriage return/newline on output.
-onlcr Do not map newline to carriage return/newline on output.
onlret A newline character executes a carriage return.
-onlret A newline character does not execute a carriage return.
onocr Do not output carriage returns at column 0.
-onocr Output carriage returns at column 0.
opost Post-process output.
-opost Do not post-process output.
parenb Enable parity generation and detection.
-parenb Disable parity generation and detection.
parity Synonym for option **evenp**.
-parity Synonym for option **-evenp**.
parmrk Mark parity errors.
-parmrk Do not mark parity errors.
parodd Odd parity.
-parodd Turn off odd parity; i.e., use even parity.
quit *c* Set the quit character to *c*.
raw Set the device into raw mode. This is a composite of the options **-parenb**, **-parodd**, **-hupcl**, **cs8**, **-opost**, **-olcuc**, **-ocrnl**, **-onocr**, **-onlret**, **-ofill**, **-ofdel**, **nl0**, **cr0**, **tab0**, **bs0**, **vt0**, and **ff0**. This turns off most character processing, including all input processing (see **c_iflag** fields in **<termio.h>**), canonical input buffering (**-icanon**), and output processing (**-opost**). It does not turn off echo.
-raw Turn off raw mode — in effect, restore the device to cooked mode. Same as **cooked**.
sane Restore the device to "sanity" — for example, after an editor or communications program has died unexpectedly. This is a composite of options **icrnl**, **opost**, **onlcr**, **isig**, **icanon**, **-xcase**, **echo**, **echoe**, **echok**, and **erase ^h**.
tab0 No delay for horizontal-tab character.
tab1 Delay for horizontal-tab character depends on column position.
tab2 Delay approximately 0.10 seconds on horizontal tab.
tab3 Expand horizontal-tab characters into spaces.
tabs A synonym for **tab0**.
-tabs A synonym for **tab3**.
time *n* Set the constant **VTIME** to decimal value *n*. For more about **VTIME**, see the Lexicon entry for **termio**.
vt0 No delay on vertical-tab characters.
vt1 Delay approximately two seconds on vertical-tab characters.
xcase Canonical presentation of upper-case and lower-case characters.
-xcase Do not process upper-case and lower-case characters.

See Also

ASCII, commands, getty, init, ioctl(), signal()

Notes

Executing **stty** with input redirected from another device does not have an effect unless the device being read is open. The last close of any terminal device resets all **termio** values to the system defaults. Thus, to change the settings of a device, you must first open the device.

For example,

```
enable com1l
```

or

```
sleep 32000 > /dev/com1l &
```

might precede:

```
stty evenp < /dev/com1l
```

Note, too, that **stty** does not check its arguments for consistency.

stty provides complete access to the System-V-style **termio** structure. Note, however, that the settings of **termio** are processed by the kernel's in-line discipline and device-driver modules. Under COHERENT, none of these modules pays attention to delay settings. Therefore, setting delays with **stty** does not, at present, affect the behavior of the terminal device.

stty() — System Call (libc)

Set terminal modes
#include <sgtty.h>
int stty(*fd, sgp***)**
int *fd***;**
struct sgttyb **sgp***;*

The COHERENT system call **stty()** sets a terminal's attributes. See the Lexicon article for **stty** for information on terminal attributes and their legal values.

Example

This example demonstrates both **stty()** and **gtty()**. It sets terminal input to read one character at a time (that is, it reads the terminal in "raw" form). When you type 'q', it restores the terminal to its previous settings, and exits. For an additional example, see the **pipe** Lexicon article.

```
#include <sgtty.h>

main()
{
        struct sgttyb os, ns;
        char buff;

        printf("Waiting for q\n");
        gtty(1, &os);               /* save old state */
        ns = os;                    /* get base of new state */
        ns.sg_flags |= RAW;         /* prevent <ctl-c> from working */
        ns.sg_flags &= ~(ECHO|CRMOD);/* no echo for now... */
        stty(1, &ns);               /* set mode */

        do {
                buff = getchar();      /* wait for the keyboard */
        } while(buff != 'q');

        stty(1, &os);          /* reset mode */
}
```

Files

<sgtty.h> — Header file

See Also

exec, gtty(), ioctl(), libc, open(), read(), sgtty.h, stty, write()

Notes

Please note that if you use **stty()** to change the baud rate on a port, you must first invoke **sleep()**. If you do not, the port reverts back to its default settings.

stune — System Administration

Set values of tunable kernel variables
/etc/conf/stune

File **stune** names each tunable variable within the kernel, and gives the value to which it is actually set. Command **idmkcoh** reads this file when it builds a new kernel, and uses its contents to patch the kernel appropriately.

Each entry within this file has two fields. The first field names the variable; the name must match that given in **stune**. The second field gives the value of the variable; this value must fall between the minimum and maximum values given in **stune**.

If a line begins with a pound sign '#', it is a comment and **idmkcoh** ignores it. If a tunable variable is not named in

this file, **idmkcoh** uses the default value given in **stune**.

See Also

Administering COHERENT, device drivers, mdevice, mtune, sdevice

su — Command

Substitute user id, become superuser
su [*user* **[***command***]]**

Default *user* is **root**; default *command* is **sh**. **su** changes the real user id and the effective user id to that of the *user*. If *user* has a login password, **su** requests it. Then it executes the specified *command*.

If *command* is absent, **su** invokes an interactive sub-shell.

If *user* is absent, **su** assumes user name **root** (the superuser).

Files

/etc/passwd — Login names and passwords

See Also

commands, login, newgrp, sh, superuser

sum — Command

Print checksum of a file
sum [*file ...***]**

sum prints an unsigned integer checksum and a size in blocks (rounding up) for each *file* specified. If more than one *file* is specified, **sum** also prints the file name. If no *file* is specified, **sum** reads the standard input.

sum may be used to verify the integrity of data transferred across phone lines or stored on an unreliable medium.

See Also

cmp, commands

superuser — Definition

The *superuser* is the user who has system-wide permissions. He can execute any program, read any file, and write into any directory. Thus, superuser status is reserved to the system administrator, also called **root**, who needs this status to control the operation of the system.

No person should be able to become the superuser without knowing a password. Because the superuser in effect "owns" the system, the superuser password should be guarded most carefully.

See Also

root, su, Using COHERENT

swab() — General Function (libc)

Swap a pair of bytes
void swab(*src***,** *dest***,** *nb***) char** ******src***,** ******dest***; unsigned** *nb***;**

The ordering of bytes within a word differs from machine to machine. This may cause problems when moving binary data between machines. **swab()** interchanges each pair of bytes in the array *src* that is *n* bytes long, and places the result into the array *dest*. The length *nb* should be an even number, or the last byte will not be touched. *src* and *dest* may be the same place.

Example

This example prompts for an integer; it then prints the integer both as you entered it, and as it appears with its bytes swapped.

```
#include <stdio.h>

main()
{
      int word;
```

```
      printf("Enter an integer: \n");
      scanf("%d", &word);
      printf("The word is 0x%x\n", word);
      swab(&word, &word, 2);
      printf("The word with bytes swapped is 0x%x\n", word);
}
```

See Also

byte ordering, dd, canon.h, libc

switch — C Keyword

Test a variable against a table

switch is a C keyword that lets you perform a number of tests on a variable in a convenient manner. For example,

```
while(foo < 10)
      switch(foo) {
      case 1:
            dosomething();
            break;
      case 2:
            somethingelse();
      case 3:
            anotherthing();
            break;
      default:
            break;
      }
}
```

is equivalent to

```
while(foo < 10) {
      if(foo == 1) {
            dosomething();
            continue;
      } else if (foo == 2) {
            somethingelse();
            anotherthing();
            continue;
      } else if(foo == 3) {
       /* Note: compiler eliminates duplicate code */
            anotherthing();
            continue;
      } else
            break;
}
```

switch is always used with the **case** statement, and nearly always with the **default** statement.

See Also

break, C keywords, case, default, while
ANSI Standard, §6.6.4.2

sync — Command

Flush system buffers
sync

Most COHERENT commands manipulate files stored on a disk. To improve system performance, the COHERENT system often changes a copy of part of the disk in a buffer in memory, rather than repeatedly performing the time-consuming disk access required.

sync writes information from the memory buffers to the disk, updating the disk images of all mounted file systems which have been changed. In addition, it writes the date and time on the root file system.

sync should be executed before system shutdown to ensure the integrity of the file system.

See Also

commands

sync() — System Call (libc)

Flush system buffers
sync()

sync() is the COHERENT system call that copies the contents of all memory buffers to disk.

See Also

libc

sys — System Administration

Data base for UUCP connections
/usr/lib/uucp/sys

The file **/usr/lib/uucp/sys** describes how to communicate with a remote system. The UUCP daemon **uucico** uses the information in this file to telephone a remote system, log into a remote system, and control what it allows a remote system to do on your system.

Command **cu** also reads file **sys** for information on how it can call a remote system. However, the following descriptions concentrate on how **sys** is used by **uucico**.

Structure of the sys File

sys has the following structure:

command *argument*
...
alternate
command *argument*
...
alternate
command *argument*
...
system *remotesystem*
command *argument*
...
alternate
command *argument*
...
system *remotesystem*
command *argument*
...
alternate
command *argument*
...

Blank lines in the file are ignored. The body of the file consists of a series of commands. Each command defines one or more values; each value, in turn, determines one aspect of how your system interacts with a remote system. A backslash at the end of a line lets an entry extend over more than one one line.

The commands from the top of **sys** to the first **system** command set global values — that is, the values used by default when dealing with every remote system. Note that **uucico** recognizes a number of global values that are not explicitly written in **sys**.

The command **system** names a remote system. The commands from one **system** command to the next (or the end of the file, whichever comes first) define the values that **uucico** uses when it communicates with that system. These system-specific values can override any of the global values.

The command **alternate** introduces a block of alternate values. The commands from one **alternate** command to the next **alternate** command (or to the next **system** command or to the end of the file, whichever comes first) set a block of alternate values. **uucico** uses a block of alternate values when the default values (and all preceding blocks of alternate values) fail for any reason. By defining blocks of alternate values, you can define multiple ways to interact with a remote system.

Order of Command Execution

As you can see the above display, both the global section and each system-specific section can contain blocks of alternate commands. The order in which **uucico** reads blocks of commands is important: each block can contain its own version of a given command, and **uucico** uses the value set by the command that it has read *last*.

The following describes the order in which **uucico** reads commands when it attempts to call site *remotesite*:

1. **uucico** reads its default global values (which are described below). It then reads the global-values section of **sys**, up to the first **alternate** command.
2. **uucico** reads the section *remotesite*, from its **system** command to the first **alternate** command.
3. **uucico** calls *remotesite*.
4. If the call to *remotesite* succeeds, then all is well. If it fails, however, then **uucico** reads the first block of alternate commands in the global section, then the first block of alternate commands in the section for *remotesite*.

   Note that a block of alternate values can simply reproduce values set previously. This in effect forces **uucico** to try the same values once again.
5. **uucico** again calls *remotesite*.
6. If the call succeeds, then all is well. If it fails, **uucico** reads the second block of alternate values (should there be one) in the global-defaults section, then the second block of alternate values (again, should there be one) in the section for *remotesystem*. **uucico** makes its third attempt to call *remotesite*.

This process continues either until **uucico** succeeds in getting through to *remotesite*, or until it runs out of blocks of alternate values in both the global section and in the site-specific section.

As you can see, it can be difficult at times to tell just what values **uucico** is using at any given time. The command **uuchk** can help you untangle this skein of values. See its Lexicon entry for details.

Structural Commands

The following commands help control the manner in which **uucico** reads commands from **sys**:

system *remotesystem*
: Name the remote system. All commands up to the next **system** command refer to the system *remotesystem*.

alternate [*name***]**
: Introduce an alternate set of commands. The optional *name* lets you name this block of alternate commands; if **uucico** uses this block of alternate commands, it records *name* in the log file for *remotesystem*.

default-alternates true | false
: If its argument is **false**, do not use any blocks of alternate values from the global section. The default is **true**.

Chat Commands

The command **chat** defines a chat script. A *chat script* summarizes the conversation that your system has with the remote system as it attempts to log into that system.

chat has the following structure:

> **chat** *expect respond expect respond ... expect respond*

As you can see, a chat script consists of pairs of strings. Each pair contains an *expect* string, which gives what you expect the remote system to say to your system; and a *respond* string, which gives what your system sends in reply. When **uucico** runs out of *expect/respond* pairs, it assumes that it has succeeded in logging into *remotesystem*. If you want to send something to the remote system without waiting an *expect* string, then the *expect* string in a *expect/respond* pair should be simply a pair of quotation marks with nothing between.

Each string in the chat script is demarcated by white space. Therefore, you must use the escape sequence '**\s**' to indicate white space within a string. You can embed other escape sequences within the *respond* string; these are given below.

LEXICON

An *expect* string can contain several sub-strings separated by hyphens. The sub-strings themselves comprise pairs of *expect*/*respond* strings. If your system does not receive the first expect sub-string, it can send the first *respond* string (to prod the remote system), then await the second *expect* string; and so on, until your system either runs out of sub-strings or it receives an expect sub-string that it recognizes. You can, of course, repeat the same *expect*/*respond* pair more than once. Because sub-strings are separated by hyphens, you cannot use a literal hyphen in a string; you should indicate a literal hyphen by the escape sequence '\055' (ASCII for the hyphen character).

You can embed the following escape sequences in a *respond* string:

****	Literal backslash character
*****DDD*	Character with octal value *DDD*
\b	Backspace
\c	Suppress carriage return at end of send string
\d	Delay sending for one or two seconds
\E	Enable echo checking
\e	Disable echo checking
\K	Same as **BREAK**
\L	Your system's login name
\N	NUL
\n	Newline or line feed
\P	The password on the system being contacted
\p	Pause sending for a fraction of a second
\r	Carriage return
\s	Space
\t	Tab
\x*DDD*	Character with hexadecimal value *DDD*
\Z	Send name of the system being called
EOT	End-of-transmission character (**<ctrl-D>**)
BREAK	Break character

As in C, up to three octal digits may follow a backslash The escape sequence **\x** can be followed by an indefinite number of hexadecimal digits. To follow a hexadecimal escape sequence with a hexadecimal digit, interpose a send string of "".

uucico sends a carriage return at the end of each send string, unless the escape sequence **\c** appears in the string.

"Echo checking" means that after **uucico** writes each character, it waits for the remote system to echo it. You must turn on echo checking separately for each send string for which you want it.

The following gives an example chat script; the numbers simply mark the elements of the chat script for the discussion that follows, and are not part of the chat script:

```
          1    2      3                            4     5     6
    chat "" \r\c ogin:-BREAK-ogin:-BREAK-ogin: nuucp word: public
```

This script does the following:

1. Expect nothing from the modem (as indicated by the empty string "").
2. Send newline and carriage-return characters, as indicated by the escape sequence **\r\c**.
3. Expect the string **ogin:** (or a string that ends with **ogin:**). If this is not received within the defined pause period, send a break character (as indicated by the escape sequence **BREAK**), and wait again for **ogin:**. If the procedure times out again, send another break character and wait again. If the third attempt times out, quit.
4. Having received **ogin:** from **remotesystem**, send the string **nuucp**.
5. Wait for the string **word:**, that is, the tail of the prompt **Password:**.
6. When the password prompt is received, reply with the password **public**.

Some users may experience trouble when logging into a machine that is running SCO UNIX: it appears not to recognize carriage returns. The simplest work around is to embed the "delay" escape sequence **\d** in the send strings. For example, if you were using the above chat script to communicate with a SCO UNIX system, and the system was not responding to your transmission, you could modify it as follows:

```
chat "" \r\c ogin:-BREAK-ogin:-BREAK-ogin: \d\dnuucp\d\d\d\d word: \d\dpublic\d\d\d
```

This slows how your system responds to the SCO system; giving it enough time to "digest" your transmission appears to work around the problem. You can try adjusting the number of \d characters to get best performance.

The following commands help control the chat your system has with a *remotesystem*:

chat-fail *string*
Abort the chat if *string* is received. *string* cannot contain white-space characters; use escape sequences instead.

The description for *remotesystem* can contain multiple **chat-fail** commands. The default is to have none.

chat-program *program arguments*
Pass *arguments* (if any) to *program*, which is the path name of a program that you want **uucico** to execute before it executes your **chat** command. *program* can contain its own version of a chat script, but this is not required. If both a system's description contains both the commands **chat-program** and **chat**, **uucico** always executes the former first.

arguments can contain any of the following escape sequences:

\Y Port device name
\S Port speed
**** Literal backslash

uucico connects the standard input and standard output of *program* to the port in use, and connects the standard error of *program* to the UUCP log file. If *program* does not exit with a status of zero, **uucico** assumes that it has failed.

uucico runs *program* as user **uucp**, and the environment is that of the process that invoked **uucico**. Take care that by using *program*, you do not compromise your system's security.

chat-seven-bit true | false
If the argument is **true**, **uucico** strips all incoming characters to seven bits before it compares them with the expect string; otherwise, it uses all eight bits. The default is **true** because some UNIX systems generate parity bits during the login prompt that must be ignored while running a chat script.

chat-timeout *seconds*
Wait *seconds* for the remote system to respond to a send string. If send string times out, **uucico** sends the next send sub-string (if there is one), or fails. The default is timeout time is 60 seconds.

Aliases and Identifiers

The following commands let you manipulate how your system identifies itself to a *remotesystem*:

alias *systemalias*
Define *systemalias* to be an alias for *remotesystem*. The commands **uucp** and **uux** can use *systemalias*, as can *remotesystem* itself. This command is helpful should *remotesystem* change its name: it spares you the trouble of having to comb through your system to replace every occurrence of the old name. The default is to have no aliases.

myname *mysysname*
Tell your system to identify itself as *mysysname* instead of its true name (as kept in file **/etc/uucpname**) when it calls *remotesystem*.

If the description of *remotesystem* includes the command **called-login** without the argument **ANY**, your system will identify itself as *mysysname* when it is called by *remotesystem*.

call-login *loginname*
Tell **uucico** how to expand the escape sequence **\L**, which stands for the login name. With this command, you can use a default chat script with several different systems, expanding the login escape sequence (and password, as will be shown next) with the appropriate strings.

call-password *password*
Tell **uucico** how to expand the escape sequence **\P**, which stands for a password. As with the command **call-login**, described above, this command lets you use the same chat script with a number of different systems, by expanding the login and password escape sequences as needed.

Accepting a Call

The following commands affect how your system handles a call from another system:

called-login *login_identifer* **[***remotesystem ...* **]**
Recognize the remote system with the name *login_identifier* when it attempts to log into your system. If you set *login_identifier* to **ANY**, **uucico** will accept any login identifier. The optional *remotesystem* arguments name each remote system that is allowed to log in under that login identifier.

Some systems use this command to select a number of different alternate sections within **sys**; in effect, this allows **uucico** to jump to a given portion of **sys** based upon the identity of the system that is attempting to log in. In this case, the *remotesystem* arguments will not be used.

callback true|false
If **true**, this command tells **uucico** to hang up when the given remote system calls, and call it back. This is a security measure, to protect your system from being penetrated by remote systems. The default is **false**.

called-chat '' '' \r\d\r in:--in: nuucp word: public word: *serialnumber*
called-chat-fail *string*
called-chat-program *program arguments*
called-chat-seven-bit true|false
called-chat-timeout *seconds*
These commands control how a remote system logs into your system. They are analogous to the commands **chat**, **chat-fail**, **chat-program**, **chat-seven-bit**, and **chat-timeout**, and are structured just like them.

Note that **called-chat** the rest of these commands are invoked after protocol negotiation has been completed between **uucico** on your system and its counterpart on the remote system, but before data exchange has begun. How this chat sequence dovetails with the conversation that COHERENT has with the remote system when it logs into your system depends upon a number of factors, in particular whether COHERENT or **uucico** controls the port in question. It is customary to let COHERENT control logging in through serial ports, as these ports can be used by interactive users as well as by UUCP sessions, while **uucico** usually is allowed to control its well-known TCP port (540). However, **called-chat** can be used to perform special tasks on normal serial lines, such as put the modem into a special state that is required by a given remote site's hardware.

Time Strings and Time Commands

Many of the commands that you can use in *sys* commands use a special kind of string, the *time string*, to specify a range of time. The following describes the structure of a time string.

Each simple time string begins with a token that sets the day of the week. You can use any one of the following values:

Su	Sunday only
Mo	Monday only
Tu	Tuesday only
We	Wednesday only
Th	Thursday only
Fr	Friday only
Sa	Saturday only
Wk	Every week (Monday through Friday)
Any	Every day of the week

You can name more than one day of the week in a time string: just use commas to separate entries.

After the day of the week comes a range of hours and minutes The beginning and ending times are separated by a hyphen. Military time is used, i.e., hour 0 (midnight) through hour 23 (11 PM). **uucico** uses the local time on your system. The range of time can may cross midnight; for example **2300-0700** indicates 11 PM to 7 AM the following day.

If no time is given, any time applies. The word **Never** in place of the time string indicates that this remote system is never to be contacted. You should use this setting for systems that contact you but which you never contact.

You can specify more than one day/time combination in a time string; use commas to separate entries.

The following gives examples of time strings:

Wk2305-0855,Sa,Su2305-1655
Weekdays from 11:05 PM to 8:55 AM the following day; any time on Saturday; and Sunday from 11:05 PM to 4:55 PM the following day.

Wk0905-2255,Su1705-2255
Weekdays from 9:05 AM to 10:55 PM, and Sunday from 5:05 PM to 10:55 PM. The remote system cannot be called on Saturday.

The following commands control when *remotesystem* is contacted:

time *timestring* **[***retry***]**
Specify when your system can call *remotesystem*. *timestring* gives a time string; the section **Time Strings**, above, describes how to construct one. *retry*, if used, defines how long to wait before your system attempt to call *remotesystem* again. The default time for each *remotesystem* is **Never**.

The optional argument *retry* sets many minutes your system will wait before it attempts to recontact *remotesystem*, should a call made during *timestring* fail. If *retry* is not defined, **uucico** uses an exponentially increasing retry time: after each failure the next retry period is longer.

The description of *remotesystem* can contain multiple **time** commands. **uucico** will call *remotesystem* if the current time matches the time defined by any of them.

timegrade *grade timestring* **[retry]**
This command tells **uucico** to call *remotesystem* only if a file with a grade greater than or equal to *grade* is awaiting transfer to that system.

grade gives the grade of file to await. It is a single letter or digit, from '0' to '9', 'A' to 'Z', and 'a' to 'z', in this order from highest grade to lowest.

timestring gives the period of time to which this command applies. *retry* gives the length of time, in minutes, that your system must wait before it recontacts *remotesystem* should a call made during *timestring* fail.

The command **time** is equivalent to the command **timegrade** with a grade of 'z', which permits all jobs to be run. The command **uucico -S** overrides *grade*; the command **uucico -s** does not.

The *grade* applies only to calls made to *remotesystem*, not to calls that it makes to you.

The description of *remotesystem* can have multiple **timegrade** commands.

The command **call-timegrade**, described below, complements this command.

call-timegrade *grade timestring*
This command tells **uucico** to ask *remotesystem* to execute only the jobs with a grade of *grade* or higher, should it call *remotesystem* during the period of time defined in *timestring*. This commands complements the command **timegrade**: while **timegrade** limits what your system does with the remote system, **call-timegrade** attempts to limit what *remotesystem* does to your system.

grade gives the grade of the job to send. It is be a single letter or digit, from '0' to '9', 'A' to 'Z', and 'a' to 'z', in this order from highest grade to lowest. *timestring* gives the period of time to which this command applies. It is a time string, as defined above.

The description of a *remotesystem* can contain multiple **call-timegrade** commands.

Please note that not every implementation of UUCP will cooperate in setting grades to its jobs. If this command does not appear, or if no time string matches, the remote system can send whatever grade of work it chooses.

Retries and Waiting

The following commands define how often **uucico** will try to do something, and how long it will wait for a particular event to happen.

max-retries *retries*
Recontact a *remotesystem* no more than than *retries* times during any time time period. The default number of retries is 26.

success-wait *seconds*
Wait *seconds* before recontacting a *remotesystem* after a successful call. This limits the number of times a *remotesystem* will be contacted during a given time period. The default is zero.

Ports and Telephones

The following commands govern how **uucico** selects a port and telephones *remotesystem*.

address *ip_address* | *domain_name*
Name a remote system to contact a TCP/IP network. This command can name the remote system to contact either by its domain name (e.g., **lepanto.com**) or its IP address (e.g., 199.3.32.100). Note that if the port named by **port** command is not a TCP port, **uucico** ignores this command.

baud *speed*
speed *speed*
Set the speed (or "baud rate") at which to call *remotesystem*. This tells **uucico** to try every port defined in file **/usr/lib/uucp/port** until it finds an unlocked port that runs at *speed*.

If the description of *remotesystem* contains both the **baud** and **port**, **uucico** uses both when it selects a port.

If you wish to try multiple speeds when contacting a *remotesystem*, you must embed each **baud** command in its own set of alternate commands.

uucico does not use a default speed. The command

```
baud 0
```

tells **uucico** to use the "natural" speed of a port (whatever that is), and override and overrides any **baud** or **speed** commands that appear in the global defaults.

To place a call to a *remotesystem*, its description (or the global defaults) must name a port through which to dial out, either with **baud** or with the command **port** (described below).

port *portname*
Name or describe the port through which to contact *remotesystem*.

If used with only one argument, **uucico** assumes that that string names a port defined in the file **/usr/lib/uucp/port**. *portname* may point to more than one physical device; **uucico** tries each in turn until it finds one that is unlocked.

If used with more than one string, **uucico** assumes that the strings define a port, in the same way as done in the file **port**.

To place a call to a *remotesystem*, its description (or the global defaults) must name a port through which to dial out, either with **port** or with the command **baud** (described above).

phone *number*
Give the telephone number of *remotesystem*. An '=' character in the telephone number tells **uucico** to wait for a secondary dial tone. A '-' character tells **uucico** to pause for one second while dialing

The description of a *remotesystem* can have more than one **phone** command, one for each number at which you can call that *remotesystem*. If you want your system to telephone *remotesystem*, then its description must contain at least one **phone** command.

Protocols and Protocol Variables

The command

protocol *codes*

names the communication protocols to use with *remotesystem*. *code* must one or more lower-case letters, each of which names a protocol. If more than one protocol is named, **uucico** considers them in the order in which you give them.

uucico recognizes the following protocol codes:

t
e These protocols perform no checking at all. They are intended to be used over a communication path that has end-to-end reliability, e.g., TCP. **uucico** will consider them only when it is talking to a TCP port that is both reliable and eight-bit.

i This is a bidirectional protocol; that is, your system and *remotesystem* can both send and receive simultaneously. It requires an eight-bit connection. This protocol is preferred for a serial connection, as it offers the fastest transmission of data.

g This is the first, and the commonest UUCP protocol. Every implementation of UUCP supports this protocol; some support no other. It requires an eight-bit connection. For a detailed description of how this protocol works, see the article by Steven Baker, cited below.

G This is the System V Release 4 version of the **g** protocol.

a This mimics the Z-Modem protocol. It requires an eight-bit connection; but unlike the **g** and **i** protocols, it works even if certain control characters cannot be transmitted. (Code for this protocol was contributed by Doug Evans.)

j This is a variant of the **i** protocol, which can avoid certain control characters. The set of characters it avoids can be set by a parameter. It is useful over a eight-bit connection that will not transmit certain control characters.

f This protocol supports X.25 connections. It checksums each file as a whole, so any error causes the entire file to be retransmitted. It requires a reliable connection, but uses only seven-bit transmissions. It is a streaming protocol; therefore, you can use it with a serial port, but the port must be completely reliable and flow controlled.

If you do not use the **protocol** command to specify a protocol, **uucico** considers the protocols in the order given above, and chooses one based on the characteristics of the port and the dialer specified in the files **/usr/lib/uucp/port** and **/usr/lib/uucp/dial**. The port and dial must meet the requirements of a protocol before **uucico** will consider it during negotiation with *remotesystem*.

If neither the **seven-bit** nor the **reliable** command is used, **uucico** will use the **i** protocol (subject, of course, to what is supported by the remote system; you cannot assume that all systems support the **i** protocol). No current protocol can be used with a port for which you have specified **seven-bit true** and **reliable false**. You must use the **protocol** command for the system, or **uucico** will select no protocol at all. (The only reasonable choice would be **protocol f**.) You can use the command

protocol-parameter *protocol parameter* **[***argument ...* **]**

to modify a protocol's default parameters.

The **i** protocol recognizes the following parameters:

window *size*
Request that *remotesystem* use a *size* window, between one and 31, inclusive. The default is 16.

packet-size *size*
Request that *remotesystem* use a packet of *size* bytes, between one and 4,095, inclusive. The default is 1,024.

remote-window *size*
Ignore the window size requested by *remotesystem*, and instead us a window of *size*. The default is zero, which means that the request of *remotesystem* is honored.

remote-packet-size *size*
Ignore the packet size requested by *remotesystem*, and instead use a packet of *size* bytes. The default is zero, which means that the request of *remotesystem* is honored.

sync-timeout *seconds*
Wait *seconds* for a SYNC packet from *remotesystem*. The default is ten.

sync-retries *number*
Resend a SYNC packet *number* times before giving up. The default is six.

timeout *seconds*
Wait *seconds* for an incoming packet before sending a NAK (negative acknowledgement) The default is ten.

retries *number*
Resend a packet or negative acknowedgement *number* times before giving and closing the connection. The default is six.

errors *number*
Quit after *number* errors have occurred. The default is 100.

error-decay *number*
Decrease the count of errors by one after receiving *number* packets. This keeps occasional errors from accumulating during a long conversation, and so aborting what is actually a successful transmission. The default is ten.

ack-frequency *number*
Send an acknowledgement after receiving *number* packets. By default, this is set to half the requested size of the window.

The protocols **g** and **G** recognize the following parameters:

window *size*
Request that *remotesystem* use a *size* window, between one and seven, inclusive. The default is seven.

packet-size *size*
Request that *remotesystem* use a packet size of *size* bytes. **size** must be a power of two, between 32 and 4,096, inclusive. The default is 64, which is the only packet size supported by many older UUCP packages.

startup-retries *number*
Retry the entire initialization sequence *number* times before quitting. The default is eight.

init-retries *number*
Retry one phase of the initialization sequence *number* times before quitting. The default is four.

init-timeout *seconds*
Wait for *seconds* before timing out one phase of the initialization sequence. The default is ten.

retries *number*
Resend a packet or a request for a packet *number* times before quitting. The default is six.

timeout *seconds*
Wait for *seconds* for a packet or an acknowledgement before timing out. The default is ten.

garbage *number*
Drop the connection after receiving *number* unrecognized characters. *number* must be larger than the packet size. The default is 10,000.

errors *number*
Quit after *number* errors have occurred. Errors include malformed packets, out-of-order packets, bad checksums, and packets rejected by the remote system. The default is 100.

error-decay *number*
Decrease the count of errors by one after receiving *number* packets. This keeps occasional errors from accumulating during a long conversation, and so aborting what is actually a successful transmission. The default is ten.

remote-window *size*
Ignore the window size requested by *remotesystem*, and instead us a window of *size*. The default is zero, which means that the request of *remotesystem* is honored.

remote-packet-size *size*
Ignore the packet size requested by *remotesystem*, and instead use a packet of *size* bytes. The default is zero, which means that the request of *remotesystem* is honored.

short-packets true | false
If **true**, optimize transmission by sending shorter packets when there is less data to send. This confuses some UUCP packages; when connecting to such a package, this parameter must be set to **false**. The default is **true** for the **g** protocol and **false** for the **G** protocol.

The **a** protocol mimics the Z-modem protocol. It supports the following parameters: All take numeric arguments, except for **escape-control**, which takes a Boolean argument:

timeout *seconds*
Wait *seconds* for a packet before timing out. The default is ten.

retries *number*
Resend a packet *number* times before quitting. The default is ten.

startup-retries *number*
Retry sending the initialization sequence *number* times before quitting. The default is four.

garbage *number*
Drop the connection after receiving *number* unrecognized "garbage" characters. *number* must be larger than the packet size. The default is 2,400.

send-window *number*
Send *number* characters before waiting for an acknowledgement. The default is 1,024.

escape-control true | false
If **true**, **uucico** can use the protocol over a connection that does not transmit certain control characters, such as **XON** or **XOFF**. The connection must still transmit eight-bit characters other than control characters. The default is **false**.

The **j** protocol can be used over an eight-bit connection that will not transmit certain control characters. It accepts the same parameters as the **i** protocol, plus the following:

avoid *string*
Avoid every character defined in *string*. *string* can contain escape sequences, as defined above for the chat script. Each character must be a non-printable ASCII character (i.e., ASCII values less than 32 or greater than 126). Each must be defined using using the escape sequence *****DDD*, where *DDD* gives three octal digits.

The default value is **\021\023** (i.e., **XON** and **XOFF**). If the package is configured to use **HAVE_BSD_TTY**, then you may have to avoid **\377** as well.

The **f** protocol is intended for use with error-correcting modems only. It checksums each file as a whole, so any error causes the entire file to be retransmitted. It recognizes the following parameters:

timeout *seconds*
Wait *seconds* before timing out. The default is 120.

retries *number*
Retry sending a file *number* times before quitting. The default is two.

The protocols **t** and **e** recognize the following parameter:

timeout *seconds*
Wait *seconds* before timing out. The default is 120.

Note that the command **protocol-parameter** can be used in files **/usr/lib/uucp/dial** and **/usr/lib/uucp/port** as well as in **sys**. In case of a conflict between the entries in these files, the entries in **dial** takes precendence; then those in **port**. The entries in **sys** have lowest precedence.

File Transfers

The following commands help to control the transfer of files.

send-request yes | no
Set whether *remotesystem* can request files from your system. The default is **yes**, that is, *remotesystem* may request files.

receive-request yes | no
Set whether *remotesystem* can send files to your system. The default is **yes**, that is, *remotesystem* may send files.

request yes | no
This combines the commands **send-request** and **receive-request** into one.

call-transfer yes | no
Set whether your system may transfer files to *remotesystem* when it calls *remotesystem*. The default is **yes**.

called-transfer yes | no
Set whether your system may transfer files to *remotesystem* when *remotesystem* calls your system. The default is **yes**.

LEXICON

transfer yes|no
This combines commands **call-transfer** and **called-transfer** into one.

call-local-size *number timestring*
Send or receive no file larger than *number* bytes when your system calls *remotesystem* during the time defined in *timestring*. You can use this command to help limit the length of a call made during times when toll charges are higher. The description of a system may contain multiple **call-local-size** commands, one for each period during which you wish to limit activity. The default is to have no limit.

Please note that the size-control commands, are guaranteed only to limit the size of files being sent by your system. The size of files being sent from *remotesystem* can be checked if the other system is running the Taylor UUCP package. Other UUCP packages do not understand a maximum-size request, nor do they inform this package of the size of the files they are sending.

call-remote-size *number timestring*
Limit to *number* bytes the size of a file that can be fetched by remote request (either by your system on *remotesystem* or by it on your system) when your system calls *remotesystem* during the time defined in *timestring*. The description of a *remotesystem* can contain multiple **call-local-size** commands, one for each period during which you wish to limit activity. The default is to have no limit.

called-local-size *number timestring*
Send or receive no file larger than *number* bytes when *remotesystem* calls your system during the time defined in *timestring*. The description of a system may contain multiple **call-local-size** commands, one for each period during which you wish to limit activity. The default is to have no limit.

called-remote-size *number timestring*
Limit to *number* bytes the size of a file that can be fetched by remote request (either by your system on *remotesystem* or by it on your system) when *remotesystem* calls your system during the time defined in *timestring*. The description of a *remotesystem* can contain multiple **call-local-size** commands, one for each period during which you wish to limit activity. The default is to have no limit.

local-send *directorylist ...*
Limit to *directorylist* the directories from which your system can send files to *remotesystem*. Each directory in *directorylist* must be separated by white space. You can use a tilde '~' for the public directory, i.e., **/usr/spool/uucppublic**. Listing a directory within *directorylist* lets your system send all files within that directory and its subdirectories.

Prefixing a directory with an exclamation point '!' specifically excludes it and its subdirectories from being sent. For example, the command

```
local-send /v/fwb !/v/fwb/Personal
```

means that your system can send all files in directory **/v/fwb** to *remotesytem* except for the files in directory **/v/fwb/Personal**.

uucico reads *directorylist* from left to right, and the last directory to apply takes effect. Therefore, you should list directories from top down. The default is the root directory, i.e., your system can send any file to *remotesystem*.

remote-send *directorylist*
Limit to *directorylist* the directories from which *remotesystem* can request files. The default is **/usr/spool/uucppublic**.

local-receive *directorylist*
Limit to *directorylist* the directories into which your system can write files requested from *remotesystem*. The default is **/usr/spool/uucppublic**.

remote-receive *directorylist*
Limit to *directorylist* the directories on *remotesystem* into which your system can write files. The default is **/usr/spool/uucppublic**. This command cannot override permissions that *localsystem* has granted to your system.

forward-to *systemlist*
Limit the systems to which your system will forward files to those named in *systemlist*. A *systemlist* of **ANY** lets *remotesystem* forward files through your system to any system it wants. The default is not to permit forwarding to other systems. Note that if you permit *remotesystem* to execute the command **uucp** on your system, it effectively has permission to forward to any system.

forward-from *systemlist*
Limit the systems from which *remotesystem* request files through your system to those named in *systemlist*. A *systemlist* of **ANY** lets *remotesystem* request files from any system. The default is not to permit *remotesystem* to request files from anywhere. Note that if you permit *remotesystem* to execute the command **uucp** on your system, it effectively has permission to fetch files through your system from any other system.

forward *systemlist*
This command combines the commands **forward-to** and **forward-from**.

Miscellaneous Commands

The following gives miscellaneous commands that can be used in **sys**:

sequence yes | no
If **true**, this commands tells **uucico** to use the conversation sequencing for *remotesystem*. This means that if somebody impersonates *remotesystem* and logs into your system, that fact will be discovered the next time *remotesystem* actually calls. The default is **false**.

command-path *path*
Limit to *path* the directories that a command file forwarded from *remotesystem* can search for commands to execute. The directories named in *path* must be separated by white space.

commands *commandlist*
Limit the commands that *remotesystem* can execute on your system to those named in *commandlist*. A *commandlist* **ALL** lets **remotesystem** execute all programs on your system. The default is **rnews rmail**.

free-space *number*
This command tells **uucico** always to leave free *number* bytes of space in a file system. This command ensures that **uucico** will not permit *remotesystem* to fill up your file system. If an incoming file is too large to leave *number* bytes free on the file system, **uucico** refuses the file or aborts its downloading.

Note that not every version of UUCP supports this.

pubdir *directory*
Name the public directory available to remote UUCP systems. The default is **/usr/spool/uucppublic**.

debug *activitylist*
Log each UUCP activity named in *activitylist* when talking with *remotesystem*. These logs can help you debug problems with **uucico** and **cu**. **uucico** recognizes the following activities:

abnormal	**chat**	**handshake**
uucp-proto	**proto**	**port**
config	**spooldir**	**execute**
incoming	**outgoing**	

none tells **uucico** to log nothing.

max-remote-debug *typelist*
Limit to *typelist* the types of debugging that *remotesystem* can request on your system. This command is designed to stop *remotesystem* from filling your disk with debugging information.

Defaults

The following gives the default settings for all systems. You should regard these as appearing at the head of **/usr/lib/uucp/sys**, even though they do not explicit appear in that file:

```
time Never
chat "" \r\c ogin:-BREAK-ogin:-BREAK-ogin: \L word: \P
chat-timeout 10
callback n
sequence n
request y
transfer y
local-send /
remote-send ~
local-receive ~
remove-receive ~
commands rnews rmail
max-remote-debug abnormal,chat,handshake
```

Example

The following gives the entry in **/usr/lib/uucp/sys** for the system **mwcbbs**, which is the Mark Williams bulletin board:

```
system mwcbbs
time Any
baud 2400
port MODEM
phone 17085590412
chat "" \r\d\r in:--in: nuucp word: public word: serialnumber
protocol g
protocol-parameter g window 3
protocol-parameter g packet-size 64
myname bbsuser
request yes
transfer yes
remote-send /usr/spool/uucppublic /tmp
remote-receive /usr/spool/uucppublic /tmp
commands rmail uucp
```

The following describes each command in detail:

system mwcbbs
Name the sytem being described, in this case **mwcbbs**.

time Any
Set the time during your system can contact **mwcbbs**, in this case any time.

baud 2400
Set the speed at which your system can contact **mwcbbs**; here 2400 baud.

port MWCBBS
Set the port through which your system can dial out to **mwcbbs**; here, port **MWCBBS**. This port is defined in the file **/usr/lib/uucp/port**; for details on this file and how to modify its data, see the Lexicon entry for **port**.

phone 17085590412
This gives the telephone number of **mwcbbs**.

chat '' '' \r\d\r in:--in: nuucp word: public word: *serialnumber*
Give the chat script with which your system logs into **mwcbbs**. See the section on chat scripts, above, for details on how to interpret this command.

protocol g
Use the **g** protocol.

protocol-parameter g window 3
Set the window used with protocol **g** to three.

protocol-parameter g packet-size 64
Set the size of the packet used with protocol **g** to 64 bytes.

myname bbsuser
Identify yourself to **mwcbbs** as user **bbsuser**.

request yes
Let **mwcbbs** send files to your system, and request files from your system. Setting this to **no** would forbid **mwcbbs** to do so.

transfer yes
Permit files to be transferred from your system to **mwcbbs**, and vice versa, regardless of whether your system calls **mwcbbs** or vice versa.

remote-send /usr/spool/uucppublic /tmp
Permit **mwcbbs** to request files only from directories **/usr/spool/uucppublic** and **/tmp**.

remote-receive /usr/spool/uucppublic /tmp
Limit the directories into which your system will write files requested from **mwcbbs** to **/usr/spool/uucppublic /tmp**.

commands rmail uucp
Limit the commands that **mwcbbs** can execute on your system to **rmail** and **uucp**.

See Also

Administering COHERENT, dial, port, UUCP
Baker, S.: From UUCP to eternity. *UNIX Review*, April 1993, pp. 15-26. *Summarizes the history of UUCP and describes the working of the* **g** *protocol.*

Notes

Only the superuser **root** can edit **/usr/lib/uucp/sys**.

The file **sys** supports many commands in addition to the ones described here. This article describes only those commands that might be used in typical UUCP connections. For more information, see the original Taylor UUCP documentation, which is in the archive **/usr/src/alien/uudoc104.tar.Z**.

sysconf() — System Call (libc)

Get configurable system variables
#include <unistd.h>
long sysconf(*name***)**
int *name***;**

sysconf() returns the value of the system limit or option identified by *name*.

In the following table, the left column gives a symbolic constant to which *name* can be set, and the right column gives the corresponding system variable (as defined in **<limits.h>** and **<unistd.h>**) that **sysconf()** reads and returns:

Name	*Variable*
_SC_ARG_MAX	**ARG_MAX**
_SC_CHILD_MAX	**CHILD_MAX**
_SC_CLK_TCK	**CLK_TCK**
_SC_NGROUPS_MAX	**NGROUPS_MAX**
_SC_OPEN_MAX	**OPEN_MAX**
_SC_PASS_MAX	**PASS_MAX**
_SC_JOB_CONTROL	**_POSIX_JOB_CONTROL**
_SC_SAVED_IDS	**_POSIX_SAVED_IDS**
_SC_VERSION	**_POSIX_VERSION**

The following describes the values returned in more detail:

ARG_MAX
Maximum number of bytes that can be occupied by a process's argument list and environment.

CHILD_MAX
Number of processes a user can run simultaneously.

CLK_TCK
Length of a clock tick, in microseconds.

NGROUPS_MAX
The maximum number of groups to which a user can belong, in addition to her primary group.

OPEN_MAX
The number of files a process can have open simultaneously.

PASS_MAX
The maximum length of a password. Please note that the constant **_SC_PASS_MAX** is defined only for programs compiled for UNIX System V release 4.

_POSIX_JOB_CONTROL
This is a Boolean flag that indicates whether the operating system supports the POSIX job-control functions.

_POSIX_SAVED_IDS
This is a Boolean flag that indicates whether the operating system permits each process to have a saved set-user ID and a saved set-group ID.

_POSIX_VERSION
This is a long integer that encodes the four-digit year and two-digit month of approval for the version of the POSIX standard supported by the operating system. For example, 199009L indicates the version approved in September of 1990.

The value of variable **CLK_TCK** can vary; you should not assume that it is a compile-time constant.

If *name* is an invalid value, **sysconf()** returns -1 and set **errno** to an appropriate value. If **sysconf()** fails due to a value of *name* that is not defined on the system, it returns -1 without setting **errno**.

Example

At the time of this writing (August 1994), the program

```
#include <unistd.h>
main()
{
      printf("_SC_ARG_MAX: %d\n", sysconf(_SC_ARG_MAX));
      printf("_SC_CHILD_MAX: %d\n", sysconf(_SC_CHILD_MAX));
      printf("_SC_CLK_TCK: %d\n", sysconf(_SC_CLK_TCK));
      printf("_SC_NGROUPS_MAX: %d\n", sysconf(_SC_NGROUPS_MAX));
      printf("_SC_OPEN_MAX: %d\n", sysconf(_SC_OPEN_MAX));
      printf("_SC_JOB_CONTROL: %d\n", sysconf(_SC_JOB_CONTROL));
      printf("_SC_SAVED_IDS: %d\n", sysconf(_SC_SAVED_IDS));
      printf("_SC_VERSION: %d\n", sysconf(_SC_VERSION));
}
```

returns the following values:

```
      _SC_ARG_MAX: 5120
      _SC_CHILD_MAX: 25
      _SC_CLK_TCK: 100
      _SC_NGROUPS_MAX: 32
      _SC_OPEN_MAX: 60
      _SC_JOB_CONTROL: 0
      _SC_SAVED_IDS: 1
      _SC_VERSION: 199009
```

See Also

libc, unistd.h
POSIX Standard, §4.8.1

Notes

Programs can use the appropriate **#ifndef** guards to control whether they use **sysconf()** or a symbol from **<limits.h>** for each kind of limit. For example:

```
      #include <unistd.h>
      #include <limits.h>
```

```
      #ifdef      _SC_OPEN_MAX
            max = sysconf (_SC_OPEN_MAX);
      #elif defined (OPEN_MAX)
            max = OPEN_MAX;
      #else
      /* either complain, or make some rational assumption, e.g. */
      #error      Open file descriptor limits cannot be determined
      #endif
```

sysi86() — System Call (libc)

Identify parts within Intel-based machines
#include <sys/sysi86.h>
int sysi86(*hardware***,** *type***)**
int *hardware***,** ******type***;**

The system call **sysi86()** identifies parts within Intel-based computers. *hardware* names the machine part that you wish to identify; you should always use one of the constants defined in header file **<sys/sysi86.h>**. *type* point to the **int** into which **sysi86()** writes an identifying code.

sysi86() returns -1 if it was unable to read your machine. It returns a value other than -1 if it succeeds in reading your machine.

Example

The following program identifies the type of floating-point processor in your machine.

```
#include <sys/sysi86.h>

#ifndef     FP_NO
/*
 * The following header may be needed to get the FP_... constants on some
 * other implementations of the iBCS2 specification; while the sysi86()
 * system call and the SI86FPHW constant are part of the iBCS2 specification,
 * the FP_... constants and the <sys/fp.h> header are not.
 */
#include <sys/fp.h>
#endif

const char *
floating_point_provider ()
{
      int fp_type;

      if (sysi86 (SI86FPHW, & fp_type) == -1)
            return "unable to retrieve FP type";

      switch (fp_type) {

      case FP_NO:
            return "no FP hardware or emulation available";

      case FP_SW:
            return "software emulation of FP hardware";

      case FP_287:
            return "80287 hardware FP";

      case FP_387:
            return "80387 or 80486DX hardware FP";

      default:
            return "unknown floating-point provider";
      }
}

main()
{
      printf("%s\n", floating_point_provider());
}
```

See Also

libc, sysi86.h

Notes

At present under COHERENT, this system call can interrogate a machine only for the type of its floating-point processor.

system() — General Function (libc)

Pass a command to the shell for execution
#include <stdlib.h>
int system(*commandline***) char** ******commandline***;**

system() passes *commandline* to the shell **sh**, which loads it into memory and executes it. **system()** executes commands exactly as if they had been typed directly into the shell. **system()** may be used by commands such as **ed**, which can pass commands to the COHERENT shell in addition to processing normal interactive requests.

Example

This example uses **system** to list the names of all C source files in the parent directory.

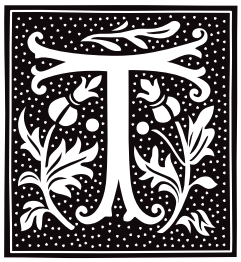
```
#include <stdio.h>
#include <stdlib.h>

main()
{
      system("cd .. ; ls *.c > mytemp; cat mytemp");
}
```

See Also

exec fork(), libc, popen(), stdlib.h, wait()
ANSI Standard, §7.10.4.5

Diagnostics

system() returns the exit status of the child process, in the format described in **wait()**: exit status in the high byte, signal information in the low byte. Zero normally means success, whereas nonzero normally means failure. This, however, depends on the *command*. If the shell is not executable, **system()** returns a special code of octal 0177.



tail — Command

Print the end of a file
tail [+*n***[bcfl]] [***file***]**
tail [-*n***[bcfl]] [***file***]**

tail copies the last part of *file*, or of the standard input if none is named, to the standard output.

The given *number* tells **tail** where to begin to copy the data. Numbers of the form *+number* measure the starting point from the beginning of the file; those of the form *-number* measure from the end of the file.

A specifier of blocks, characters, or lines (**b**, **c**, or **l**, respectively) may follow the number; the default is lines. If no *number* is specified, a default of -10 is assumed.

The **-f** option opens the tail of a file, and then displays new material as it is added to a file. This command lets you watch a file as it is being built, such as by **nroff**. Note that when **tail** is invoked with this option, it does not exit; therefore, when you wish to exit, type the interrupt character (usually **<ctrl-C>**).

See Also

commands, **dd, egrep, head, sed**

Notes

Because **tail** buffers data measured from the end of the file, large counts may not work.

tan() — Mathematics Function (libm)

Calculate tangent
#include <math.h>
double tan(*radian***) double** *radian***;**

tan() calculates the tangent of its argument *radian*, which must be in radian measure.

Example

The following program implements the Fresnel equation, which computes the percentage of light or energy reflected from perfect glass, based on the angle of incidence. It is by Dmitry Gringauz (dmitry@golem.com). Be sure to compile it with the options **-f** and **-lm**.

```
#include <math.h>
#include <stdio.h>

double deg_to_rad(deg)
double deg;
{
      return deg*PI/180.0;
}

double rad_to_deg(rad)
double rad;
{
      return rad*180.0/PI;
}
```

```
main()
{
      double i=0.0; /* incidence angle */
      double Ra=0.0; /* angle of refraction */
      double Rho=0.0; /* % reflection of the beam */
      double Ri=1.52; /* refractive index of glass */

      printf("\tAngle\t\tRho\n");
      printf("\t-----\t\t---\n");

      for (i = 5.0; i <= 90.0; i = i+5.0) {
            double x = 0.0, y = 0.0; /* temporaries */

            /* find the angle of refraction */
            Ra = rad_to_deg(asin( sin(deg_to_rad(i)) / Ri));

            /* makes sense to calculate these only once */
            x = deg_to_rad(i - Ra);
            y = deg_to_rad(i + Ra);

            /* find out percent of reflected energy */
            Rho = pow(sin(x), 2.0) / pow(sin(y), 2.0) +
                        pow(tan(x), 2.0) / pow(tan(y), 2.0);
            Rho = Rho/2.0*100.00;
            printf("\t%f\t%f\n", i, Rho);
      } /* for */
} /* main */
```

See Also

libm, tanh()
ANSI Standard, §7.5.2.7
POSIX Standard, §8.1

Diagnostics

tan() returns a very large number where it is singular, and sets **errno** to **ERANGE**.

tanh() — Mathematics Function (libm)

Calculate hyperbolic cosine
#include <math.h>
double tanh(*radian***) double** *radian***;**

tanh() calculates the hyperbolic tangent of *radian*, which is in radian measure.

See Also

libm, tan()
ANSI Standard, §7.5.3.3
POSIX Standard, §8.1

Diagnostics

When an overflow occurs, **tanh()** sets **errno** to **ERANGE**.

tape — Technical Information

Magnetic-tape devices

The COHERENT system supports two classes of magnetic-tape devices: *floppy tape*, in which the tape device is plugged into your system's floppy-disk controller; and *SCSI tape*, in which the tape device is plugged into your system's SCSI controller (should it have one). The following gives general remarks on tape devices, then briefly discusses the drivers for tape devices and the block-special files by which you can access them.

Tape Devices

A tape consists of one or more files. Each file, in turn, consists of one or more records and is terminated by a tape mark. Two tape marks terminate the last file. Tape records may vary in length, but cannot exceed 32 kilobytes (16 kilobytes is more practical).

Like other block-oriented devices, tape units can be accessed through a system's *cooked* interface or its *raw* interface. On a cooked device, seeking to any byte offset and reading in any number of bytes is possible. You

cannot read beyond the tape mark at the end of the current file. For block-I/O requests, every record in the file must be 512 bytes long. Write requests must be made in increments of 512 bytes.

A raw device bypasses the buffer cache, so that data are written directly to or from your buffer. One write request generates one tape record, and one read request returns exactly one record. The number of bytes read may be fewer than expected. If the tape mark is read, a count of zero is returned, but the system positions the tape at the start of the next tape file. Seeking on a raw device is ignored, and mounting is not allowed. Raw (or character) requests are usually performed in units much larger than 512 bytes.

A unit cannot be opened if it is off-line or already in use. If tape cartridge within the tape drive is write protected, you cannot open the tape device for writing. Closing the device has varying effects, depending on the device's minor-device number and whether the device was opened for reading or writing. If the tape had been read, the tape is rewound; if the no-rewind device was specified, the tape advances to the next file. In the case of writing, two tape marks are written at the current position and the tape is rewound; if the no-rewind device was specified, two tape marks are written and the tape is positioned between them. When you close a device that had been opened for writing, the tape volume ends at the current position; data beyond this point are undefined.

Hard errors may occur during tape operation. They include detecting the end-of-tape (EOT) reflector, reading an unexpectedly long record, or seeking a cooked tape into a tape mark. After an error, no further operations can be performed on the unit until the program closes the device and you rewind the tape. Soft parity errors may arise due to dirt on the tape, a bad tape, or misaligned heads. If an error occurs on a write, the device may attempt to place the record further along the tape. If the error occurs on a read, the driver simply rescans the record. After several failures, the driver announces a hard error.

Drivers

COHERENT includes two drivers for tape backups:

ft This driver has major number 4, the same as the floppy-disk drive. It works with QIC-40 and QIC-80 drives from Colorado, Archive, Mountain, Summit, and IBM.

hai This is a host adapter-independent SCSI driver, which supports SCSI hard disks as well as tape. This has major device number 13. **hai** works with hard disks from Adaptec, Seagate, and Future Domain. It has been tested with the Archive Viper 60, 150, 250, and 525 SCSI tape devices, and is known to work with them.

Each driver has a number of default behaviors, depending upon how you access it. For details, see the driver's entry in the Lexicon.

Devices

The following names the devices used to access tape drives. For SCSI tape devices, *N* is the SCSI identifier of your tape unit, as set when you installed COHERENT. (To change your suite of SCSI identifiers, you must reconfigure your kernel. For directions on how to do this, see the Lexicon entry for **hai**.)

/dev/rStp*N*	SCSI tape unit *N*, raw device, rewind.
/dev/nrStp*N*	SCSI tape unit *N*, raw device, no rewind.
/dev/xStp*N*	SCSI tape unit *N*, control device.
/dev/rct*N*	QIC-24 tape unit *N*, raw device, rewind.
/dev/nrct*N*	QIC-24 tape unit *N*, raw device, no rewind.
/dev/xct*N*	QIC-24 tape unit *N*, control device.
/dev/ft*N*	QIC-40/80 (floppy tape), rewind.
/dev/nft*N*	QIC-40/80 (floppy tape), no rewind.
/dev/ctmini	Default mini-cartridge device, retensioning.
/dev/rctmini	Default mini-cartridge device, no retensioning.
/dev/xctmini	Default mini-cartridge device, control device.
/dev/mc*N*	Irwin floppy tape, retensioning
/dev/rmc*N*	Irwin floppy tape, no retensioning.
/dev/xmc*N*	Irwin floppy tape, control device.

Installing Tape Devices

To install a SCSI tape device onto your system, do the following:

- Power down your system; then plug the SCSI device into your SCSI board. Do *not* plug the tape device into your SCSI board while your system is powered up, or you will damage your hardware.

- After you have rebooted your system, log in as the superuser **root**.
- **cd** to directory **/etc/conf**.
- Invoke the command **cohtune hai** and set the variable **HAI_TAPE** to the SCSI ID of the tape drive — usually two.
- Invoke the command **idmkcoh** to build a new kernel.
- Reboot your system and invoke the newly built kernel.

To install a floppy-tape device onto your system, do the following:

- If you have not already done so, make sure that the you have updated COHERENT to a version that supports floppy tape, that is, release 4.2.12 or later.
- Power down your system and install the floppy-tape device as described in its manual. Do not attempt to install your device while your system is powered up, or you can damage or destroy your system. Be very careful that DIP switches and jumpers are set correctly. Also, make sure that all cables are seated firmly — it is easy to loosen a connected while installing a tape device.
- Reboot your system into single-user mode. You can do so by typing **<ctrl-C>** while your file system is being checked during the reboot process, or invoke the command

  ```
  /etc/shutdown single 0
  ```

  after the system has checked its file system and rebooted.
- Running from single-user mode, run the script **/etc/conf/ft/mkdev**. If you know that your tape drive uses soft select and know the manufacturer, you can specify these features explicitly. If you know that your tape drive uses hard select and know the unit number (for example, a tape drive that takes the place of a second floppy-disk drive is unit 1), you can specify these features explicitly. If you are not sure of the above, select automatic configuration. The device driver **ft** will try to sense which type of drive you are using.
- Unless you have other tape drives installed, we recommend that you link the no-rewind-on-close floppy-tape device to the default tape device **/dev/tape**.
- While still in single-user mode, run the script **/etc/conf/bin/idmkcoh**. This generates a new kernel that can access the tape drive.
- Reboot your system and invoke the newly built kernel.

Manipulating Tape Devices

The command **tape** manipulates tape devices. With this command, you can rewind a tape, check the status of a tape device, or perform other useful tasks. For details, see its entry in the Lexicon.

Command **ftbad** lets you view and edit the list of bad blocks on a floppy-tape cartridge. For details on how to use this command, use see its entry in the Lexicon.

For details on how to build backups onto tape devices, see the Lexicon entry **backups**.

See Also

Administering COHERENT, backups, ft, ftbad, hai, tape [command]

Notes

Systems with a very slow CPU (e.g., a 16-megahertz 80386SX) may have trouble running the floppy-tape driver **ft** in multi-user mode. The reason is that floppy-tape hardware does not have much intelligence built into it, so the driver must consume many CPU cycles. In such instances, we suggest that you back up your system while in single-user mode (which is a good idea in any case).

tape — Command

Manipulate a tape device
tape command [*count***] [***device***]**

The command **tape** lets you manipulate a tape device. *device* names the tape device to manipulate. If you name no *device* on the command line, **tape** uses the device **T_DEFAULT**; header file **<tape.h>** defines this constant to be device **/dev/tape**. For a list of tape devices, see the overview article for **tape**.

command names the task that you want **tape** to execute, as follows:

erase Erase the tape. SCSI tape only.

retension Retension the tape. This rewinds the tape, then performs a full forward wind, then another rewind. The seek offset is set to zero.

rewind Rewind the tape. This command positions the tape at the beginning of track 0. It resets seek offset (see **seek** and **tell**, below) to zero. If tape is already rewound, this command has no effect.

rfm Move the tape forward to the next file mark; in effect, skip the current file. SCSI tape only.

seek *location* This command has the same effect as if the tape had just been used with no-rewind-on-close, leaving the tape at byte *location*. No tape motion occurs at the time of the command, but the next read or write begins at byte *location* on the tape. Floppy tape only.

status Display various parameters for the tape drive, and for the cartridge being used. Not every tape drive supports every status option. Unsupported features appear as "unavailable". The following gives an example of output from this command:

```
Floppy Tape Status:
      Drive Configuration  = 0x90
            500 Kbits/sec
            Non-Extra-Length Tape
            QIC-80 Mode.
      ROM Version  = 0x85
      Vendor ID  = 0x0146, Make=5, Model=6
      Tape Status Unavailable.
      Drive Status  = 0x65
            drive ready or idle
            cartridge present
            cartridge referenced
            at physical BOT
      Drive Error Status - No Error.
```

Floppy tape only.

tell Display the byte offset that will be in effect the next time the tape is read or written. Floppy tape only.

The related command **ftbad** lets you read and modify the list of bad blocks on a floppy-tape cartridge.

See Also

commands, ftbad, hai, tape

tar — Command

Archiving/backup utility
tar [*options***]** *file ...*

tar is a utility that lets you read, write, and update archives in a machine-independent format. Its name is an abbreviation for *tape archive;* however, **tar** can read/write output to files and floppy disks, as well as to magnetic tape.

tar is now a link to the command **gtar**, which implements tape archiving more robustly than did the version of **tar** shipped with earlier editions of COHERENT. For details on how to use **gtar**, turn to its entry in the Lexicon.

See Also

commands, gnucpio, gtar
POSIX Standard, §10.1.1

tboot — Technical Information

Describe the tertiary bootstrap

Booting is the process of loading COHERENT into memory and setting it into motion. This normally occurs after you have turned on your computer. The term comes from the old expression about pulling one's self up by one's bootstraps.

Booting can be quite involved, and uses a number of files, depending upon the version of COHERENT being booted and the medium from which you are booting it. The subject of this article, **tboot**, is the booting program that performs tertiary booting.

To grasp what is meant by "tertiary booting", consider how the boot sequence works:

1. The BIOS loads the first 512 bytes off of the first hard disk and runs it. This program is called the **master boot**. Mark Williams Company recommends that you use the COHERENT master boot, because it lets you boot off any partition on either of the first two drives.
2. The master boot loads the first 512 bytes off the active partition and runs that. This program is the "secondary boot" program.

   The secondary boot is generally responsible for loading the operating system off the active parition and running it.

Recent releases of COHERENT need a more sophisticated program to load the operating system than can fit into 512 bytes. In these releases of COHERENT, the secondary boot loads a program off the root file system; this program is called the "tertiary boot", or **tboot**.

tboot evaluates the hardware of your computer to provide the operating system (COHERENT) with vital information. This evaluation allows COHERENT to run without modification on a wider range of hardware.

tboot is responsible for loading the operating system kernel. It first looks for a file called **autoboot**, which it then loads. If **autoboot** does not exist, **tboot** prompts you to type in the name of a kernel, e.g., **begin** (during installation) or **coherent**. If you do not remember the name of the kernel you wish to boot, you can type **dir** or **ls** for a list of files in your root file system.

Pressing the spacebar when the prompt is displayed prevents execution of **/autoboot** and causes **tboot** to pause. You can then type the name of an alternate kernel to load (assuming it already resides within the root directory), type **ls** to see a listing of files, or type **info** for a display of hard-drive parameters.

See Also

Administering COHERENT, booting

tcdrain() — termios Macro (termios.h)

Drain output to a device
#include <termios.h>
int tcdrain(*fd***)**
int *fd***;**

The **termios** macro **tcdrain()** waits until all output written to device *fd* has been transmitted. *fd* must have returned by a call to **open()**, and must describe a terminal device.

If all goes well, **tcdrain()** returns zero. If something goes wrong, it returns -1 and sets **errno** to an appropriate value, as follows:

EBADF *fd* is not a valid file descriptor.

EINTR A signal interrupted **tcdrain()**.

ENOTTY
fd does not describe a terminal device.

See Also

termios
POSIX Standard, §7.2.2

tcflow() — termios Macro (termios.h)

Control flow on a terminal device
#include <termios.h>
int tcflow(*fd***,** *action***)**
int *fd***;**
int *action***;**

The **termios** macro **tcflow()** suspends transmission of data to, or receiption of data from, the device described by file descriptor *fd*. When a terminal device is opened, by default neither its input nor its output is suspended. *action* gives the action to take, as follows:

TCOOFF
Suspend output.

TCOON
Restart output.

TCIOFF
Transmit character **STOP**,
which tells the terminal to stop sending data to the system.

TCION Transmit character **START**, which tells the terminal to resume sending data to the system.

These constants are defined in header file **<termios.h>**.

Should all go well, **tcflow()** returns zero. If something goes wrong, it returns -1 and sets **errno** to an appropriate value, as follows:

EBADF *fd* is not a valid file descriptor.

EINVAL
action is not set to an appropriate value.

ENOTTY
fd does not describe a terminal device.

See Also

termios
POSIX Standard, §7.2.2

tcflush() — termios Macro (termios.h)

Flush data being exchanged with a terminal
#include <termios.h>
int tcflush (*fd***,** *queue_selector***)**
int *fd***;**
int *queue_selector***;**

The **termios** macro **tcflush()** discards, or "flushes," data send to or received from the terminal device described by the file descriptor *fd*. *queue_selector* indicates what to do, as follows:

TCIFLUSH
Flush data received but not read.

TCOFLUSH
Flush data written but not transmitted.

TCIOFLUSH
Flush both data written and data read.

These constants are defined in header file **<termios.h>**.

If all goes well, **tcflush()** returns zero. If something goes wrong, it returns -1 and sets **errno** to an appropriate value, as follows:

EBADF *fd* is not a valid file descriptor.

EINVAL
queue_selector is not a proper value.

ENOTTY
fd does not describe a terminal device.

See Also

termios
POSIX Standard, §7.2.2

tcgetattr() — termios Macro (termios.h)

Get terminal attributes
#include <termios.h>
int tcgetattr (*fd***,** *termios_p***);**
int fd;
struct termios *termios_p;

The **termios** macro **tcgetattr()** gets the parameters for the terminal device described by file descriptor *fd*, and stores them in the **termios** structure to which *termios_p* points.

ttcgetattr() can be called from a background process. Please note, however, a foreground process can subsequently change the terminal device's attributes, which renders obsolete the information in *termios_p*.

If all goes well, **tcgetattr()** returns zero. If a problem occurs, it returns -1 and sets **errno** to an appropriate value, as follows:

EBADF *fd* is not a valid file descriptor.

ENOTTY
fd does not describe a terminal device.

See Also

tcsetattr(), termios
POSIX Standard, §7.2.1

tcsendbreak() — termios Macro (termios.h)

Send a break to a terminal
#include <termios.h>
int tcsendbreak(*fd***,** *duration***)**
int *fd***;**
int *duration***;**

The **termios** macro **tcsendbreak()** transmits NUL characters to the terminal device described by file descriptor *fd*.

duration gives the length of time to transmit NUL characters. If *duration* is zero, **tcsnedbreak()** transmits zero-valued bits for at least 0.25 seconds and no more than 0.5 seconds. If *duration* is not set to zero, **tcsendbreak()** sends zero-valued bits for the time specified by the implementation. Under COHERENT, **tcsendbreak()** is a macro defined as follows:

```
#define tcsendbreak(filedes,duration) ioctl(filedes,TCSBRK,0)
```

TCSBRK is defined in header file **<termio.h>**: it transmits break characters for 0.25 seconds. Thus, the argument *duration* is ignored.

If *fd* does not use asynchronous serial data transmission, the implementation defines whether the **tcsendbreak()** function sends data to generate a break condition (as defined by the implementation) or returns without taking any action. Under COHERENT, it does nothing.

If all goes well, **tcsendbreak()** returns zero. If something goes wrong, it returns -1 and sets **errno** to an appropriate value, as follows:

EBADF *fd* is not a valid file descriptor.

ENOTTY
fd does not describe a terminal.

See Also

termios
POSIX Standard, §7.2.2

tcsetattr() — termios Macro (termios.h)

Set terminal attributes
#include <termios.h>
int tcsetattr(*fd***,** *optional_actions***,** *termios_p***)**
int *fd***,** *optional_actions***;**
struct termios **termios_p***;*

The **termios** macro **tcsetattr()** sets the attributes of the terminal device described by file descriptor *fd* to those stored in the **termios** structure to which *termios_p* points.

option_actions defines the manner in which the attributes are set, as follows:

TCSANOW
The attributes are set immediately.

TCSADRAIN
The attributes are set after all data that has been sent to *fd* has been written. Use this when changing parameters that affect output.

TCSAFLUSH
The change occurs after all output send to *fd* has been written: all input received but not read is discarded.

These constants are defined in header file **<termios.h>**.

If all goes well, **tcsetattr()** returns zero. If something goes wrong, it returns -1 and sets **errno**

EBADF *fd* is not a valid file descriptor.

EINVAL
optional_actions is not a proper value, or an attempt was made to change an attribute in the **termios** structure to an unsupported value.

ENOTTY
fd does not describe a terminal.

See Also

termios
POSIX Standard, §7.2.1

tee — Command

Copy input to multiple output streams
tee [-a] [-i] [*file ...***]**

tee reads from standard input, usually a pipe, and writes to the standard output, usually a pipe. **tee** also writes a copy of the input data to each *file* specified.

The **-a** flag tells **tee** to append data to each *file*, analogous to the shell construct "**>>***file*". Otherwise, it creates each *file,* analogous to the construct "**>***file*".

The flag **-i** means ignore interrupts.

See Also

commands, ksh, sh

telldir() — General Function (libc)

Return the current position within a directory stream
off_t telldir (*dirp***)**
DIR **dirp***;*

The COHERENT function **telldir()** is one of a set of COHERENT routines that manipulate directories in a device-independent manner. It returns the current position within the directory stream pointed to by *dirp*.

If an error occurs, **telldir()** exits and sets **errno** to an appropriate value.

See Also

closedir(), dirent.h, getdents(), libc, opendir(), readdir(), rewinddir(), seekdir(),

Notes

The value returned by **telldir()** should only be used as an argument to *seekdir()*.

telldir() and **seekdir()** are unreliable when directory stream has been closed and reopened. It is best to avoid using **telldir()** and **seekdir()** altogether.

Because directory entries can dynamically appear and disappear, and because directory contents are buffered by these routines, an application may need to continually rescan a directory to maintain an accurate picture of its active entries.

The COHERENT implementation of the **dirent** routines was written by D. Gwynn.

tempnam() — General Function (libc)

Generate a unique name for a temporary file
#include <stdio.h>
char *tempnam(*directory***,** *name***);**
char ****directory*, ****name***;**

tempnam() constructs a unique temporary name that can be used to name a file. *directory* points to the name of the directory in which you want the temporary file written. If this variable is NULL, **tempnam()** reads the environmental variable **TMPDIR** and uses it for *directory*. If neither *directory* nor **TMPDIR** is given, **tempnam()** uses **/tmp**.

name points to the string of letters that will prefix the temporary name. This string should not be more than three or four characters, to prevent truncation or duplication of temporary file names. If *name* is NULL, **tempnam()** sets it to **t**.

tempnam() uses **malloc()** to allocate a buffer for the temporary file name it returns. If all goes well, it returns a pointer to the temporary name it has written. Otherwise, it returns NULL if the allocation fails or if it cannot build a temporary file name successfully.

See Also

libc, mktemp(), TMPDIR, tmpfile(), tmpnam()

TERM — Environmental Variable

Name the default terminal type
TERM=*terminal type*

The environmental variable **TERM** names the type of terminal that you are using. This variable is read by every program that uses the **termcap** or **terminfo** library, to ensure that the correct terminal description is read when the program is invoked. You should set this variable in your **profile**, to ensure that the system understands what type of terminal you use. The file **/etc/profile** sets **TERM** to **ansipc**.

See Also

environmental variables, me, termcap

term — System Administration

Format of compiled terminfo file

Before it can be used, a file of **terminfo** information must be compiled with the command **tic**. It is read by the command **setupterm**.

Once compiled, the binary **terminfo** file is moved into a sub-directory of directory **/usr/lib/terminfo**. To avoid a linear search of a huge COHERENT directory, a two-level scheme is used to name the subdirectories: **/usr/lib/terminfo/***C***/***name*, where *name* names the terminal and *C* is the first character of *name*. For example, the **terminfo** entry for the Wyse 150 terminal is kept in the file **/usr/lib/terminfo/w/wyse150**. Synonyms for a terminal exist as links to the same compiled file.

The binary format of a **terminfo** file has been designed to be the same on all hardware. The file is divided into six parts: header, terminal names, boolean flags, numbers, strings, and string table.

Header

The *header* section begins the file. This section contains the following six short integers:

1. The magic number (octal 0432).
2. The size, in bytes, of the *names* section.
3. The number of bytes in the *boolean* section.
4. The number of short integers in the *numbers* section.
5. The number of offsets (short integers) in the *strings* section.
6. The size, in bytes, of the *string* table.

A *short integer* is two bytes long. Under the **term** file format, 0xFFFF represents -1; all other negative value are illegal. Minus 1 generally means that a capability is missing from this terminal. All short integers are aligned on a short-word boundary.

Names

The *names* section contains the first line of the **terminfo** description, which lists the names for the terminal, each name separated by a vertical bar '|'. The section is terminated with a NUL.

Boolean

The *boolean* section contains the boolean flags for terminals. There is one flag for each boolean capacity recognized by **terminfo**. The flags appear in the order described in the header file **term.h**. Each flag is one byte long, and is set to zero or one, depending upon whether the capacity is absent or present in this terminal. If necessary, this section is ended with a NUL to ensure that the next section begins on an even byte.

Numbers

The *numbers* section is similar to the *flags* section. There is one entry for each numeric capacity recognized by **terminfo**, each capacity being represented by a short integer. A value of -1 indicates that this terminal lacks this capability. Entries appear in the order described in the header file **term.h**.

Strings

The *strings* section also contains one short integer for each string capability recognized by **terminfo**. A value of -1 means that this terminal lacks this capability. Otherwise, the value gives an offset from the beginning of the string table. Entries appear in the order described in the header file **term.h**.

Special characters in **^X** or *****c notation are stored in their interpreted form. Padding information and parameter information are stored intact in uninterpreted form.

String Table

The final section is the *string table*. It contains all the values of string capabilities referenced in the *string* section. Each string is null terminated.

Files

/usr/lib/terminfo/* — Default location of object files

See Also

Administering COHERENT, curses, infocmp, tic, terminfo
Strang, J., Mui, L., O'Reilly, T.: *termcap and terminfo*. Sebastopol, CA: O'Reilly & Associates, Inc., 1991.

Notes

The total compiled file cannot exceed 4,096 bytes. The *name* field cannot exceed 128 bytes.

termcap — System Administration

Terminal-description language
/etc/termcap

termcap is a language for describing terminals and their capabilities. Terminal descriptions are collected in the file **/etc/termcap** and are read by **tgetent** and its related programs to ensure that output to a particular terminal is in a format that that terminal can understand.

COHERENT also supports the terminal-description language **terminfo**. For a description of how these languages differ, see the Lexicon entry for **terminfo**.

Overview

A terminal description written in **termcap** consists of a series of fields, which are separated from each other by colons ':'. Every line in the description, with the exception of the last line, must end in a backslash '\'. Tab characters are ignored. Lines that begin with a '#' are comments. A **termcap** description must not exceed 1,024 characters.

The first field names the terminal. Several different names may be used, each separated by a vertical bar '|'; each name given, however, must be unique within the file **/etc/termcap**. By convention, the first listed must be two characters long. The second name is the name by which the terminal is most commonly known; this name may contain no blanks in it. Other versions of the name may follow. By convention, the last version is the full name of the terminal; here, spaces may be used for legibility. Any of these may be used to name your terminal to the COHERENT system. For example, the name field for the VT-100 terminal is as follows:

```
d1|vt100|vt-100|pt100|pt-100|dec vt100:\
```

Note that the names are separated by vertical bars '|', that the field ends with a colon, and that the line ends with a backslash. Using any of these names in an **export** command will make the correct terminal description available to programs that need to use it.

The remaining fields in the entry describe the *capabilities* of the terminal. Each capability field consists of a two-letter code, and may include additional information. There are three types of capability:

Boolean

This indicates whether or not a terminal has a specific feature. If the field is present, the terminal is assumed to have the feature; if it is absend, the terminal is assumed not to have that feature. For example, the field

am:

is present, **termcap** assumes that the terminal has automatic margins, whereas if that field is not present, the program using **termcap** assumes that the terminal does not have them.

Numeric

This gives the size of some aspect of the terminal. Numeric capability fields have the capability code, followed by a '#' and a number. For example, the entry

co#80:

means that the terminal screen is 80 columns wide.

String capabilities

These give a sequence of characters that trigger a terminal operation. These fields consist of the capability code, an equal sign '=', and the string.

Strings often include escape sequences. A "**\E**" indicates an **<ESC>** character; a control character is indicated with a carat '^' plus the appropriate letter; and the sequences **\b, \f, \n, \r**, and **\t** are, respectively, backspace, formfeed, newline, **<return>**, and tab.

An integer or an integer followed by an asterisk in the string (e.g., '*int**') indicates that execution of the function should be delayed by *int* milliseconds; this delay is termed *padding*. Thus, deletion on lines on the Microterm Mime-2A is coded as:

```
dl=20*^W:
```

dl is the capability code for *delete*, the equal sign introduces the deletion sequence, **20*** indicates that each line deletion should be delayed by 20 milliseconds, and **^W** indicates that the line-deletion code on the Mime-2A is **<ctrl-W>**.

The asterisk indicates that the padding required is proportional to the number of lines affected by the operation. In the above example, the deletion of four lines on the Mime-2A generates a total of 80 milliseconds of padding; if no asterisk were present, however, the padding would be only 20 milliseconds, no matter how many lines were deleted. Also, when an asterisk is used, the number may be given to one decimal place, to show tenths of a millisecond of padding.

Note that with string capabilities, characters may be given as a backslash followed by the three octal digits of the character's ASCII code. Thus, a colon in a capability field may be given by **\072**. To put a null character into the string, use **\200**, because **termcap** strips the high bit from each character.

Finally, the literal characters '**^**' and '****' are given by "**\^**" and "****".

Capability Codes

The following table lists **termcap**'s capability codes. **Type** indicates whether the code is boolean, numeric, or string; a dagger '†' indicates that this capability may include padding, and a dagger plus an asterisk "†*" indicates that it may be used with the asterisk padding function described above.

Variable	*Type*	*Definition*
ae	string†	End alternate set of characters
al	string†*	Add blank line
am	boolean	Automatic margins
as	string†	Start alternate set of characters
bc	string	Backspace character, if not **<ctrl-H>**
bs	boolean	Backspace character is **<ctrl-H>**
bt	string†	Backtab
bw	boolean	Backspace wraps from column 0 to last column
CC	string	Command character in prototype if it can be set at terminal
cd	string†*	Clear to end of display
ce	string†	Clear line
ch	string†	Horizontal cursor motion
cl	string†*	Clear screen
cm	string†	Cursor motion, both vertical and horizontal
co	number	Number of columns
cr	string†*	**<return>**; default **<ctrl-M>**
cs	string†	Change scrolling region (DEC VT100 only); resembles **cm**
cv	string†	Vertical cursor motion
da	boolean†	Display above may be retained
dB	number	Milliseconds of delay needed by **bs**
db	boolean	Display below may be retained
dC	number	Milliseconds of delay needed by **cr**
dc	string†*	Delete a character
dF	number	Milliseconds of delay needed by **ff**
dl	string†*	Delete a line
dm	string	Enter delete mode
dN	number	Milliseconds of delay needed by **nl**
do	string	Move down one line
dT	number	Milliseconds of delay needed by tab
ed	string	Leave delete mode
ei	string	Leave insert mode; use **:ei=:** if this string is the same as **ic**
eo	string	Erase overstrikes with a blank
ff	string†*	Eject hardcopy terminal page; default **<ctrl-L>**
hc	boolean	Hardcopy terminal
hd	string	Move half-line down, i.e., forward 1/2 line feed)
ho	string	Move cursor to home position; use if **cm** is not set
hu	string	Move half-line up, i.e., reverse 1/2 line feed
hz	string	Cannot print tilde '~' (Hazeltine terminals only)
ic	string†	Insert a character
if	string	Name of the file that contains **is**
im	string	Begin insert mode; use **:im=:** if **ic** has not been set
in	boolean	Nulls are distinguished in display
ip	string†*	Insert padding after each character listed
is	string	Initialize terminal
k0-k9	string	Codes sent by function keys 1 through 10 (**k0** = F10)
kb	string	Code sent by backspace key
kd	string	Code sent by down-arrow key
ke	string	Leave "keypad transmit" mode
kh	string	Code sent by home key
kl	string	Code sent by left-arrow key
kn	number	No. of function keys; default is 10
ko	string	Entries for for all other non-function keys
kr	string	Code sent by right-arrow key
ks	string	Begin "keypad transmit" mode

ku . . . . . string . . . . . Code sent by up-arrow key
l0-l9. . . . . string . . . . . Function keys labels if not f0-f9
li. . . . . number. . . . . Number of lines
ll. . . . . string . . . . . Move cursor to first column of last line (**cm** not set)
ma. . . . . string . . . . . Map keypad-to-cursor movement for **vi** version 2
mi . . . . . boolean . . . . . Cursor may be safely moved while in insert mode
ml . . . . . string . . . . . Turn on memory lock for area of screen above cursor
ms. . . . . boolean . . . . . Cursor can be moved while in standout or underline mode
mu . . . . . string . . . . . Turn off memory lock
nc . . . . . boolean . . . . . **<return>** does not work
nd . . . . . string . . . . . Move cursor right non-destructively
nl . . . . . string†* . . . . . Newline character; default is **\n** (*Obsolete*)
ns . . . . . boolean . . . . . Terminal is CRT, but does not scroll
os . . . . . boolean . . . . . Terminal can overstrike
pc . . . . . string . . . . . Pad character any character other than null
PS . . . . . string . . . . . Print start: redirect input to auxiliary port
PN . . . . . string . . . . . Print end: stop redirecting input to auxiliary port
pt . . . . . boolean . . . . . Terminal's tabs set by hardware; may need to be set with **is**
se . . . . . string . . . . . Exit standout mode
sf . . . . . string† . . . . . Scroll forward
sg . . . . . number. . . . . Blank characters left by **so** or **se**
so . . . . . string . . . . . Enter standout mode
sr . . . . . string† . . . . . Reverse scroll
ta . . . . . string† . . . . . Tab character other than **<ctrl-I>**, or with padding
tc . . . . . string . . . . . Similar terminal — must be last field in entry
te . . . . . string . . . . . End a program that uses **cm**
ti . . . . . string . . . . . Begin a program that uses **cm**
uc . . . . . string . . . . . Underscore character and skip it
ue . . . . . string . . . . . Leave underscore mode
ug . . . . . number. . . . . Blank characters left by **us** or **ue**
ul . . . . . boolean . . . . . Terminal underlines but does not overstrike
up . . . . . string . . . . . Move up one line
us . . . . . string . . . . . Begin underscore mode
vb . . . . . string . . . . . Visible bell; may not move cursor
ve . . . . . string . . . . . Exit open/visual mode
vs . . . . . string . . . . . Begin open/visual mode
xb . . . . . boolean . . . . . Beehive terminal (f1=**<esc>**, f2=**<crtl-C>**)
xn . . . . . boolean . . . . . Newline is ignored after wrap
xr . . . . . boolean . . . . . **<return>** behaves like **ce \r \n**
xs . . . . . boolean . . . . . Standout mode is not erased by writing over it
xt . . . . . boolean . . . . . Tabs are destructive

Examples

The following is the **termcap** description for the IBM Personal Computer, also known as **ansipc**. This is the default description used with your COHERENT system console:

```
ap|ansipc|ansi personal computer:\
        :al=\E[L:am:bs:bt=\E[Z:bw:cd=\E[O:ce=\E[K:ch=\E[%i%d‘:\
        :cl=\E[2O:cm=\E[%i%d;%dH:co#80:cs=\E[%i%d;%dr:\
        :cv=\E[%i%dd:dl=\E[M:ho=\E[H:is=\E[25f\E[2K\E[m\E[H:\
        :k0=\E[0x:k1=\E[1x:k2=\E[2x:k3=\E[3x:k4=\E[4x:k5=\E[5x:\
        :k6=\E[6x:k7=\E[7x:k8=\E[8x:k9=\E[9x:kb=^h:kd=\E[B:kh=\E[H:\
        :kl=\E[D:kr=\E[C:ku=\E[A:li#24:ll=\E[24;1H:hd=\E[C:se=\E[m:\
        :sf=\E[S:sg#0:so=\E[7m:sr=\E[T:ue=\E[m:up=\E[A:us=\E[4m:\
        :KI=\E[5x:KD=\E[3x:Kd=\E[P:KB=\E[6x:KU=\E[4x:Ku=\E[@:\
        :KM=\E[7x:KJ=\E[8x:Kt=\E[Z:KT=\t:KL=\E[1x:KR=\E[2x:KP=\E[U:\
        :Kp=\E[V:KX=\E[9x:KC=\E[0x:KE=\E[24H:KW=^F:Kw=^R:Kr=^N:do=\E[B:
```

The first field, which occupies line 1, gives the various aliases for this device. The remaining fields mean the following:

:al=\E[L: **<esc>L** adds new blank line; use one millisecond for each line added.

:am: Terminal has automatic margins.
:bs: Backspace character is **<ctrl>-H** (the default).
:bt=\E[Z: **<esc>[Z** back-tabs.
:bw: On this device, a backspace character wraps from column 0 to the last column (in this case, column 79) on the previous line.
:cd=\E[O: **<esc>[O** clears to the end of display.
:ce=\E[K: **<esc>[K** clears to end of line.
:ch=\E[%i%d‘: The string for horizontal cursor motion (described later).
:cl=\E[2O: **<esc>[2O** clears screen.
:cm=\E[%i%d;%dH:
Cursor motion (described later).
:co#80: Screen has 80 columns.
:cs=\E[%i%d;%dr:
String for changing the scrolling region.
:cv=\E[%i%dd: String for vertical cursor motion.
:dl=\E[M: **<esc>E[M** deletes a line.
:ho=\E[H: **<esc>[H** moves cursor to home position.
:is=\E[25f\E[2K\E[m\E[H:
The string with which the device is initialized.
:k0=\E[0x: Function key 10 sends sequence **<esc>[0x**.
:k1=\E[1x: Function key 1 sends sequence **<esc>[1x**.
:k2=\E[2x: Function key 2 sends sequence **<esc>[2x**.
:k3=\E[3x: Function key 3 sends sequence **<esc>[3x**.
:k4=\E[4x: Function key 4 sends sequence **<esc>[4x**.
:k5=\E[5x: Function key 5 sends sequence **<esc>[5x**.
:k6=\E[6x: Function key 6 sends sequence **<esc>[6x**.
:k7=\E[7x: Function key 7 sends sequence **<esc>[7x**.
:k8=\E[8x: Function key 8 sends sequence **<esc>[8x**.
:k9=\E[9x: Function key 9 sends sequence **<esc>[9x**.
:kb=^h: Backspace key sends **<Ctrl>-H**.
:kd=\E[B: Down-arrow key sends **<esc>[B**.
:kh=\E[H: Home key sends **<esc>[H**.
:kl=\E[D: Left-arrow key sends **<esc>[D**.
:kr=\E[C: Right-arrow key sends **<esc>[C**.
:ku=\E[A: Up-arrow key sends **<esc>[A**.
:li#24: Terminal has 24 lines.
:ll=\E[24;1H: **<esc>[24;1H** moves the cursor to the first column of the last line.
:hd=\E[C: **<esc>[C** moves the cursor downward by one-half line.
:se=\E[m: **<esc>[m** exits standout mode.
:sf=\E[S: **<esc>[S** scrolls the screen forward.
:sg#0: The **so** and **se** instructions leave zero blank lines on the screen.
:so=\E[7m: **<esc>[7m** begins standout mode.
:sr=\E[T: **<esc>[T** reverse-scrolls the screen.
:ue=\E[m: **<esc>m** ends underline mode.
:up=\E[A: **<esc>[A** moves the cursor up one line.
:us=\E[4m: **<esc>4m** begins underscore mode.
:do=\E[B: **<esc>E[B** moves cursor down one line.

Note that the last field did *not* end with a backslash; this indicated to the COHERENT system that the **termcap** description was finished.

A terminal description does not have to be nearly so detailed. If you wish to use a new terminal, first check the following table to see if it already appears by **termcap**. If it does not, check the terminal's documentation to see if it mimics a terminal that is already in **/etc/termcap**, and use that description, modifying it if necessary and changing the name to suit your terminal. If you must create an entirely new description, first prepare a skeleton file that contains the following basic elements: number of lines, number of columns, backspace, cursor motion, line delete, clear screen, move cursor to home position, newline, move cursor up a line, and non-destructive right space. For example, the following is the **termcap** description for the Lear-Siegler ADM-3A terminal:

```
la|adm3a|3a|lsi adm3a:\
        :am:bs:cd=^W:ce=^X:cm=\E=%+ %+ :cl=^Z:co#80:ho=^^:li#24:\
        :nd=<ctrl-L>:up=^K:
```

LEXICON

Once you have installed and debugged the skeleton description, add details gradually until every feature of the terminal is described.

Cursor Motion

The cursor motion characteristic contains **printf**-like escape sequences not used elsewhere. These encode the line and column positions of the cursor, whereas other characters are passed unchanged. If the **cm** string is considered as a function, then its arguments are the line and the column to which the cursor is to move; the **%** codes have the following meanings:

%d Decimal number, as in **printf**. The origin is 0.

%2 Two-digit decimal number. The same as **%2d** in **printf()**.

%3 Three-digit decimal number. The same as **%3d** in **printf()**.

%. Single byte. The same as **%c** in **printf()**.

%+*n* Add *n* to the current position value. *n* may be either a number or a character.

%>*nm* If the current position value is greater than *n*+*m*; then there is no output.

%r Reverse order of line and column, giving column first and then line. No output.

%i Increment line and column.

%% Give a **%** sign in the string.

%n Exclusive or line and column with 0140 (Datamedia 2500 terminal only).

%B Binary coded decimal (16 * (*n*/10))+(*n*%10). No output.

%D Reverse coding (*n*-**(2*(***n***%16)**). No output (Delta Data terminal only).

To send the cursor to line 3, column 12 on the Hewlett-Packard 2645, the terminal must be sent **<esc>&a12c03Y** padded for 6 milliseconds. Note that the column is given first and then the line, and that the line and column are given as two digits each. Thus, the **cm** capability for the Hewlett-Packard 2645 is given by:

```
:cm=6\E&%r%2c%2Y:
```

The Microterm ACT-IV needs the current position sent preceded by a **<Ctrl-T>**, with the line and column encoded in binary:

```
:cm=^T%.%.:
```

Terminals that use **%.** must be able to backspace the cursor (**bs** or **bc**) and to move the cursor up one line on the screen (**up**). This is because transmitting **\t**, **\n**, **\r**, or **<ctrl-D>** may have undesirable consequences or be ignored by the system.

Similar Terminals

If your system uses two similar terminals, one can be defined as resembling the other, with certain exceptions. The code **tc** names the similar terminal. This field must be *last* in the **termcap** entry, and the combined length of the two entries cannot exceed 1,024 characters. Capabilities given first over-ride those in the similar terminal, and capabilities in the similar terminal can be cancelled by **xx@** where *xx* is the capability. For example, the entry

```
hn|2621nl|HP 2621nl:ks@:ke@:tc=2621
```

defines a Hewlett-Packard 2621 terminal that does not have the **ks** and **ke** capabilities, and thus cannot turn on the function keys when in visual mode.

Initialization

A terminal initialization string may be given with the **is** capability; if the string is too long, it may be read from a file given by the **if** code. Usually, these strings set the tabs on a terminal with settable tabs. If both **is** and **if** are given, **is** will be printed first to clear the tabs, then the tabs will be set from the file specified by **if**. The Hewlett-Packard 2626 has:

```
:is=\E&j@\r\E3\r:if=/usr/lib/tabset/stdcrt:
```

Programming With termcap

The COHERENT library **libterm.a** contains the following routines that extract and use the descriptions for **termcap**:

tgetent() Read a **termcap** entry.

tgetflag() Check if a given Boolean capability is present in the terminal's **termcap** entry.

tgetnum() Return the value of a numeric **termcap** feature (e.g., the number of columns on the terminal).

tgetstr() Read and decode a **termcap** string feature.

tgoto() Read and decode a cursor-addressing string.

tputs() Read and decode the leading padding information of a **termcap** string feature.

See the Lexicon entry for each function for details.

The external variable **ospeed** is the output speed to the terminal as encoded by **stty**. The external variable **PC** is a padding character if a NUL (**<crtl-@>**) is not appropriate.

The following example shows how to read a **termcap** entry:

```
#include <stdio.h>

static char *CM, *SO, *SE, *CL;
static int rows, cols;
static int am;
static int errflag;
static char *ptr;
static char *tv_stype;

extern char *tgoto();                       /* termcap cursor position command */
extern char *tgetstr();                     /* get string code from termcap */
extern int tgetflag();                      /* get boolean flag from termcap */
extern int tgetnum();                       /* get numeric code from termcap */
extern void tputs();                        /* termcap put data command */
extern char PC;                             /* termcap's pad character */

/*
 * Get a required termcap string or exit with a message.
 */
static char *
qgetstr(ref)
char *ref;
{
      register char *tmp;

      if ((tmp = tgetstr(ref, &ptr)) == NULL) {
            printf("/etc/termcap terminal %s must have a %s= entry\n",
                  tv_stype, ref);
            errflag = 1;
      }
      return (tmp);
}

/*
 * Get required termcap information for this terminal type.
 */
static void
tcapopen()
{
      extern char *getenv(), *realloc();
      char *tcapbuf;
      char tcbuf[1024]; /* this must hold the whole tml entry */
      char *p;

      /* set up termcap type */
      if ((tv_stype = getenv("TERM")) == NULL) {
            printf("Environment variable TERM not defined\n");
            exit(1);
      }
```

```
      if (tgetent(tcbuf, tv_stype) != 1) {
            printf("Terminal type %s not in /etc/termcap\n", tv_stype);
            exit(1);
      }

      /* get far too much and shrink later */
      if ((ptr = tcapbuf = malloc(1024)) == NULL) {
            printf("out of space\n");
            exit(1);
      }

      /* get termcap entries for later use */
      CM = qgetstr("cm"); /* this string used by tgoto() */
      CL = qgetstr("cl"); /* this string used to clear screen */
      SO = qgetstr("so"); /* this string used to set standout */
      SE = qgetstr("se"); /* this string used by clear standout */
      if (errflag)      /* set if any missing entries */
            exit(1);

      /* set termcap's pad char */
      PC = (((p = tgetstr("pc", &ptr)) == NULL) ? 0 : *p);

      if (tcapbuf != realloc(tcapbuf, (unsigned)(ptr - tcapbuf))) {
            printf("Buffer not shrunk in place!\n");
            exit(1);
      }

      if ((cols = tgetnum("co")) < 0)     /* Get rows and columns */
            cols = 80;
      if ((rows = tgetnum("li")) < 0)
            rows = 24;

      am = tgetflag("am");    /* automatic margins ? */
}

/*
 * output char function.
 */
static void
ttputc(c)
{
      fputc(c, stdout);
}

/*
 * output command string, set padding to one line affected.
 * use ttputc as character output function. Use only for
 * termcap created data not your own strings.
 */
void
putpad(str)
char *str;
{
      tputs(str, 1, ttputc);
}

/*
 * Move cursor.
 */
void
move(col, row)
{
      putpad(tgoto(CM, col, row));
}
```

```
/*
 * Demonstrate termcap.
 */
main()
{
      tcapopen();

      putpad(CL);             /* clear the screen */

      move(30, 5);
      putpad(SO);             /* standout mode */
      printf("Termcap Demo");
      putpad(SE);             /* end standout mode */

      move(0, 7);
      printf("This terminal has %d columns and %d rows.", cols, rows);

      if (am) {
            move(0, 8);
            printf("Automatic margins.");
      }

      move(0, rows);    /* quit at bottom of screen */
      exit(0);
}
```

Files

/etc/termcap — Terminal-description data base
/usr/lib/libterm.a — Routines for reading a **termcap** description

See Also

Administering COHERENT, captoinfo, curses, libterm, terminal, terminfo, tgetent(), tgetflag(), tgetnum(), tgetstr(), tgoto(), tputs()
Strang, J., Mui, L., O'Reilly, T.: *Termcap & Terminfo.* Sebastopol, CA: O'Reilly & Associates, Inc., 1991. *Highly recommended.*

Notes

To see which terminals are currently supported, see file **/etc/termcap**.

COHERENT also supports **terminfo**, a clone of the UNIX System-V terminal-description system. **terminfo** enjoys a number of features not available under **termcap**, and is the preferred system under COHERENT.

Should you wish to convert to **terminfo**, the command **captoinfo** converts a file of **termcap** descriptions to their **terminfo** analogues.

terminal — Technical Information

This article describes how you can hook up a terminal to your COHERENT system via a serial port. It also discusses common problems that arise with this procedure, as diagnosed daily by the technical support staff at Mark Williams Company. For information on connecting a modem to your computer's serial port, see the article **modem**.

Terminals for Beginners

To a beginner, a terminal — with its many wires and controls — may be daunting. However, connecting a terminal or PC-based terminal emulator to a COHERENT system is really very easy. To make matters even easier, this section discusses how to use a simple configuration, using only one COM port.

What you need:

1. A COHERENT system with a COM port.
2. Either a terminal or a PC with a COM port and communications software. An old PC or PC-AT is fine. (You can also use a Macintosh or other such computer, but that is beyond the scope of this discussion.)
3. A "null modem cable" purchased from your nearest computer supply store. This should cost between $10 and $20, in U.S. currency. The only tricky part about the cable is making sure that the connectors on the ends match the connectors on your COM ports (i.e., nine pin vs. 25 pin, and male vs female. Adaptors are also available.

What you do:

1. Connect the cable to the COM port on the COHERENT system. Note that the COM port on your COHERENT system is always a *male* plug (that is, it has pins rather than sockets). Do *not* plug your connection into a female plug, as this is the parallel port. If you do so, you can damage your equipment.

2. Connect the cable to the COM port on the terminal/PC. If you are using a PC as a terminal, the COM port on it will also be male. If you are using a stand-alone terminal, the plug could be either male or female.

   If you are using a stand-alone terminal, there may be a plug on the back that is labelled "AUX". Do *not* use this plug; use the other one.

3. Configure the terminal/PC to use 9600 speed (or "baud") and 8N1 character setting (that is, eight bits, no parity, one stop bit. Note that you do not need a telephone number: you won't be dialing anywhere.

4. Log in on your COHERENT system as the superuser **root**. Type the following command:

   ```
   /etc/enable com?l
   ```

   where '?' is the number of the COM port on your COHERENT system into which you are plugging the cable from the PC or terminal Note that the last character is lower-case 'L', *not* a one.

5. Now come a tricky part: use your favorite text editor and edit file **/etc/ttytype**. This file gives the default type of terminal that will be connecting to COHERENT via a given COM port. This is important. If you don't do this, such screen-oriented programs as editors and **vsh** will not work properly.

   Each entry in **/etc/ttytype** has two columns: the first gives a type of terminal, and the second names the port. The following shows an example entry:

   ```
   vt100 com3l
   ```

   In this instance, COM port **com3l** has a DEC VT-100 plugged into it by default. Look for an entry for the COM port into which you plugged your terminal device. If you can't find you, you can create one easily enough; however, having two entries for the same port is not a good idea, as COHERENT will become confused.

   If you are plugging in a PC, use the terminal type **vt100**. If you are using a stand-alone terminal, name the type of terminal you are using, e.g., **wyse370** for a Wyse 370 color terminal. If you cannot figure out what type of terminal you are using, use **dumb**. This will not allow you to use screen-oriented programs like MicroEMACS, but at least you will be able to connect to COHERENT; you can figure out the type of terminal later.

6. Return to your terminal device. If you are using a PC, invoke the terminal emulator, and put it into "connect" mode. If you are using a stand-alone terminal, turn it on. Press `(¢)`. After a short pause, you should see the prompt

   ```
   Coherent login:
   ```

   on your screen. You can then log in and run normally.

That's all there is to it. In this way, you can get more use out of an old, obsolete PC. After you get it working, you may need to adjust some other settings, particularly if you are using a communications package on a PC for a terminal.

If you see garbage characters when you log in, probably the speed (or "baud rate") is not set correctly either on the COM port of your COHERENT system or on your terminal device. The Lexicon article on **ttys** describes the magic character string used to describe each com port and one of the letters is the port speed.

Problems with Terminal Hookup

As noted above, it is easy to hook up a terminal. However, problems can arise if you overlook any details.

When problems arise, they usually come from some form of confusion. These can include send/receive confusion, baud rate confusion, and shell/no shell confusion. The following sections describes eacy type of confusion in turn.

Send/Receive Confusion

This most often happens when you've soldered your own connectors, and you made a mistake. If you purchased a connector, as we recommended above, then skip to the next section.

A serial connection between your computer and a terminal requires at least three wires: one each for pins 2, 3, and 7. These pins, respectively, control send (TD), receive (RD), and signal-ground (Gnd or SG). These pin numbers

correspond to the 25-pin "DB-25" connectors used on most equipment. If your system has the AT-style nine-pin "DB-9" connectors, you will need to wire to the corresponding signals. See the Lexicon entry for **RS-232** for details of the pin-outs for these two connectors.

When hooking up a terminal to a serial port using a three-wire connection, you must cross pins 2 and 3, so that each device's send pin talks to the other device's receive pin. You can plug a device called a "null modem" between the cable and the serial port, to do this automatically.

Note that the only symptom of a problem in the cable is that nothing appears on your terminal when you type.

Baud-Rate Confusion

The terminal and the computer must speak to each other at the same speed, or "baud rate". A typical symptom of baud-rate confusion is garbage characters on the screen. When the wiring is wrong, you see nothing; when the baud rate is wrong, you see random collections of characters on the screen, but nothing meaningful.

You can fix baud-rate problems by using the command **stty** to reset the baud rate on the port, or resetting the baud rate on the terminal. The problem should also be solved by editing file **/etc/ttys**. For directions on how to reset the baud rate for a port, see the Lexicon entry for **stty**; see the Lexicon entry for **ttys** for information on how to edit **/etc/ttys**.

Please note, too, that COHERENT supports the following configuration for terminals:

8 word bits
1 stop bit
No parity bits

These settings, as well as the baud rate, must match before your terminal will work correctly.

The Old Shell Game

Before a terminal is useful to you, you must *enable* the port into which it is plugged. Enabling a port means that the COHERENT system creates a shell for that port: this, in turn, means that COHERENT prints a login prompt on the device plugged into that port, and reads and processes interactively commands that are entered from that port. The COHERENT system also restricts permissions on all enabled serial ports, so that only the superuser **root** can read and write to the port. This prevents other users who may be using the system from accessing the serial port.

Note that not all ports need be enabled: for example, you should not enable a COM port into which you've plugged a printer.

When you boot the COHERENT system, it reads system file **/etc/ttys** and creates a shell for each serial port that needs one. One way to enable a port is to log in as the superuser **root**, then use a text editor to change the port's entry in **/etc/ttys**, as described its Lexicon article. Finally, typing the command

```
kill quit 1
```

forces COHERENT to re-read **/etc/ttys** and so create a shell for the port. Note that doing this will ensure that the port is re-enabled every time you boot.

A better way to enable a port is to use the command **enable**, as described in its Lexicon article. For example, to put up a shell on COM port **/dev/com1r**, log in as the superuser **root** and type the command:

```
/etc/enable com1r
```

Exiting Raw Mode

A terminal is in *cooked* mode. In cooked mode, the tty driver interprets and correctly processes such predefined characters as the end-of-file character or the quit character. In *raw* mode, however, processing of such characters is turned off; and in general the terminal will behave bizarrely. Raw mode is used by programs that do not want the tty driver to interpret characters; for example, a program that uses a tty to transmit a binary to another machine does not want the tty driver to be interpreting the binary information being passed through it.

Occasionally, a program will exit abruptly and leave the terminal in raw mode. To return to cooked mode, use the command **<ctrl-J> stty sane <ctrl-J>**. This invokes the command **stty**, which lets you manipulate terminal settings, to restore the previous cooked state. See the Lexicon entry on **stty** for details on raw and cooked modes; this article also describes the options of this most useful command.

See Also

Administering COHERENT, asy, device drivers, hs, modem, RS-232, sgtty, stty, termcap, terminfo, termio, termios, ttys

LEXICON

Notes

One final bit of hard-won wisdom: once you have something working, write down what you did, and store it in a place where you won't lose it. Note especially what connectors are where and how they have been cabled together. It makes life easier just knowing that you are looking for a female-to-female cable instead of male-to-female or male-to-male. If you know whether to insert a null modem, you are even better off.

COHERENT supports multi-port serial cards as well as COM ports 1 through 4. See the Lexicon entry on **asy** for a list of the devices that COHERENT supports, and for details.

Thanks to Dave Hough (tecdah1@sdc.cs.boeing.com), whose posting to comp.os.coherent is the basis of the "Beginners" section, above.

terminfo — System Administration

Terminal-description language
/usr/lib/terminfo

terminfo is a system for describing terminals. Descriptions are collected in the file **/usr/lib/terminfo** and are read by **curses**, **more**, **vi**, and other utilities. By passing her terminal's **terminfo** entry to a program, a user can make sure that the program can take full advantage of her terminal's capacities.

terminfo resembles the terminal-description language **termcap**; however, it enjoys a number of features that **termcap** does not, as follows:

- A **termcap** entry cannot exceed a predefined limit. **terminfo** lifts this restriction.
- **terminfo** entries are compiled; therefore, they are read and loaded more quickly.
- **termcap** entries are all kept in file **/etc/termcap**. Each **terminfo** resides in its own file; thus, a program and find and load an entry more quickly.
- **terminfo** is a little more easily read by human beings.

Whether a program uses **termcap** or **terminfo** descriptions depends entirely on that program. For example, MicroEMACS uses **termcap** descriptions; but **vsh** (and other **curses**-based programs) use **terminfo**. In general, **terminfo** is regarded as being more flexible and up-to-date.

terminfo Entries

Directory **/usr/lib/terminfo** consists of a number of sub-directories, one for each terminal type being described. A *terminal type* describes a given make of terminal (e.g., the Wyse 150) plus some special attribute, such as the number of characters on a line or a specially defined bank of function keys. A **terminfo** entry can extend over more than one line by indenting every line after the first. A line that begins with a pound sign '#' is a comment.

A **terminfo** entry consists of an indefinite number of comma-separated fields. White space after each comma is ignored. The first field names the terminal; the remaining fields hold capability codes. (*Capability codes* are discussed in detail below.) Preceding a field with a period '.' comments out that field, and only that field.

Naming Terminals

The first field in a **terminfo** entry names the terminal being described. The name field consists of one or more names, which are separated by vertical-bar characters. The first name given is the most common abbreviation for the terminal. The last name is usually a long name that fully identifies the terminal. All names in between the first and the last give common synonyms for that terminal. All names can contain upper-case characters; the last name can also contain white space.

Terminal names (except for the last, verbose entry) should use the following conventions:

- The hardware should have a root name chosen, e.g., "wyse150".
- The root name should not contain hyphens, except to prevent synonyms from colliding with other names.
- Modes that the hardware can be in, or user preferences, should be indicated by appending a hyphen and an indicator of the mode. For example, a **wyse150** with an old-fashioned 82-key keyboard could be called **wyse150-o**.

Use the following suffixes whenever possible:

Suffix	*Meaning*
-w	Wide (more than 80 columns)
-am	With automatic margins (usually default)
-nam	Without automatic margins
-*n*	Number of lines on the screen
-na	No arrow keys
-*n***p**	*n* pages of memory
-rv	Reverse video

Capability Codes

A *capability code* describes a capability of a terminal. Capability codes come in three varieties:

Boolean This indicates whether a terminal has a given feature. If the field is present, the terminal is assumed to have the capability; if not, then it is assumed not to be present. For example, the code **am** indicates "automatic margins". If **am** appears in a terminal's **terminfo** entry, then it can execute automatic margins; if not, then it can't.

Numeric This gives the size of some aspect of a terminal, such as the number of lines or the number of columns. A numeric code is followed by a number sign '#' and then a string of digits, which set the value for that code For example, the code **cols#80** indicates that a terminal has 80 columns per row.

String Capabilities
This gives a sequence of characters that trigger a terminal operation. For example, a terminal may expect a "magic sequence" to wipe the screen clean, to print in reverse video, or to change the shape of its cursor. Likewise, a terminal may send a "magic sequence" when a particular function key is pressed. For example, the code **klf1=\E5** indicates that this terminal sends the string **<esc>5** when the user presses function-key 1.

Some terminal capabilities may involve *padding* — that is, telling the terminal to delay execution of the capability for a fraction of a second. In some instances, padding may make the difference between a terminal's drawing information correctly, or displaying a jumble.

A delay code can appear anywhere in a string capability code. It is introduced by a dollar sign '$' and enclosed in angle brackets '<>'. The numeric value is always in milliseconds. For example, the code **el=\EK$<3>** indicates that the clear-to-end-of-line code **el** is invoked by the "magic sequence" **<esc>K**, and that it should involve a three-millisecond delay. Function **tputs()** provides the delay.

The delay can be either a number, e.g., "20", or a number followed by an asterisk, e.g., "3*". An asterisk indicates that the padding must be proportional to the number of lines affected by the operation; the amount given is the amount of padding required by each line of output. (This is true even in the case of the insert-character code.) When an asterisk is specified, it is sometimes useful to give a delay of the form "3.5" to specify a delay-per-unit to tenths of milliseconds. (Only one decimal place is allowed.)

The following table gives the commonest **terminfo** capability codes. The *variable* is the name by which the programmer (at the **terminfo** level) accesses the capability. The *code* is the name used in the **terminfo** entry. There is no fixed limit to the length of a code, but the convention is to keep them to five characters or fewer. Whenever possible, names are the same as, or similar to, those in the ANSI Standard X3.64-1979.

The semantics describe features of the code:

† You may specify padding.

†* Padding may be based on the number of lines affected.

The string is passed through **tparm()** with the number of parameters given in the description.

#*i* Indicate the *i*th parameter.

Boolean Codes

Code	*Variable*	*Description*
am	auto_right_margin	Automatic margins
bce	back_color_erase	Erase screen with background color
bw	auto_left_margin	**cub1** wraps from column 0 to last column
ccc	can_change	Terminal can redefine a color

LEXICON

chts	hard_cursor	Cursor is difficult to see
cpix	cpi_changes_res	Changing character pitch also changes resolution
crxm	cr_cancels_micro_mode	Carriage return cancels micro mode
da	memory_above	Display can be retained above the screen
daisy	has_print_wheel	You must change print wheel on printer
db	memory_below	Display can be retained below the screen
eo	erase_overstrike	Erase overstrikes with a blank
eslok	status_line_esc_ok	Escape can be used on the status line
gn	generic_type	Generic line type (e.g., dialup, switch).
hc	hard_copy	Hardcopy terminal
hls	hue_lightness_saturation	Terminal uses HLS color notation
hs	has_status_line	Has an extra "status line"
hz	tilde_glitch	Hazeltine cannot print tildes '~'
in	insert_null_glitch	Insert mode distinguishes NULs
km	has_meta_key	Has a metakey (shift sets parity bit)
mc5i	prtr_silent	Printer does not echo on screen
mir	move_insert_mode	Safe to move while in insert mode
msgr	move_standout_mode	Safe to move in standout modes
npc	no_pad_char	No padding character
nxon	needs_xon_xoff	Padding does not work: needs XON/XOFF
os	over_strike	Terminal overstrikes
sam	semi_auto_right_margin	Printing in last column returns carriage
ul	transparent_underline	Underline character overstrikes
xenl	eat_newline_glitch	Newline ignored after 80 columns (Concept)
xhp	ceol_standout_glitch	Standout not erased by overwriting (HP)
xhpa	col_addr_glitch	Only positive motion for HPA/MHPA capitals
xon	xon_xoff	Terminal uses XON/XOFF handshaking
xsb	no_esc_ctlc	Beehive terminal (F1=escape, F2=**<ctrl-C>**)
xvpa	row_addr_glitch	Only positive motion for VPA/MVPA capitals
xt	teleray_glitch	Tabs destructive, magic SO char (Teleray 1061)

Numeric Codes

Code	*Variable*	*Description*
bufsz	buffer_capacity	Number of bytes buffered before printing
colors	max_colors	Maximum number of colors on the screen
cols	columns	Number of columns in a line
it	init_tabs	Tabs initially every *n* spaces
lines	lines	Number of lines on screen or page
lm	lines_of_memory	Lines of memory if greater than **lines**; zero, variable
maddr	max_micro_address	Maximum value in micro_ ... _address
mjump	max_micro_jump	Maximum value in parm_ ... _micro
mls	micro_line_size	Line-step size when in micro mode
ncv	no_color_video	Video attributes that cannot be used with color
nlab	num_labels	Number of labels on the screen
npins	number_of_pins	Number of pins in the print-head
orc	output_res_char	Horizontal resolution, units per character
orhi	output_res_horz	Horizontal resolution in units per inch
orl	output_res_line	Vertical resolution, units per line
orvi	output_res_vert	Vertical resolution, units per inch
pairs	max_pairs	Maximum number of color_pairs on screen
pb	padding_baud_rate	Lowest baud rate where CR/NL padding is needed
spinh	dot_horz_spacing	Spacing of pins horizontally (pins/inch)
spinv	dot_vert_spacing	Spacing of pins vertically (pins/inch)
vt	virtual_terminal	Virtual terminal number
widcs	wide_char_size	Character step size, double-width mode
wsl	width_status_line	Number of columns in the status line
xmc	magic_cookie_glitch	Number of blank characters left by **smso** or **rmso**

String Capabilities

Code	*Variable*	*Description*

CC	command_character	Terminal-settable command character in prototype
acsc	acs_chars	Pairs of graphical character set (aAbBcC ...)
bel	bell	Audible signal (bell)†
blink	enter_blink_mode	Turn on blinking
bold	enter_bold_mode	Turn on bold (extra bright)
cbt	back_tab	Back tab†
ch	erase_charse	Erase #1 characters†#
chr	change_res_horz	Change horizontal resolution
civis	cursor_invisible	Make cursor invisible
clear	clear_screen	Clear screen†*
cnorm	cursor_normal	Make cursor appear normal (undo **vs** and **vi**)
cpi	change_char_pitch	Change number of characters per inch
cr	carriage_return	Carriage return†*
csnm	char_set_names	Names of character sets
csr	change_scroll_region	change to lines #1 through #2 (vt100)†#
cub	parm_left_cursor	Move cursor left #1 spaces†#
cub1	cursor_left	Move cursor left one space
cud	parm_down_cursor	Move cursor down #1 lines.†*#
cud1	cursor_down	Move cursor down one line
cuf	parm_right_cursor	Move cursor right #1 spaces†*#
cuf1	cursor_right	Move cursor right one space
cup	cursor_address	Cursor motion relative to row 1 column 2†#
cuu	parm_up_cursor	Move cursor up #1 lines†*#
cuu1	cursor_up	Upline (cursor up)
cvr	change_rs_vert	Change vertical resolution
cvvis	cursor_visible	Make cursor very visible
dch	parm_dch	Delete #1 chars†*#
dch1	delete_character	Delete character†*
defc	define_char	Define a character in a character set
dim	enter_dim_mode	Turn on half-bright mode
dl	parm_delete_line	Delete #1 lines†*#
dl1	delete_line	Delete line†*
docr	these_cause_cr	List of characters that trigger carriage return
dsl	dis_status_line	Disable status line
ech	erase_chars	Erase no. 1 characters
ed	clr_eos	Clear to end of display†*
el	clr_eol	Clear to end of line†
el1	clr_bol	Clear to beginning of line, inclusive
enacs	ena_acs	Enable alternate character set
flash	flash_screen	Visible bell (may not move cursor)
ff	form_feed	Hardcopy terminal page eject†*
fsl	from_status_line	Return from status line
hd	down_half_line	Half-line down (forward 1/2 linefeed)
home	cursor_home	Move cursor to home position (if no **cup**)
hpa	column_address	Set cursor column†#
ht	tab	Tab to next eight-space hardware tab stop
hts	set_tab	Set a tab in all rows, current column.
hu	up_half_line	Half-line up (reverse 1/2 linefeed)
ich	parm_ich	Insert #1 blank characters†*#
ich1	insert_character	Insert character†
if	init_file	Name of file containing **is**
il	parm_insert_line	Add #1 new blank lines†*#
il1	insert_line	Add new blank line†*
ind	scroll_forward	Scroll text up†
indn	parm_index	Scroll forward #1 lines†#
initc	initialize_color	Initialize color definition
initp	initialize_pair	Initialize color pairs
invis	enter_secure_mode	Turn on blank mode (characters invisible)
ip	insert_padding	Insert pad after character inserted†*
iprog	init_prog	Full path name of initialization program
is1	init_1string	Terminal-initialization string
is2	init_2string	Terminal-initialization string

is3	init_3string	Terminal-initialization string
kBEG	key_sbeg	Sent by shifted beginning key
kCAN	key_scancel	Sent by shifted cancel key
kCMD	key_scommand	Sent by shifted command key
kCPY	key_scopy	Sent by shifted copy key
kCRT	key_screate	Sent by shifted create key
kDC	key_sdc	Sent by shifted delete-character key
kDL	key_sdl	Sent by shifted delete-line key
kEND	key_send	Sent by shifted end key
kEOL	key_seol	Sent by shifted EOL clear-line key
kEXT	key_sexit	Sent by shifted exit key
kFND	key_sfind	Sent by shifted find key
kHLP	key_shelp	Sent by shifted help key
kHOM	key_shome	Sent by shifted home key
kIC	key_sic	Sent by shifted input key
kLFT	key_sleft	Sent by shifted (æ) key
kMOV	key_smove	Sent by shifted move key
kMSG	key_smessage	Sent by shifted message key
kNXT	key_snext	Sent by shifted next key
kOPT	key_soptions	Sent by shifted option key
kPRT	key_sprint	Sent by shifted print key
kPRV	key_sprevious	Sent by shifted previous key
kRDO	key_sredo	Sent by shifted redo key
kRES	key_srsume	Sent by shifted resume key
kRIT	key_sright	Sent by shifted (Æ) key
kRPL	key_sreplace	Sent by shifted replace key
kSAV	key_ssave	Sent by shifted save key
kSPD	key_ssuspend	Sent by shifted suspend key
kUND	key_sundo	Sent by shifted undo key
ka1	key_a1	Sent by key A1, upper left of keypad
ka3	key_a3	Sent by key A3, upper right of keypad
kb2	key_b2	Sent by key B2, center of keypad
kbeg	key_beg	Sent by "begin" key
kbs	key_backspace	Sent by backspace key
kc1	key_c1	Sent by key C1, lower left of keypad
kc3	key_c3	Sent by key C3, lower right of keypad
kcan	key_cancel	Sent by cancel key
kcbt	key_btab	Sent by back-tab key
kclo	key_close	Sent by close key
kclr	key_clear	Sent by clear-screen or erase key
kcmd	key_command	Sent by "cmd" key
kcpy	key_copy	Sent by copy key
kcrt	key_create	Sent by create key
kctab	key_ctab	Sent by clear-tab key
kcub1	key_left	Sent by (æ) key
kcud1	key_down	Sent by (°) key
kcuf1	key_right	Sent by (Æ) key
kcuu1	key_up	Sent by terminal (ª) key
kdch1	key_dc	Sent by delete-character key
kdl1	key_dl	Sent by delete-line key
ked	key_eos	Sent by clear-to-end-of-screen key
kel	key_eol	Sent by clear-to-end-of-line key
kend	key_end	Sent by end key
kent	key_enter	Sent by (¢) key
kext	key_exit	Sent by exit key
kf0	key_f0	Sent by function key 0
kf1	key_f1	Sent by function key 1
kf10	key_f10	Sent by function key 10
kf11	key_f11	Sent by function key 11
kf12	key_f12	Sent by function key 12
kf13	key_f13	Sent by function key 13
kf14	key_f14	Sent by function key 14

kf15. . . . key_f15 . . . . . Sent by function key 15
kf16. . . . key_f16 . . . . . Sent by function key 16
kf17. . . . key_f17 . . . . . Sent by function key 17
kf18. . . . key_f18 . . . . . Sent by function key 18
kf19. . . . key_f19 . . . . . Sent by function key 19
kf2 . . . . key_f2 . . . . . Sent by function key 2
kf20. . . . key_f20 . . . . . Sent by function key 20
kf21. . . . key_f21 . . . . . Sent by function key 21
kf22. . . . key_f22 . . . . . Sent by function key 22
kf23. . . . key_f23 . . . . . Sent by function key 23
kf24. . . . key_f24 . . . . . Sent by function key 24
kf25. . . . key_f25 . . . . . Sent by function key 25
kf26. . . . key_f26 . . . . . Sent by function key 26
kf27. . . . key_f27 . . . . . Sent by function key 27
kf28. . . . key_f28 . . . . . Sent by function key 28
kf29. . . . key_f29 . . . . . Sent by function key 29
kf3 . . . . key_f3 . . . . . Sent by function key 3
kf30. . . . key_f30 . . . . . Sent by function key 30
kf31. . . . key_f31 . . . . . Sent by function key 31
kf32. . . . key_f32 . . . . . Sent by function key 32
kf33. . . . key_f33 . . . . . Sent by function key 33
kf34. . . . key_f34 . . . . . Sent by function key 34
kf35. . . . key_f35 . . . . . Sent by function key 35
kf36. . . . key_f36 . . . . . Sent by function key 36
kf37. . . . key_f37 . . . . . Sent by function key 37
kf38. . . . key_f38 . . . . . Sent by function key 38
kf39. . . . key_f39 . . . . . Sent by function key 39
kf4 . . . . key_f4 . . . . . Sent by function key 4
kf40. . . . key_f40 . . . . . Sent by function key 40
kf41. . . . key_f41 . . . . . Sent by function key 41
kf42. . . . key_f42 . . . . . Sent by function key 42
kf43. . . . key_f43 . . . . . Sent by function key 43
kf44. . . . key_f44 . . . . . Sent by function key 44
kf45. . . . key_f45 . . . . . Sent by function key 45
kf46. . . . key_f46 . . . . . Sent by function key 46
kf47. . . . key_f47 . . . . . Sent by function key 47
kf48. . . . key_f48 . . . . . Sent by function key 48
kf49. . . . key_f49 . . . . . Sent by function key 49
kf5 . . . . key_f5 . . . . . Sent by function key 5
kf50. . . . key_f50 . . . . . Sent by function key 50
kf51. . . . key_f51 . . . . . Sent by function key 51
kf52. . . . key_f52 . . . . . Sent by function key 52
kf53. . . . key_f53 . . . . . Sent by function key 53
kf54. . . . key_f54 . . . . . Sent by function key 54
kf55. . . . key_f55 . . . . . Sent by function key 55
kf56. . . . key_f56 . . . . . Sent by function key 56
kf57. . . . key_f57 . . . . . Sent by function key 57
kf58. . . . key_f58 . . . . . Sent by function key 58
kf59. . . . key_f59 . . . . . Sent by function key 59
kf6 . . . . key_f6 . . . . . Sent by function key 6
kf60. . . . key_f60 . . . . . Sent by function key 60
kf61. . . . key_f61 . . . . . Sent by function key 61
kf62. . . . key_f62 . . . . . Sent by function key 62
kf63. . . . key_f63 . . . . . Sent by function key 63
kf7 . . . . key_f7 . . . . . Sent by function key 7
kf8 . . . . key_f8 . . . . . Sent by function key 8
kf9 . . . . key_f9 . . . . . Sent by function key 9
kfnd. . . . key_find . . . . . Sent by find key
khlp. . . . key_help. . . . . Sent by help key
khome . . key_home . . . . . Sent by home key
khts. . . . key_stab. . . . . Sent by set-tab key
kich1 . . . key_ic . . . . . Sent by insert char/enter insert-mode key

kil1	key_il	Sent by insert line
kind	key_sf	Sent by scroll-forward/down key
kll	key_ll	Sent by home-down key
kmrk	key_mark	Sent by mark key
kmsg	key_message	Sent by message key
kmov	key_move	Sent by move key
knp	key_npage	Sent by next-page key
knxt	key_next	Sent by next-object key
kopn	key_open	Sent by open key
kopt	key_options	Sent by options key
kpp	key_ppage	Sent by previous-page key
kprt	key_print	Sent by print (copy) key
kprv	key_previous	Sent by previous-object key
krdo	key_redo	Sent by redo key
kref	key_reference	Sent by reference key
kres	key_resume	Sent by resume key
krfr	key_refresh	Sent by refresh key
kri	key_sr	Sent by scroll-backward/up key
krmir	key_eic	Sent by **rmir** or **smir** in insert mode
krpl	key_replace	Sent by replace key
krst	key_restart	Sent by restart key
ksav	key_save	Sent by save key
kslt	key_select	Sent by select key
kspd	key_suspend	Sent by suspend key
ktbc	key_catab	Sent by clear-all-tabs key
kund	key_undo	Sent by undo key
lf0	label_f0	Label on function key 0 if not F0
lf1	label_f1	Label on function key 1 if not F1
lf10	label_f10	Label on function key 10 if not F10
lf2	label_f2	Label on function key 2 if not F2
lf3	label_f3	Label on function key 3 if not F3
lf4	label_f4	Label on function key 4 if not F4
lf5	label_f5	Label on function key 5 if not F5
lf6	label_f6	Label on function key 6 if not F6
lf7	label_f7	Label on function key 7 if not F7
lf8	label_f8	Label on function key 8 if not F8
lf9	label_f9	Label on function key 9 if not F9
ll	cursor_to_ll	Last line, first column (if no **cup**)
lpi	change_line_pitch	Change number of lines per inch
mc0	print_screen	Print contents of the screen
mc4	prtr_off	Turn off printer
mc5	prtr_on	Turn on printer
mcub	parm_left_micro	Like **cub** for micro adjustment
mcub1	micro_left	Like **cub1** for micro adjustment
mcud	parm_down_micro	Like **cud** for micro adjustment
mcud1	micro_down	Like **cud1** for micro adjustment
mcuf	parm_right_micro	Like **cuf** for micro adjustment
mcuf1	micro_right	Like **cuf1** for micro adjustment
mcuu	parm_up_micro	Like **cuu** for micro adjustment
mcuu1	micro_up	Like **cuu1** for micro adjustment
mgc	clear_margins	Clear all margins (top, bottom, sides)
mhpa	micro_column_address	Like **hpa** for micro adjustment
mrcup	cursor_mem_address	Memory-relative cursor addressing
mvpa	micro_row_address	Like **vpa** for micro adjustment
nel	newline	Newline (behaves like CR followed by LF)
oc	orig_colors	Set all colors to originals
op	orig_pair	Set default color_pair to original
pad	pad_char	Pad character (rather than NUL)
pfkey	pkey_key	Program function key 1 to type string 2
pfloc	pkey_local	Program function key 1 to execute string 2
pfx	pkey_xmit	Program function key 1 to transmit string 2
pln	plab_norm	Program label 1 to show string 2

porder	order_of_pins	Match software bits to print-head pins
prot	enter_protected_mode	Turn on protected mode
rc	restore_cursor	Restore cursor to position of last **sc**
rep	repeat_char	Repeat character #1 #2 times. †*#
rev	enter_reverse_mode	Turn on reverse-video
rf	reset_file	Name of file containing reset string
rfi	reg_for_input	Send next input character
ri	scroll_reverse	Scroll text down†
rin	parm_rindex	Scroll backward one line†#
ritm	exit_italics_mode	Disable italics
rlm	exit_leftward_mode	Enable rightward motion
rmacs	exit_alt_charset_mode	End alternate character set†
rmam	exit_am_mode	Turn off automatic margins
rmcup	exit_ca_mode	String to end programs that use **cup**
rmdc	exit_delete_mode	End delete mode
rmicm	exit_micro_mode	Disable micro-motion capabilities
rmir	exit_insert_mode	End insert mode
rmkx	keypad_local	Exit "keypad transmit" mode
rmln	label_off	Turn off soft labels
rmm	meta_off	Turn off "meta mode"
rmp	char_padding	Like **ip**, but in replace mode
rmso	exit_standout_mode	End stand out mode
rmul	exit_underline_mode	End underscore mode
rmxon	exit_xon_mode	Turn off XON/XOFF handshaking
rs1	reset_1string	Reset terminal completely to sane modes
rs2	reset_2string	Reset terminal completely to sane modes
rs3	reset_3string	Reset terminal completely to sane modes
rshm	exit_shadow_mode	Disable shadow printing
rsubm	exit_subscript_mode	Disable subscript printing
rsupm	exit_superscript_mode	Disable superscript printing
rum	exit_upward_motion	Enable downward motion
rwidm	exit_doublewide_mode	Disable double-width printing
sbim	start_bit_margin	Start printing bit-mapped graphics
sc	save_cursor	Save cursor position†
scp	set_color_pair	Set current color pair
scs	select_char_set	Select character set
scsd	start_char_set_def	Start definition of a character set
sdrfq	enter_draft_quality	Set draft-quality printing
setb	set_background	Set current background color
setf	set_foreground	Set current foreground color
sgr	set_attributes	Define the nine video attributes†*#
sgr0	exit_attribute_mode	Turn off all attributes
sitm	enter_italics_mode	Enable italics
slm	enter_leftward_mode	Enable leftward carriage motion
smacs	enter_alt_charset_mode	Start alternate character set†
smam	enter_am_mode	Turn on automatic margins
smcup	enter_ca_mode	String to begin programs that use **cup**
smdc	enter_delete_mode	Delete mode (enter)
smgb	set_bottom_margin	Set bottom margin to current line
smgbp	set_bottom_margin_parm	Set bottom margin at lines 1 or 2
smgl	set_left_margin	Set left margin to current column
smglp	set_left_margin_parm	Set left margin to columns 1 or 2
smgr	set_right_margin	Set right margin to current column
smgrp	set_right_margin_parm	Set right margin to columns 1 or 2
smgt	set_top_margin	Set top margin to current line
smgtp	set_top_margin_parm	Set top margin to lines 1 or 2
smicm	enter_micro_mode	Enable micro-motion capabilities
smir	enter_insert_mode	Insert mode (enter)
smkx	keypad_xmit	Enter "keypad transmit" mode
smln	label_on	Turn on soft labels
smm	meta_on	Turn on "meta mode" (eighth bit)
smso	enter_standout_mode	Begin stand-out mode

smul	enter_underline_mode	Start underscore mode
smxon	enter_xon_mode	Turn on XON/XOFF handshaking
snlq	enter_near_letter_quality	Set near-letter-quality printing
snrmq	enter_normal_quality	Set normal-quality print
sshm	enter_shadow_mode	Enable shadow printing
ssubm	enter_subscript_mode	Enable subscript printing
ssupm	enter_superscript_mode	Enable superscript printing
subcs	subscript_characters	List of "subscript-able" characters
sum	enter_upward_mode	Enable upward carriage motion
supcs	superscript_characters	List of "superscript-able" characters
swidm	enter_doublewide_mode	Enable double-wide printing
tbc	clear_all_tabs	Clear all tab stops†
tsl	to_status_line	Go to status line, column 1
uc	underline_char	Underscore one char and move past it
vpa	row_address	Vertical position absolute (set row)†#
wind	set_window	Current window is lines #1-#2, columns 3—4
xoffc	xoff_character	XOFF character
xonc	xon_character	XON character
zerom	zero_motion	No motion for subsequent character

Escape Sequences

You can use the following escape sequences with any string-capability entry:

\E	**<esc>** character
\e	**<esc>** character
^*X*	**<ctrl-***X***>** for any appropriate *X*
\n	Newline
\r	Carriage return
\t	Horizontal tab
\b	Backspace
\f	Formfeed
\s	Space
*****000*	Value of a character in three octal digits
\^	Literal carat
\,	Literal comma
****	Literal backslash

Parameterized Strings

Cursor-addressing and other strings requiring parameters in the terminal are described by a parameterized string capability, with **printf()**-like escape sequences in it. Each escape sequence is introduced with a percent sign '%', followed by one character that described the type of formatting to be performed, as follows:

%%	Literal '%'
%d	Decimal integer
%2d	Decimal integer with at least two places
%02d	Decimal integer, two places, zero padding
%3d	Decimal integer with at least three places
%03d	Decimal integer, three places, zero padding
%c	Character
%s	String
%p[*i***]**	Push *i*th parameter
%P[*a-z***]**	Set variable [*a-z*] to **pop()**
%g[*a-z***]**	Push variable [*a-z*]
%'*c***'**	Character constant *c*
%{*nn***}**	Integer constant *nn*

%+	Addition: **push(pop() + pop())**
%-	Subtraction: **push(pop() - pop())**
%*	Multiplication: **push(pop() * pop())**
%/	Division: **push(pop() / pop())**
%m	Modulo: **push(pop() % pop())**
%&	Bitwise AND: **push(pop() op pop())**
%|	Bitwise OR: **push(pop() op pop())**
%^	Bitwise NOR: **push(pop() op pop())**
%=	Logical AND: **push(pop() op pop())**
%>	Logical OR: **push(pop() op pop())**
%<	Logical NOR: **push(pop() op pop())**
%!	Unary NOT: **push(op pop())**
%~	Unary complement: **push(op pop())**
%i	Add one to first two parmameters (for ANSI terminals)

The parameterized mechanism is based on a stack. **%** operations push parameters and constants onto the stack, do arithmetic and other operations on the top of the stack, and print out values in various formats. Up to nine parameters can be used at once. If-then-else testing is possible, as is storage in a limited number of variables. There is no provision for loops or printing strings in any format other than **%s**.

For example, compare the **termcap** entry **cm** and the **terminfo** entry **cup**. **%+** (add space and print as a character) **cm** would be treated as **%p1%' '%+%c**, that is, push the first parameter, push space, add the top two numbers onto the stack, and output the top item on the stack using character (**%c**) format. For the second parameter, change **%p1** to **%p2**. **%.** (print as a character) becomes **%p1%c**. **%d** (print in decimal) becomes **%p1%d**.

As with **tgoto()**, characters standing by themselves (no '%' sign) are output as is.

Alternate Characters

The instruction **acsc** defines a set of alternate characters. These alternate characters define, among other things, the characters used to draw boxes.

acsc is set to a string composed of pairs of characters. The first character in each pair gives the character used by a VT100 in graphics mode to display; the second character is the one for the terminal in use. The following table shows the VT100 graphic-character set:

Arrow right	**+**
Arrow left	**,**
Arrow down	**.**
Full block (inverted space)	**0**
Lantern	**I**
Arrow up	**-**
Diamond	**'**
Checkboard	**a**
Degree	**f**
+/- Sign	**g**
Centered rectangles	**h**
Lower right corner	**j**
Upper right corner	**k**
Upper left corner	**l**
Lower left corner	**m**
Cross	**n**
Upper horizontal line	**o**
Middle horizontal line	**q**
Lower horizontal line	**s**
Left tee	**t**
Right tee	**u**
Lower tee	**v**
Upper tee	**w**
Vertical line	**x**
Centered dot	**~**

Changes from termcap to terminfo

This section describes features of **terminfo** that **termcap** does not contain.

Defaults

terminfo does not contain every default found in **termcap**. **termcap,** for example, assumed that \r was a carriage return unless **nc** was present, indicating that it did not work, or **cr** was present, indicating an alternative. In **terminfo**, if **cr** is present, the string so given works; otherwise it should be assumed *not* to work. The **bs** and **bc** capabilities are replaced by **cub** and **cub1**. (The former takes a parameter, moving left that many spaces. The latter is probably more common in terminals and moves left one space.) **nl** (linefeed) has been split into two functions: **cud1** (moves the cursor down one line) and **ind** (scroll forward). **cud1** applies when the cursor is not on the bottom line, **ind** applies when it is on the bottom line. The bell capability is now explicitly given as **bel**.

The **terminfo** data base is compiled, unlike termcap. This means that a **terminfo** source file (describing some set of terminals) is processed by the terminfo compiler, producing a binary description of the terminal in a file under **/usr/lib/terminfo**. The function **setupterm()** reads this file. The advantage to compilation is that starting up a program using **terminfo** is faster. The increase in speed comes partly from not having to skip past other terminal descriptions, and partly from the compiler having sorted the capabilities into order so that a linear scan can read them in.

The **terminfo** compiler **tic** uses the environment variable **TERMINFO** to be the destination directory of the new object files. It is also used by **setupterm()** to find an entry for a given terminal. First it looks in the directory given in **TERMINFO** and, if not found there, checks **/usr/lib/terminfo**.

Basic Example

The following gives the **terminfo** description for a simple terminal, the Lear-Siegler ADM-3:

```
adm3|3|lsi adm3,
      cr=^M, cud1=^J, ind=^J, bel=^G,
      am, cub1=^H, clear=^Z, lines#24, cols#80
```

As you can see, the description is divided into comma-separated fields. The following discusses each field in detail.

adm3 | 3 | lsi adm3,
: The first field names the terminal. This field is unique in that it is divided into a number of sub-fields, which are separated by vertical bar characters. The first sub-field gives the name by which the terminal is normally addressed in a program; the last gives a longer, descriptive name.

cr=^M, To move the cursor to the left margin, send **<ctrl-M>**.

cud1=^J, To move the cursor down one row, send **<ctrl-J>**.

ind=^J, To scroll the screen up, send **<ctrl-J>**. Note that the ADM-3, like most terminals, does not scroll unless the cursor is on the last row.

bel=^G, To ring the terminal's bell, send **<ctrl-G>**.

am, This boolean code indicates that the ADM-3 wraps to the leftmost column of the of the next row when the cursor reaches the rightmost column.

cub1=^H, To move the cursor nondestructively one column to the left, send **<ctrl-H>**.

clear=^Z, To clear the screen, send **<ctrl-Z>**.

lines#24, The ADM-3 has 24 rows (lines).

cols#80, The ADM-3 has 80 columns.

Modifying an Entry

A full discussion of how to modify a **terminfo** entry is beyond the scope of this article. The references, below, name several volumes that discuss this topic at length.

In brief, modifying a **terminfo** entry requires that you use the command **infocmp** to de-compile the entry for a given terminal, modify the text by hand, then use the command **tic** to recompile and re-install the entry.

C-Level Routines

The library **/usr/lib/libcurses.a** contains a suite of C functions with which you can read a given terminal's **terminfo** capabilities. You must reference the **terminfo** capabilities in your program as global variables whose names are identical to the full names of the capabilities themselves; e.g., **auto_left_margin**. These functions are

declared in the header files **<curses.h> <term.h>,** and **<terminfo.h>**; note that you must **#include** all three header files in your C program.

You can call the following functions from within a C program to read a **terminfo** entry:

fixterm()	Set the terminal into program mode
putp()	Write a string into *stdwin*
resetterm()	Reset the terminal into a saved mode
setupterm()	Initialize terminal capabilities
tparm()	Output a parameterized string
tputs()	Process a capability string
vidattr()	Set the terminal's video attributes
vidputs()	Set video attributes into a function

setupterm() initializes a terminal. It inhales all terminal capabilities at once, and performs all other system-dependent initialization.

A program should call **resetterm()** when it exits or calls a shell escape, to restore the tty modes. When it returns from a shell escape, the program should call **fixterm()** to set the tty modes back to their internal settings.

tparm() is a more powerful, parameterized string mechanism. It resembles the **termcap** function **tgoto()**. **tgoto()** is still available for compatibility. **tputs()** is unchanged.

Files

/usr/lib/libcurses.a — Routines for reading **terminfo** descriptions
/usr/lib/terminfo/?/* — Directories containing compiled descriptions

See Also

Administering COHERENT, captoinfo, curses, fixterm(), infocmp, putp(), resetterm(), setupterm(), term, termcap, tic, tparm(), tputs(), vi, vidputs()
Strang, J., Mui, L., O'Reilly, T.: *Termcap and Terminfo.* Sebastopol, CA: O'Reilly & Associates, Inc., 1991. *Highly recommended.*

Notes

As mentioned above, each **terminfo** description is kept in its own file, in a subdirectory of directory **/usr/lib/terminfo**. Each file is named after the device it describes. Thus, to see what terminal devices have **terminfo** descriptions, type the command:

```
ls -laR /usr/lib/terminfo
```

You may wish to redirect the output of this command into a file, for further study later on.

This implementation of **terminfo** was written by Pavel Curtis of Cornell University. It was ported to COHERENT by Udo Munk.

termio — Device Driver

General terminal interface

COHERENT uses two methods for controlling terminals: **sgtty** and **termio**. To use **sgtty**, simply include the statement **#include <sgtty.h>** in your sources. To use **termio**, include the statement **#include <termio.h>**.

The rest of this article discusses the **termio** method of controlling terminals.

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, users' programs seldom open these files: they are opened by the program **getty** and become a user's standard input, output, and error files. The very first terminal file opened by the process group leader of a terminal file not already associated with a process group becomes the *control terminal* for that process group. The control terminal plays a special role in handling **quit** and **interrupt** signals, as discussed below. The control terminal is inherited by a child process during a call to **fork()**. A process can break this association by changing its process group using **setpgrp()**.

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters can be typed at any time, even while output is occurring, and are lost only when the system's input buffers become completely full, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. Currently, this limit is 256 characters. When the input limit is reached, the system throws away all the saved characters without notice.

Normally, terminal input is processed in units of lines. A line is delimited by a newline character (ASCII LF), an end-of-file character (ASCII EOT), or an end-of-line character. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line is returned. It is not, however, necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, the system normally processes **erase** and **kill** characters. By default, the backspace character erases the last character typed, except that it will not erase beyond the beginning of the line. By default, the **<ctrl-U>** kills (deletes) the entire input line, and optionally outputs a newline character. Both these characters operate on a keystroke-by-keystroke basis, independently of any backspacing or tabbing which may have been done. Both the erase and kill characters may be entered literally by preceding them with the escape character (\). You can change the erase and kill characters.

Certain characters have special functions on input. These functions and their default character values are summarized as follows:

INTR (**<ctrl-C>** or ASCII ETX) generates an *interrupt* signal that is sent to all processes with the associated control terminal. Normally, each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location. For details and a table of legal signals, see the Lexicon entry for **signal()**.

QUIT (**<ctrl-\>** or ASCII ES) generates a *quit* signal. Its treatment is identical to that of the interrupt signal except that, unless a receiving process has made other arrangements, it not only terminates but dumps a core image file (named **core**) into the current working directory.

ERASE (**<backspace>** or ASCII BS) erases the preceding character. It does not erase beyond the start of a line, as delimited by a newline, **EOF**, or EOL character.

KILL (**<ctrl-U>** or ASCII NAK) deletes the entire line, as delimited by a newline, **EOF**, or EOL character.

EOF (**<ctrl-D>** or ASCII EOT) generates an end-of-file character from a terminal. When received, all the characters waiting to be read are immediately passed to the program without waiting for a newline, and the **EOF** is discarded. Thus, if no characters are waiting, which is to say the **EOF** occurred at the beginning of a line, zero characters are passed back; this is the standard end-of-file indication.

NL (ASCII LF) is the normal line delimiter. It cannot be changed or escaped.

EOL (ASCII LF) is an additional line delimiter, like NL. It is not normally used.

STOP (**<ctrl-S>** or ASCII DC3) can be used to suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read. While output is suspended, **STOP** characters are ignored and not read.

START (**<ctrl-Q>** or ASCII DC1) resumes output that has been suspended by a **STOP** character. While output is not suspended, **START** characters are ignored and not read. The **START/STOP** characters cannot be changed or escaped.

You can change the character values for **INTR**, **QUIT**, **ERASE**, **KILL**, **EOF**, and **EOL** To suit your taste. The **ERASE**, **KILL**, and **EOF** character can be preceded by a '\' character, in which case the system ignores its special meaning.

When the carrier signal from the data-set drops, the system sends a *hangup* signal to all processes that have this terminal as their control terminal. Unless other arrangements have been made, this signal causes the process to terminate. If the hangup signal is ignored, any subsequent read returns EOF. Thus, programs that read a terminal and test for end-of-file can terminate appropriately when hung up on.

When one or more characters are written, they are transmitted to the terminal as soon as previously written characters have finished typing. Input characters are echoed by putting them into the output queue as they arrive. If a process produces characters more rapidly than they can be printed, it is suspended when its output queue exceeds a preset limit. When the queue has drained down to that threshold, the program resumes.

Several calls to **ioctl()** apply to terminal files. The primary calls use the following structure, defined in **<termio.h>**:

```
#define NCC 8
struct termio {
        unsigned short c_iflag;    /* input modes */
        unsigned short c_oflag;    /* output modes */
        unsigned short c_cflag;    /* control modes */
        unsigned short c_lflag;    /* local modes */
        char           c_line;     /* line discipline */
        unsigned char  c_cc[NCC];  /* control chars */
};
```

The special control characters are defined by the array **c_cc**. The relative positions and initial values for each function are as follows:

```
0   INTR      ^C
1   QUIT      ^\
2   ERASE     \b
3   KILL      ^U
4   EOF       ^D
5   EOL       \n
6   reserved
7   reserved
```

The field **c_iflag** describes the basic terminal input control:

BRKINT	Signal interrupt on break
IGNPAR	Ignore characters with parity errors
INPCK	Enable input parity check
ISTRIP	Strip character
ICRNL	Map CR to NL on input
IXON	Enable start/stop output control
IXOFF	Enable start/stop input control

If **INPCK** is set, input parity checking is enabled. If it is not set, then checking is disabled. This allows output parity generation without input parity errors.

If **ISTRIP** is set, valid input characters are stripped to seven bits before being processed; otherwise, all eight bits are processed.

If **IXON** is set, **START/STOP** output control is enabled. A received **STOP** character suspends output and a received **START** character restarts output. All start/stop characters are ignored and not read.

If **IXOFF** is set, the system transmits **START/STOP** characters when the input queue is nearly empty or nearly full.

The initial input control value is all bits clear.

The field **c_oflag** field specifies the system treatment of output:

OPOST	Postprocess output.
OLCUC	Map lower case to upper on output.
ONLCR	Map NL to CR-NL on output.

If **OPOST** is set, output characters are post-processed as indicated by the remaining flags; otherwise, characters are transmitted without change.

If **OLCUC** is set, a lower-case alphabetic character is transmitted as the corresponding upper-case character. This function is often used with **IUCLC**.

If **ONLCR** is set, the NL character is transmitted as the CR-NL character pair.

The initial output control value is all bits clear.

The field **c_cflag** describes the hardware control of the terminal, as follows:

CBAUD	Baud rate
B0	Hang up
B50	50 baud
B75	75 baud
B110	110 baud
B134	134.5 baud

B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19200 baud
B38400	38400 baud
CREAD	Enable receiver
PARENB	Parity enable
PARODD	Odd parity, else even
HUPCL	Hang up on last close
CLOCAL	Local line, else dial-up

The **CBAUD** bits specify the baud rate. The zero-baud rate, **B0**, hangs up the connection. If **B0** is specified, the data-terminal-ready signal is not asserted. Normally, this disconnects the line. For any particular hardware, the system ignores impossible changes to the speed.

If **PARENB** is set, parity generation and detection is enabled and a parity bit is added to each character. If parity is enabled, the **PARODD** flag specifies odd parity if set; otherwise, even parity is used.

If **CREAD** is set, the receiver is enabled. Otherwise, no characters will be received.

If **HUPCL** is set, COHERENT disconnects the line when the last process with the line open closes the line or terminates; that is, the data-terminal-ready signal is not asserted.

If **CLOCAL** is set, the system assumes that the line to be a local, direct connection with no modem control. Otherwise, it assumes modem control.

The line discipline uses the field **c_lflag** to control terminal functions. The basic line discipline (zero) provides the following:

ISIG	Enable signals
ICANON	Canonical input (erase and kill processing)
XCASE	Canonical upper/lower presentation
ECHO	Enable echo
ECHOE	Echo erase character as BS-SP-BS
ECHOK	Echo NL after kill character
ECHONL	Echo NL

The following gives the meaning of each flag in detail:

ISIG If this flag is set, the system checks each input character against the special control characters **INTR** and **QUIT**. If an input character matches one of these control characters, the system executes the function associated with that character. If it is not set is not set, the system performs no checking; thus, these special input functions are possible only if **ISIG** is set. You can disable these functions individually by changing the value of the control character to an unlikely or impossible value (e.g., 0377).

ICANON

If this flag is set, the system enables canonical processing. This enables the erase and kill-edit functions, and limits the assembly of input characters into lines delimited by NL, EOF, and EOL. The system also interprets the *vmin* and *vtime* locations in the **termio** structure as **c_cc[VEOF]** and **c_cc[VEOL]**, respectively.

When the **ICANON** bit is cleared, you must set **c_cc[VMIN]** and **c_cc[VTIME]** to appropriate *vmin* and *vtime* values. *vmin* is a number from 0 to 255 that gives the minimum number of characters required before any read operation completes. *vtime* is a number from 0 to 255 that specifies how long, in tenths of a second, to wait for completion of input. The following describes how **termio** processes the *vmin* and *vtime* values:

1. If *vmin* is greater than zero and *vtime* equals zero, block until *vmin* characters are received.
2. If both *vmin* and *vtime* are greater than zero, block until the first character is received, then return after *vmin* characters are received or *vtime*/10 seconds have elapsed since the last character was received, whichever occurs first.

3. If *vmin* equals zero, return after first character is received or after *vtime*/10 seconds have passed, whichever occurs first. It may return a read count of zero — but will return one character if it is available, even if *vtime* is zero.

You can use the command **stty** to reset the *vmin* and *vtime* values. The header file **termio.h** includes the constants **VMIN** and **VTIME**, which set default values for *vmin* and *vtime*, respectively.

XCASE If this flag is set, and if **ICANON** is set, an upper-case letter is accepted on input by preceding it with a '\' character, and is output preceded by a '\' character. In this mode, the following escape sequences are generated on output and accepted on input:

For:	*Use:*
`	\'
|	\!
~	\^
{	\(
}	\)
\	\\

For example, **A** is input as **\a**, **\n** as **\\n**, and **\N** as **\\\n**.

ECHO If this flag is set, characters are echoed as received. When **ICANON** is set, the following echo functions are possible:

- If **ECHO** and **ECHOE** are set, the erase character is echoed as ASCII BS SP BS, which clears the last character from the screen.
- If **ECHOE** is set and **ECHO** is not set, the erase character is echoed as ASCII SP BS.
- If **ECHOK** is set, the NL character is echoed after the kill character to emphasize that the line will be deleted. Note that an escape character preceding the erase or kill character removes any special function.
- If **ECHONL** is set, the NL character is echoed even if **ECHO** is not set. This is useful for terminals set to local echo ("half duplex").

Unless escaped, the EOF character is not echoed. Because EOT is the default EOF character, this prevents terminals that respond to EOT from hanging up.

The initial line-discipline control value is all bits clear.

The primary calls to **ioctl()** have the following form:

```
ioctl( fildes, command, arg )
struct termio *arg;
```

The following commands use this form:

TCGETA Get the parameters associated with the terminal and store in the **termio** structure referenced by **arg**.

TCSETA Set the parameters associated with the terminal from the structure referenced by **arg**. The change is immediate.

TCSETAW Wait for the output to drain before setting the new parameters. This form should be used when changing parameters that affect output.

TCSETAF Wait for the output to drain, then flush the input queue and set the new parameters.

Additional calls to **ioctl()** have the following form:

```
ioctl( fildes, command, arg )
int arg;
```

The following command uses this form:

TCFLSH
Flush both the input and output queues.

Note that header **<termio.h>** defines other constants for purposes of portability. Features designated by these constants are unavailable in the current release of COHERENT 386.

LEXICON

Example

The following example gives some functions that let you perform a non-blocking read of the keyboard — that is, a

```
read(0, &c, sizeof(char));
```

that returns zero (failure) rather than waiting for input if there is no current typed character.

To do so, you must do the following:

- Set up keyboard input appropriately with the ioctls **TCGETA** and **TCSETA**.
- Turn off **ICANON**.
- Turn off the various versions of **ECHO**.
- Use **ISIG** to disable keyboard interrupts.
- Finally, set:

```
termiob.c_cc[VMIN] = 0;
termiob.c_cc[VTIME] = 0;
```

This lets **read()** return after reading zero bytes in .0 seconds.

```
#include <termio.h>
#include <stdlib.h>

void
ttyinit()
{
      struct      termio      termiob;

      ioctl(0, TCGETA, &termiob); /* get tty characteristics */
      termiob.c_cc[VMIN] = 0;
      termiob.c_cc[VTIME] = 0; /* non-blocking read */
      ioctl(0, TCSETA, &termiob); /* set new mode */
}

int
ttycheck()
{
      static int done = 0;
      char c;

      if (done)
            return 0;
      if (read(0, &c, 1) != 0) {
            if (c == 'a')
                  return 0;
            else if (c != ' ') {
                  ++done;
                  return 0;
            }

            /* After <space>, pause until another character is typed */
            while (read(0, &c, 1) == 0)
                  ;
      }
      return 1;
}

main()
{
      ttyinit();

      while (1) {
            printf("Still checking ...\n");
            if (!ttycheck())
                  exit(EXIT_SUCCESS);
      }
}
```

For another example of how to manipulate the **termio** structure, see the entry for **ioctl()**.

Files

/dev/tty*

See Also

device drivers, ioctl(), stty, terminal, termio.h, termios
POSIX Standard, §7.1

Notes

The version of **stty** that is supplied with COHERENT 386 provides complete access to the System-V-style **termio** structure. This lets you specify and view any combination of the fields therein, including various delays. How these fields are processed, however, depends on the device in question. The settings of **termio** are processed by the kernel's in-line discipline and device-driver modules. In COHERENT 4.0.1, none of these modules pays attention to delay settings.

termio.h — Header File

Definitions used with terminal input and output
#include <termio.h>

termio.h defines structures and constants used by functions that control terminal input and output.

See Also

header files, termio
POSIX Standard, §7.1.2

Notes

COHERENT lets you choose between **sgtty** and **termio** to control terminals. For more information, see the Lexicon entries for **sgtty** and **termio**.

termios — Overview

POSIX extended terminal interface

The name **termios** describes a group of routines that POSIX Standard defines to extend the **termio** interface to terminals. **termios** includes the following routines:

cfgetispeed() . . . . . Get input speed
cfgetospeed() . . . . . Get output speed
cfsetispeed(). . . . . Set input speed
cfsetospeed() . . . . . Set output speed
tcdrain() . . . . . Drain output to a device
tcflow(). . . . . Control flow on a terminal device
tcflush() . . . . . Flush data being exchanged with a terminal
tcgetattr() . . . . . Get terminal attributes
tcsendbreak() . . . . . Send a break to a terminal
tcsetattr() . . . . . Set terminal attributes

Each is described in its own Lexicon entry. Under COHERENT, all are defined as macros in header file **<termios.h>**.

Example

The following example returns the input and output speeds for the terminal device that you now are using:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <termios.h>

int main()
{
      struct termios term;
      int speed;
```

```
      if (tcgetattr(STDIN_FILENO, &term) < 0) {
            fprintf(stderr, "tcgetattr error");
            exit(EXIT_FAILURE);
      }

      speed = cfgetispeed(&term);
      printf("tty line input speed is ");

      if      (speed == B50)        printf("50 baud\n");
      else if (speed == B75)        printf("75 baud\n");
      else if (speed == B110)       printf("110 baud\n");
      else if (speed == B134)       printf("134 baud\n");
      else if (speed == B150)       printf("150 baud\n");
      else if (speed == B200)       printf("200 baud\n");
      else if (speed == B300)       printf("300 baud\n");
      else if (speed == B600)       printf("600 baud\n");
      else if (speed == B1200)      printf("1200 baud\n");
      else if (speed == B1800)      printf("1800 baud\n");
      else if (speed == B2400)      printf("2400 baud\n");
      else if (speed == B4800)      printf("4800 baud\n");
      else if (speed == B9600)      printf("9600 baud\n");
      else if (speed == B19200)     printf("19200 baud\n");
      else if (speed == B38400)     printf("38400 baud\n");
      else                    printf("unknown speed\n");

      speed = cfgetospeed(&term);
      printf("tty line output speed is ");

      if      (speed == B50)        printf("50 baud\n");
      else if (speed == B75)        printf("75 baud\n");
      else if (speed == B110)       printf("110 baud\n");
      else if (speed == B134)       printf("134 baud\n");
      else if (speed == B150)       printf("150 baud\n");
      else if (speed == B200)       printf("200 baud\n");
      else if (speed == B300)       printf("300 baud\n");
      else if (speed == B600)       printf("600 baud\n");
      else if (speed == B1200)      printf("1200 baud\n");
      else if (speed == B1800)      printf("1800 baud\n");
      else if (speed == B2400)      printf("2400 baud\n");
      else if (speed == B4800)      printf("4800 baud\n");
      else if (speed == B9600)      printf("9600 baud\n");
      else if (speed == B19200)     printf("19200 baud\n");
      else if (speed == B38400)     printf("38400 baud\n");
      else                    printf("unknown speed\n");

      exit(EXIT_SUCCESS);
}
```

See Also

cfgetispeed(), cfgetospeed(), cfsetispeed(), cfsetospeed(), Programming COHERENT, tcdrain(), tcflow(), tcflush(), tcgetattr(), tcsendbreak(), tcsetattr(), termio, termios.h

Notes

If a program that uses **termios** has set the **termio** flag **ISIG** (which enables signals) and receives character **SUSP** (normally **<ctrl-Z>**), it sends the signal **SIGTSTP** to the current process group. By default, **termios** then discards **SUSP**. Character **SUSP**, as its name implies, tells a program to suspend operation and recede into the background. Please note that because COHERENT does not yet support job control, **SUSP** at present will do nothing.

termios.h — Header File

Definitions used with POSIX extended terminal interface
#include <termios.h>

Header file **<termios.h>** defines the structures and macros that implement the POSIX Standard's extensions to the **termio** interface.

See Also

header files, termios

test — Command

Evaluate conditional expression
test *expression ...*

test evaluates an *expression*, which consists of string comparisons, numerical comparisons, and tests of file attributes. For example, a **test** command might be used within a shell script to test whether a certain file exists and is readable.

The logical result (true or false) of the *expression* is returned by the command for use by another shell construct, such as the command **if**.

Under the Korn shell, **test** is a built-in command that returns zero if *expression* is true, and one if it is false. Under the Bourne shell, **test** is not a built-in command; rather, the Bourne shell uses the command **/bin/test** to test expressions. **/bin/sh** returns zero if the expression is true, one if it is false, and two if a syntax error (or other error) occurred.

Expression Options

test recognizes the following options, one or more of which can be built into an *expression*:

! *exp*	Negates the logical value of expression *exp*.
string1 **!=** *string2*	*string1* is not equal to *string2*.
string1 **<** *string2*	*string1* is lexicographically less than *string2* (**sh** only).
string1 **=** *string2*	*string1* is equal to *string2*.
string1 **>** *string2*	*string1* is lexicographically greater than *string2* (**sh** only).
(*exp***)**	Parentheses allow expression grouping.
exp1 **-a** *exp2*	Both expressions *exp1* and *exp2* are true.
-b *file*	*file* is a block-special device.
-c *file*	*file* is a character-special file.
-d *file*	*file* exists and is a directory.
-e *file*	*file* exists (**/bin/test** only).
file1 **-ef** *file2*	*file1* is the same file as *file2*.
n1 **-eq** *n2*	Numbers *n1* and *n2* are equal. Please note that **test** evalutes the expression as zero. Thus, if one of the arguments is a variable that is not set, **test** treats it as if it were zero. For example, consider the expression: `if [ "$notset" -eq 0 ]` If **notset** is not set, **test** evaluates it to zero and so returns **true**.
-f *file*	*file* exists and is an ordinary file.
-g *file*	File mode has **setgid** bit.
n1 **-ge** *n2*	Number *n1* is greater than or equal to *n2.*
n1 **-gt** *n2*	Number *n1* is greater than *n2.*
-k *file*	File mode has sticky bit.
-L *file*	File is a symbolic link.
n1 **-le** *n2*	Number *n1* is less than or equal to *n2.*
n1 **-lt** *n2*	Number *n1* is less than *n2.*
-n *string*	*string* has nonzero length.
n1 **-ne** *n2*	Numbers *n1* and *n2* are not equal.
file1 **-nt** *file2*	*file1* is newer than *file2*.
exp1 **-o** *exp2*	Either expression *exp1* or *exp2* is true. **-a** has greater precedence than **-o**.
file1 **-ot** *file2*	*file1* is older than *file2*.
-p *file*	*file* is a named pipe.
-r *file*	*file* exists and is readable.
-s *file*	*file* exists and has nonzero size.
-t *[fd]*	*fd* is the file descriptor number of a file that is open and a terminal. The Bourne shell requires that *fd* be given; under the Korn shell, however, defaults to the standard output (file descriptor 1) if no *fd* is given.
-u *file*	File mode has **setuid** set.
-w *file*	*file* exists and is writable.
-x *file*	*file* exists and executable.
-z *string*	*string* has zero length (is a null string).
string	*string* has nonzero length.

LEXICON

Implementations of test

The implementation of **test** under the Bourne shell has been rewritten for COHERENT release 4.2, both to extend its range of features and to make it more compliant with published standards. Although this makes **test** more useful to programmers, it may create problems when you try to execute a Bourne-shell script written under COHERENT release 4.2 on an earlier release of COHERENT. The following describes how the Bourne shell's implementation of **test** was designed; and how it differs both from earlier implementations under the Bourne shell and from the implementation under the Korn shell.

To begin, the Bourne shell's implementation of **test** attempts to comply with the POSIX Standard, comply with previous COHERENT releases of **test**, and comply with System-V UNIX to the greatest extent possible. However, these objectives are mutually exclusive. See the POSIX Standard P1003.2/D11.2 §4.62, especially the Rationale, for details of some of the problems. In particular, System V and Berkeley differ in the way they parse some expressions, which leads the POSIX Standard to specify test behavior for a minimal set of expressions, *not* including **-a** and **-o**.

The following details differences among the various implementations of **test**. First, the following options were not implemented in the Bourne shell's implementation of **test** prior to COHERENT release 4.2, but were included in the Korn shell's implementation: **-b**, **-c**, **-ef**, **-g**, **-k**, **-L**, **-nt**, **-ot**, **-p**, **-u**, and **-x**. Of these, the following are not described in the POSIX Standard: **-k**, **-L**, **-ef**, **-nt**, and **-ot**. Note that Bourne-shell scripts that use any of the above options to **test** will *not* run on versions of COHERENT prior to release 4.2, but will run under the Korn shell.

Next, the Bourne shell for COHERENT 4.2 implements the POSIX Standard's option **-e** and the options **<** and **>**. Bourne-shell scripts that use any of these three options to **test** will *not* run on versions of COHERENT prior to release 4.2, nor will they run under the Korn shell.

The definitions of the options **-f** and **-t** have been changed from the Berkeley standard to that described in the POSIX Standard. Berkeley defines **-f** as meaning that a file exists and is not a directory; whereas the POSIX Standard defines it as meaning that a file exists and is a regular file. Versions of the Bourne shell prior to COHERENT 4.2 use the Berkeley definitions; whereas all version of the Korn shell and Bourne shell under COHERENT 4.2 use the POSIX Standard's definition. Berkeley gives **-t** a default value of one if it is not used with an argument; whereas the POSIX Standard requires that **-t** have an argument. The Korn shell and all versions of the Bourne shell prior to COHERENT 4.2 use the Berkeley definition; whereas the Bourne shell under COHERENT 4.2 uses the POSIX Standard's definition. These differences are subtle, but important. Thus, a Bourne shell script that uses either of these options may not run correctly when imported into COHERENT 4.2 from earlier versions of COHERENT, or when exported from COHERENT 4.2 to them or to the Korn shell.

Finally, **test** under the Korn shell and under the Bourne shell prior to COHERENT 4.2 returns zero if an expression is true and one either if the expression is false or if the expression contained a syntax error. However, **test** under the Bourne shell for COHERENT 4.2 returns zero if an expression is true, one if it is false, and two if a syntax error occurred. Bourne-shell scripts that pay close attention to what **test** returns may not run correctly when imported into COHERENT 4.2 from earlier implementations of COHERENT, or when exported from COHERENT 4.2 to earlier versions of COHERENT or to the Korn shell.

Example

The following example uses the **test** command to determine whether a file is writable.

```
if test ! -w /dev/lp
then
      echo The line printer is inaccessible.
fi
```

Under COHERENT, the command '[' is linked to **test**. If invoked as '[', **test** checks that its last argument is ']'. This allows an alternative syntax: simply enclose *expression* in square brackets. For example, the above example can be written as follows:

```
if [ ! -w /dev/lp ]
then
      echo The line printer is inaccessible.
fi
```

For a more extended example of the square-bracket syntax, see **sh**.

See Also

commands, expr, find, if, ksh, sh, while

Notes

The Korn shell's version of this command is based on the public-domain version written by Erik Baalbergen and Arnold Robbins.

tgetent() — termcap Function (libterm)

Read termcap entry
#include <curses.h>
#include <term.h>
int tgetent(*bp, name***)**
char **bp, *name**;

tgetent() is one of a set of functions that read a **termcap** terminal description. It extracts the entry from file **/etc/termcap** for the terminal *name* and writes it into a buffer at address *bp*. *bp* should be a character buffer of 1,024 bytes and must be retained through all subsequent calls to the other functions. It returns **-1** if it cannot open **/etc/termcap**, zero if the terminal name given does not have an entry, and one upon a successful search.

tgetent() first looks in the environment to see if the **termcap** variable had already been set. If it finds that the variable **termcap** has been set, that the value does *not* begin with a slash, and that the terminal type name in the **termcap** variable is the same as that in the environment variable **TERM**, then **tgetent()** uses the **termcap** string instead of reading the file **/etc/termcap**. However, if the **termcap** string does begin with a slash, then it is used as the path name of a terminal-capabilities file other than **/etc/termcap.** This can speed entry into programs that call **tgetent()**, and can be used to help debug new terminal descriptions.

Files

/etc/termcap — Terminal capabilities data base
/usr/lib/libterm.a — Function library

See Also

termcap

tgetflag() — termcap Function

Get termcap Boolean entry
#include <curses.h>
#include <term.h>
int tgetflag(*name***)**
char **name*;

tgetflag() is one of a set of functions that read a **termcap** terminal description. It returns one if the requested Boolean capability *name* is present in the terminal's **termcap** entry, zero if it is not.

Files

/etc/termcap — Terminal capabilities data base
/usr/lib/libterm.a — Function library

See Also

termcap

tgetnum() — termcap Function (libterm)

Get termcap numeric feature
#include <curses.h>
#include <term.h>
int tgetnum(*name***)**
char **name*;

tgetnum() is one of a set of functions that read a **termcap** terminal description. It returns the value of the numeric feature *name*, as defined in the terminal's **termcap** entry. It returns **-1** if the feature is not present in the terminal's entry.

Files

/etc/termcap — Terminal capabilities data base
/usr/lib/libterm.a — Function library

See Also

termcap

tgetstr() — termcap Function (libterm)

Get termcap string entry
#include <curses.h>
#include <term.h>
char ***tgetstr(***name, area***)**
char **name***, ***area***;**

tgetstr() is one of a set of functions that read a **termcap** terminal description. It reads the string value of feature *name* from the terminal's **termcap** description, and writes it into the buffer at address *area*. It also advances the value of the pointer to *area*.

tgetstr() decodes the abbreviations for the fields used in the **termcap** entry, except for padding and for cursor-addressing information.

Files

/etc/termcap — Terminal capabilities data base
/usr/lib/libterm.a — Function library

See Also

termcap

tgoto() — termcap Function (libterm)

Read/interpret termcap cursor-addressing string
#include <curses.h>
#include <term.h>
char ***tgoto(***cm, destcol, destline***)**
char **cm***; **int** scrcol***,** scrline***;***

tgoto() is one of a set of functions that read a **termcap** terminal description. It decodes a cursor-addressing string from the *cm* **termcap** feature, and writes it onto the screen, at column *scrcol* and line *destline*. **tgoto()** uses the external variables **UP** (from the **up** feature) and **BC** (if **bc** is given rather than **bs**) if it is necessary to avoid placing **\n, <ctrl-D>**, or **<ctrl-@>** into the returned string. Programs calling **tgoto()** should turn off the **XTABS** bits, as **tgoto()** may write a tab. If a '%' sequence is given that is not understood, **tgoto()** returns "OOPS".

Files

/etc/termcap — Terminal capabilities data base
/usr/lib/libterm.a — Function library

See Also

termcap

tic — Command

Compile a terminfo description
tic [-v[*n***]]** *sourcefile*

The command **tic** compiles a *sourcefile* of **terminfo** information into a binary object.

sourcefile must be self-contained, i.e., it may not contain "use" entries that refer to terminals not described fully in the same file.

The object files generated by **tic** are normally placed into subdirectories of the directory **/usr/lib/terminfo**. If the environment variable **TERMINFO** is defined, it is assumed to name an alternative directory to use.

The flag **-v***n* tells **tic** to output debugging and tracing information. *n* sets the amount of debugging information to produce, as follows:

1 Names of files created
2 Information related to the "use" facility
3 Statistics from the hashing algorithm
5 String-table memory allocations
7 Entries into the string-table
8 List of tokens encountered by scanner
9 All values computed in construction of the hash table

n is set to one by default.

Files

/usr/lib/terminfo/* — Default location of object files

See Also

commands, infocmp, terminfo, term
Strang, J., Mui, L., O'Reilly, T.: *termcap and terminfo*. Sebastopol, CA: O'Reilly & Associates, Inc., 1991.

Notes

tic was written by Pavel Curtis of Cornell University. It was ported to COHERENT by Udo Munk.

time — Overview

COHERENT includes a number of routines that allow you to set and manipulate time, as recorded on the system's clock, into a variety of formats. These routines should be adequate for nearly any task that involves temporal calculations or the maintenance of data gathered over a long period of time.

All functions, global variables, and manifest constants used in connection with time are defined and described in the header files **time.h** and **timeb.h**. In brief, time manipulates two data elements: the type **time_t**, and the structure **tm**.

time_t is defined in the header file **<time.h>**. COHERENT follows the UNIX standard and initializes **time_t** to the number of seconds since January 1, 1970, 0h00m00s GMT; this moment, in turn, is rendered as day 2,440,587.5 on the Julian calendar. This allows accurate calculation of time as far back as January 1, 4713 B.C.

The structure **tm** is defined in the header file **<time.h>**. It is defined as follows:

```
struct tm {
        int tm_sec;             /* current time, seconds */
        int tm_min;             /* current time, minutes */
        int tm_hour;            /* current time, hour */
        int tm_mday;            /* day of the month, 1-31 */
        int tm_mon;             /* month, 1-12 */
        int tm_year;            /* year since 1900 */
        int tm_wday;            /* day of the week, Sunday = 0 */
        int tm_yday;            /* day in the current year */
        int tm_isdst;           /* is daylight-savings time now in effect? */
};
```

For an example of manipulating this structure in a C program, see the Lexicon entry for **localtime()**.

Standard Time Functions

The COHERENT system includes the following functions to manipulate time:

asctime() . . . . . Convert time structure to ASCII string
clock() . . . . . Get processor time
ctime() . . . . . Convert system time to an ASCII string
difftime() . . . . . Return difference between two times
gmtime() . . . . . Convert system time to calendar structure
localtime() . . . . . Convert system time to calendar structure
mktime() . . . . . Turn broken-down time into calendar time
strftime() . . . . . Format locale-specific time
time() . . . . . Get the current time
tzset() . . . . . Set local time zone

To print out the local time, a program must perform the following tasks:

1. Read the system time with **time()**. This function returns a **time_t**.
2. Pass the **time_t** returned by **time()** to the function **localtime()**. This function breaks it down into the **tm** structure,
3. Pass **localtime()**'s output to **asctime()**, which transforms the **tm** structure into an ASCII string.
4. Finally, pass the output of **asctime()** to **printf()**, to displays it on the standard output device.

See the entry for **asctime()** for an example C program that goes through the above steps.

Special Time Functions

COHERENT includes a number of extensions to the time functions used by standard UNIX systems. These are designed to increase the scope and accuracy of time-handling, and to ease calculation of some time elements.

COHERENT holds information about your time locale in the environmental variable **TIMEZONE**. This variable is described in detail in its Lexicon article. In brief, it consists of seven fields:

1. Name of the local standard time zone
2. Offset from Greenwich Mean Time, in minutes
3. Name of the local daylight time zone
4. Date when daylight-savings time begins
5. Date when daylight-savings time ends
6. Hour when daylight-savings time begins
7. Hour when daylight-savings time ends

The fields are defined in such a way that any form of daylight-saving adjustment can be handled correctly. For example, the United States shifts into daylight-savings time on the first Sunday in April; whereas Brazil shifts into daylight-savings time on a set day each spring.

The function **tzset()** parses **TIMEZONE** into the following external variables:

timezone Seconds from GMT to give local time
tzname[2][16] Character array of names of standard and daylight times

For details on manipulations these variable, see the Lexicon entry for **tzset()**. The library **libmisc.a** contains the following functions that convert time from Julian to Gregorian form:

time_to_jday() Convert **time_t** to the Julian date
jday_to_time() Convert Julian date to **time_t**
tm_to_jday() Convert **tm** structure to Julian date
jday_to_tm() Convert Julian date to **tm** structure

COHERENT's time functions assume that conversion to the Gregorian calendar occurred October 1582, when it was first adopted in Rome. However, various nations adopted the Gregorian calendar at different times; for example, it was adopted in the British Empire (including its American colonies) only in September 1752. (This, by the way, is the date assumed by the COHERENT command **cal**, as you would see if you typed the command **cal 9 1752**.) Users in northern and eastern Europe, and in European-influenced areas of Asia (e.g., India) may wish to to write their own functions to convert historical data properly from the Julian to the Gregorian calendar.

Example

For an example of some time functions, see the entry for **asctime()**.

See Also

cal, libc, libmisc

Notes

COHERENT also includes the system call **ftime()**, which returns the current system time. Because the ANSI Standard eliminates **ftime()**, users are urged to replace this system call with calls to **time()**.

UNIX System V defines **time_t** in header file **<sys/types.h>**, whereas COHERENT defines it in **time.h**. This should not affect the porting of programs from UNIX to COHERENT, but it may affect the porting of programs in the other direction.

time — Command

Time the execution of a command
time [*command***]**

time invokes the given *command* with any arguments provided. Upon termination, **time** prints the elapsed real time, CPU time in the system, and CPU time in the user program on the standard error output.

See Also

commands, date, ps, times

Diagnostics

If the *command* terminates abnormally, **time** displays an error message that explains why.

time.h — Header File

Give time-description structure
#include <time.h>

The header file **time.h** prototypes the routines that COHERENT uses to manipulate time, and declares the constants and data types they use.

See Also

header files, time [overview]
ANSI Standard, §7.12

time() — System Call (libc)

Get current system time
#include <time.h>
time_t time(*tp***)**
time_t ****tp*;**

time() reads and returns the current system time. COHERENT defines the current system time as the number of seconds since January 1, 1970, 0h00m00s GMT.

tp points to a data element of the type **time_t**, which the header file **time.h** defines as being equivalent to a **long**. If *tp* is initialized to a value other than NULL, then **time()** attempts to write the system time into the address to which *tp* points. If, however, *tp* is initialized to NULL, then **time()** returns the current system time but does not attempt to write it anywhere.

Example

For an example of this call, see the entry for **asctime()**.

See Also

date, libc, time [overview], time.h
ANSI Standard, §7.12.2.4
POSIX Standard, §4.5.1

Notes

UNIX System V defines **time_t** in header file **<sys/types.h>**, whereas COHERENT defines it in **time.h**. This should not affect the porting of programs from UNIX to COHERENT, but it may affect the porting of programs in the other direction.

timeb.h — Header File

Define timeb structure
#include <sys/timeb.h>

The header file **timeb.h** defines the structure **timeb**, which is used by the function **ftime()** to return time information.

See Also

ftime(), header files, time [overview]

timeout.h — Header File

Define the timer queue
#include <timeout.h>

timeout.h defines the timeout queue. The timeout queue can, as its name implies, be used to call a function when a process has "timed out".

See Also

header files

Notes

This header file is obsolete, and will be dropped from a future release of COHERENT. Its use is strongly discouraged.

times — Command

Print total user and system times
times

times prints the total elapsed user time and system time for the current shell and all its children. It gives each time in minutes, seconds and tenths of seconds. For example,

```
1m11.8s 1m35.8s
```

indicates a total user time of 1 minute 11.8 seconds, and a total system time of 1 minute 35.8 seconds.

The shell executes **times** directly.

See Also

commands, ksh, time, sh

times.h — Header File

Definitions used with times() system call
#include <sys/times.h>

times.h defines the structure **tms**, which is used by the system call **times()**.

See Also

header files, times()
POSIX Standard, §4.5.2

times() — System Call (libc)

Obtain process execution times
#include <sys/times.h>
#include <time.h>
int times(*tbp***)**
struct tms **tbp***;*

times() reads CPU time information about the current process and its children, and writes it into the structure pointed to by *tbp*. The structure **tms** is declared in the header file **sys/times.h**, as follows:

```
struct tms {
        clock_t tms_utime;                  /* process user time */
        clock_t tms_stime;                  /* process system time */
        clock_t tms_cutime;                 /* childrens' user times */
        clock_t tms_cstime;                 /* childrens' system times */
};
```

All of the times are measured in basic machine cycles, or **CLK_TCK**.

The childrens' times include the sum of the times of all terminated child processes of the current process and of all of their children. The *user* time represents execution time of user code, whereas *system* time represents system overhead, such as executing system calls, processing signals, and other monitoring functions.

times() returns the number of ticks that have passed since system startup.

Files

<sys/times.h>
<time.h>

See Also

acct(), ftime(), libc, time() times.h,
POSIX Standard, §4.5.2

TIMEZONE — Environmental Variable

Time zone information
TIMEZONE=*standard***:***offset***[:***daylight***: ***date***:***date***:***hour***:***minutes***]**

The COHERENT system records time internally as Greenwich Mean Time (GMT). It does so to make it easier to coordinate exchange of information across systems in different time zones around the world.

TIMEZONE is an environmental parameter that holds information about your local time zone. This information is used by COHERENT's time routines to convert GMT to the date and time in your local area. **TIMEZONE** takes into account your local time zone's offset from Greenwich, whether your country uses daylight savings time, and the date and hour that daylight savings time begins and ends.

To set **TIMEZONE**, use the command

export TIMEZONE=[*description***]**

where *description* is the string that describes your time zone. What this string consists of will be described below. Most users write this command into the file **.profile**, so that **TIMEZONE** is set automatically whenever they log onto the COHERENT system.

COHERENT's installation procedure creates file **/etc/timezone**, which sets **TIMEZONE**. This file is executed by **/etc/profile** when each user logs in. Thus, you must set the **TIMEZONE** in your **.profile** only if it differs from the system's **TIMEZONE** as set in **/etc/timezone**. This would be necessary if, for example, a user in New York were to regularly login on a system in Chicago.

The Description String

A **TIMEZONE** description string consists of seven fields that are separated by colons. Fields 1 and 2 must be filled; fields 3 through 7 are optional.

Field 1 gives the name of your standard time zone. Field 2 gives the time zone's offset from Greenwich Mean Time in minutes. Offsets are positive for time zones west of Greenwich and negative for time zones east of Greenwich. For example, users in Chicago set these fields as follows:

```
TIMEZONE=CST:360
```

CST is an abbreviation for Central Standard Time, that area's time zone; and 360 refers to the fact that Chicago's time zone is 360 minutes (six hours) ahead of (that is, earlier than) Greenwich.

Field 3 gives the name of the local daylight saving time zone. In Chicago, for example, this field would be set as follows:

```
TIMEZONE=CST:360:CDT
```

CDT is an abbreviation for Central Daylight Time. The absence of this field indicates that your area does not use daylight saving time.

Fields 4 and 5 specify the dates on which daylight saving time begins and ends. If field 3 is set but fields 4 and 5 are not, changes between standard time and daylight saving time are assumed to occur at the times legislated in the United States: at 2 A.M. standard time on the first Sunday in April, and at 2 A.M. daylight saving time on the last Sunday in October.

Fields 4 and 5 each consist of three numbers separated by periods. The first number specifies which occurrence of the day in the month marks the change, counting positive occurrences from the beginning of the month and negative occurrences from the the end of the month. The second number specifies a day of the week, numbering Sunday as one. The third number specifies a month of the year, numbering January as one. For example, in Chicago fields 4 and 5 are set to the following:

```
TIMEZONE=CST:360:CDT:1.1.4:-1.1.10
```

If the first number in either field is set to zero, then the last two numbers are assumed to indicate an absolute date. This is done because some countries switch to daylight saving time on the same day each year, instead of a given day of the week.

Finally, fields 6 and 7 specify the hour of the day at which daylight saving time begins and ends, and the number of minutes of adjustment. In Chicago, these are set as follows:

```
TIMEZONE=CST:360:CDT:1.1.4:-1.1.10:2:60
```

The '2' of field 6 indicates that the switch to daylight savings time occurs at 2 A.M. The "60" of field 7 indicates that daylight savings time changes the local time by 60 minutes. Although 60 minutes is the standard change, some regions of the world shift by 30, 45, 90, or 120 minutes; the last shift is also called "double daylight saving time".

For an example of this variable's use in a program, see the entry for **asctime()**.

See Also

environmental variables, time [overview]

Notes

File **/etc/default/login** defines **TIMEZONE** differently: it uses the same format as the COHERENT environmental variable **TZ**, which is set in file **/etc/timezone**. Note that **TZ** and **TIMEZONE** as defined in **/etc/default/login** must be identical, or much confusion will result.

For those requiring more information on this subject, much research has been performed by astrologers. See *Time Changes in the World*, compiled by Doris Chase Doane (three volumes, Hollywood, California, Professional Astrologers, Inc., 1970).

TMPDIR — Environmental Variable

Directory that holds temporary files

The command **cc** reads the environmental variable **TMPDIR** to see where you want it to write its temporary files. You can speed compilation by building a RAM disk and pointing **TMPDIR** to point at it.

For example, if you have created a RAM disk and mounted it as **/z**, then by embedding the instruction

```
export TMPDIR=/z/tmp
```

in your **.profile**, you can ensure that **cc** will write all of its temporary files onto the RAM disk.

See Also

cc, environmental variables, ram

tmpfile() — STDIO Function (libc)

Create a temporary file
#include <stdio.h>
FILE *tmpfile(void);

The function **tmpfile()** creates a file to hold data temporarily. The file is opened into binary update mode (**wb+**) and is removed automatically when it is closed or when the program exits. There is no way to access the temporary file by name. If your program needs to do so, it should open a file explicitly.

tmpfile() returns NULL if it could not create a temporary file. If it could, it returns a pointer to the **FILE** associated with the temporary file. The function **exit()** removes all files created by **tmpfile()**.

Example

This example implements a primitive file editor that can edit large files. It uses two temporary files to keep all changes. The editor accepts the following commands:

dn	delete; **d52** deletes line 52
in	insert; **i7** inserts line before line 7
pn	print; **p17** prints line 17
p	print the entire file
w	write the edited file and quit
q	quit without writing the file

The entire temporary file is copied with each command.

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <stdarg.h>

FILE *fp, *tmp[2];
int linecount;

fatal(message)
char *message;
{
      fprintf(stderr, "%s\n", message);
      exit(EXIT_FAILURE);
}

/*
 * Copy up to line number or EOF.
 * Return number of lines copied.
 */
static int
copy(line, *ifp, ofp)
int line; FILE *ifp, *ofp;
{
      int i, c, count;

      count = 0;
      for(c=i=1; (i<line || line==-1) && c!=EOF; i++) {
            while((c = fgetc(ifp)) != EOF && c != '\n')
                  fputc(c, ofp);

            if(c == '\n') {
                  count++;
                  fputc('\n', ofp);
            }
      }
      return(count);
}

/*
 * Read a file until line number is read.
 * Return 1 if line is found before EOF.
 */
static int
find(line, ifp)
int line; FILE *ifp;
{
      int i, c;

      for(c=i=1; i<line && c!=EOF; i++)
            while((c = fgetc(ifp)) != EOF && c != '\n')
                  ;
      return(c != EOF);
}

main(int argc, char *argv[])
{
      int i, line, args;
      char c, cmdbuf[80];

      if(argc != 2)
            fatal("usage: tmpfile filename\n");

      if((tmp[0]=tmpfile())==NULL||(tmp[1]=tmpfile())==NULL)
            fatal("Error opening tmpfile\n");

      if((fp = fopen(argv[1], "r")) == NULL)
            fatal("Error opening input file\n");
```

```
linecount = copy(-1, fp, tmp[i = 0]);
fclose(fp);

/* one file pass per command */
for(;;) {
      if(gets(cmdbuf) == NULL)
            fatal("EOF on stdin\n");

      if(!(args = sscanf(cmdbuf, "%c%d", &c, &line)))
            continue;
      fseek(tmp[i], 0L, SEEK_SET);

      switch(c) {
      /* Write edited file */
      case 'w':
            if((fp = fopen(argv[1], "w")) ==  NULL)
                  fatal("Error opening file\n");
            copy(linecount + 1, tmp[i], fp);
            fclose(fp);

      /* Quit */
      case 'q':
            exit(EXIT_SUCCESS);

      /* Print entire file */
      case 'p':
            if(args == 1) {
                  copy(linecount + 1, tmp[i], stdout);
                  continue;
            }
            if(find(line, tmp[i]))
                  copy(2, tmp[i], stdout);
            continue;

      /* Delete a line */
      case 'd':
            if(args == 1)
                  printf("dn where n is a number\n");
            else if(line > linecount)
                  printf("only %d lines\n", linecount);

            else {
                  copy(line, tmp[i], tmp[i^1]);
                  if(find(2, tmp[i]))
                        copy(-1, tmp[i], tmp[i^1]);

                  linecount--;
                  fseek(tmp[i], 0L, SEEK_SET);
                  i ^= 1;
            }
            continue;

      /* Insert a line */
      case 'i':
            if(1 == args)
                  printf("in where n is a number\n");
            else if(line > linecount)
                  printf("only %d lines\n", linecount);

            else {
                  copy(line, tmp[i], tmp[i^1]);
                  printf("Enter inserted line\n");
                  copy(2, stdin, tmp[i^1]);
                  copy(-1, tmp[i], tmp[i^1]);
                  linecount++;

                  fseek(tmp[i], 0L, SEEK_SET);
                  i ^= 1;
            }
            continue;
```

```
            default:
                  printf("Invalid request\n");
                  continue;
            }
      }
}
```

See Also

mktemp(), libc, tempnam(), tmpnam()
ANSI Standard, §7.9.4.3
POSIX Standard, §8.1

Notes

If a program exits abnormally or aborts, the files created by **tmpfile()** may not be removed.

tmpnam() — STDIO Function (libc)

Generate a unique name for a temporary file
#include <stdio.h>
char ***tmpnam(***name***);**
char **name***;*

tmpnam() constructs a unique name for a file. The names returned by **tmpnam()** generally are mechanical concatenations of letters, and therefore are mostly used to name temporary files, which are never seen by the user. A file named by **tmpnam()** does not automatically disappear when the program exits. You must explicitly remove it before the program ends if you want it to disappear.

name points to the buffer into which **tmpnam()** writes the name it generates. If *name* is set to NULL, **tmpnam()** writes the name into an internal buffer that may be overwritten each time you call this function.

tmpnam() returns a pointer to the temporary name. Unlike the related function **tempnam()**, **tmpnam()** assumes that the temporary file will be written into directory **/tmp** and builds the name accordingly.

Example

For an example of this function, see **execve()**.

See Also

libc, mktemp(), tempnam()
ANSI Standard, §7.9.4.4
POSIX Standard, §8.1

Notes

If you want the file name to be written into *buffer*, you should allocate at least **L_tmpnam** bytes of memory for it; **L_tmpnam** is defined in the header **stdio.h**. Under COHERENT, it is 64 characters long.

toascii() — ctype Function (libc)

Convert characters to ASCII
#include <ctype.h>
int toascii(*c***)** int *c***;**

The function **toascii()** takes the integer value *c*, keeps the low seven bits unchanged, and changes the others to zero. This, in effect, transforms the integer value to an ASCII character. **toascii()** then returns the transformed integer. If *c* is already a valid ASCII character, **toascii()** returns it unchanged.

Example

This example prompts for a file name. It then opens the file and prints its contents, while converting all non-alphanumeric characters to alphanumeric.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
```

```
main()
{
      FILE *fp;
      int ch;
      int filename[20];

      printf("Enter file name: ");
      fflush(stdout);
      gets(filename);

      if ((fp = fopen(filename, "r")) != NULL) {
            while ((ch = fgetc(fp)) != EOF)
                  putchar(isascii(ch) ? ch : toascii(ch));
      } else {
            printf("Cannot open %s\n", filename);
            exit(EXIT_FAILURE);
      }
      return(EXIT_SUCCESS);
}
```

See Also

isascii(), libc

Notes

This function is not described in the ANSI Standard. Any program that uses it does not conform strictly to the Standard, and may not be portable to other compilers or environments.

tolower() — ctype Function (libc)

Convert characters to lower case
#include <ctype.h>
int tolower(*c***) int** *c***;**

The function **tolower()** converts the character *c* to lower case. It returns *c* converted to lower case. If *c* is not upper-case character, that is, a character for which **isupper()** returns true, **toupper()** returns it unchanged.

Example

The following example demonstrates **tolower()** and **toupper()**. It reverses the case of every character in a text file.

```
#include <ctype.h>
#include <stdio.h>

main()
{
      FILE *fp;
      int ch;
      int filename[100];

      printf("Enter name of file to use: ");
      fflush(stdout);
      gets(filename);

      if ((fp = fopen(filename,"r")) != NULL) {
            while ((ch = fgetc(fp)) != EOF) {
                  if (islower(ch))
                        putchar(toupper(ch));
                  else if (isupper(ch))
                        putchar(tolower(ch));
                  else
                        putchar(ch);
            }
      } else
             printf("Cannot open %s.\n", filename);
}
```

See Also

_tolower(), libc, toupper()
ANSI Standard, §7.3.2.1
POSIX Standard, §8.1

touch — Command

Update modification time of a file
touch [-c] *file ...*

COHERENT keeps track of when each file was last modified. **touch** changes the modification time of each *file* to the current time, but does not modify its contents. By default, **touch** creates *file* if it does not already exist; the **-c** flag suppresses this.

See Also

commands, make

toupper() — ctype Function (libc)

Convert characters to upper case
#include <ctype.h>
int toupper(*c***) int** *c***;**

toupper() converts the letter *c* to upper case and returns the converted character. If *c* is not an lower-case character, that is, any character for which **islower()** returns true, **toupper()** returns it unchanged.

Example

For an example of this routine, see the entry for **tolower()**.

See Also

_toupper(), libc, tolower()
ANSI Standard, §7.3.2.2
POSIX Standard, §8.1

tparm() — terminfo Function

Output a parameterized string
#include <curses.h>
tparm(*string***,***p1...p9***)**
char **string***,* *parm1 ... par9***;**

COHERENT comes with a set of functions that let you use **terminfo** descriptions to manipulate a terminal. **tparm()** outputs a parameterized string.

A *parameterized string* is a string into which parameters can be inserted, as in a **printf()** formatting string. Under **terminfo**, a parameterized string can hold up to nine parameters. **tparm()** expands the parameters, inserts them into the appropriate "slots" within the string, and then outputs the string.

See the Lexicon entry on **terminfo** for more information on parameterized strings.

See Also

curses.h, terminfo, tputs()

tputs() — termcap/terminfo Function (libterm/libcurses)

Read/decode leading padding information
#include <curses.h>
#include <term.h>
tputs(*name, affcnt, outc***)**
char **name***;* **int** *affcnt***; int (******outc***)();**

tputs() is one of a set of functions that read a **termcap** or **terminfo** terminal description.

tputs() decodes the leading padding information of the string *name*. When you use **tputs()** to interpret the **terminfo** data base, *name* should point to a string that names one of **terminfo**'s variables, as defined in the Lexicon entry for **terminfo**; e.g., **auto_right_margin** or **auto_left_margin**. When you use **tputs()** to interpret the **termcap** data base, *name* should point to **termcap**'s variable code, e.g., **am**.

affcnt gives the number of lines affected by the operation. Set it to one if it is not applicable.

outc points to the routine that writes each character.

Files

/etc/termcap — Terminal capabilities data base
/etc/terminfo — Terminal capabilities data base
/usr/lib/libcurses.a — Routines for reading **terminfo** descriptions
/usr/lib/libterm.a — Routines for reading **termcap** descriptions

See Also

libcurses, libterm, termcap, terminfo

Notes

As noted above, **tputs()** can read either a **termcap** or a **terminfo** description. The **termcap** version of **tputs()** lives in library **/usr/lib/libterm.a**. To obtain the **termcap** version of **tputs()**, link in the library **/usr/lib/libterm.a**. To obtain the **terminfo** version, however, link in the library **/usr/lib/libcurses.a**.

tr — Command

Translate characters
tr [-cds] *string1* **[***string2***]**

tr reads characters from the standard input, possibly translates each to another value or deletes it, and writes to standard output.

Each specified *string* may contain literal characters of the form *a* or *\b* (where *b* is non-numeric), octal representations of the form *\ooo* (where *o* is an octal digit), and character ranges of the form *X-Y*. **tr** rewrites each *string* with the appropriate conversions and range expansions.

If an input character is in *string1*, **tr** outputs the corresponding character of *string2*. If *string2* is shorter than *string1*, the result is the last character in *string2*.

The following flags control how **tr** translates characters:

-c Replace *string1* by the set of characters not in *string1*.

-d Delete characters in *string1* rather than translating them.

-s The "squeeze" option: map a sequence of the same character from *string1* to one output character.

Example

The following example prints all sequences of four or more spaces or printing characters from **infile**:

```
tr -cs ' -~' '\12' <infile | grep ....
```

Here *string1* is the range from **<space>** to '~', which includes all printing characters. Because this example uses the flags **-cs**, **tr** maps sequences of nonprinting characters to newline (octal 12).

See Also

ASCII, commands, ctype.h, sed

Notes

Beginning with COHERENT 4.2, the command

```
echo "This is a test." | tr
```

returns

```
This is a test.
```

This behavior does not conform with POSIX Standard, but is required by a number of third-party packages.

tr — Device Driver

Driver to read stored error messages
/dev/trace

The device driver **tr** is the "traceback" driver for the COHERENT kernel. It manipulates an internal buffer that holds error messages from the kernel or another device driver. It has major number 6. This driver is extremely useful to persons who are writing device drivers.

The DDI/DKI kernel routine **cmn_err()** can be invoked by drivers to write formatted messages. By default, it writes

the messages both onto the system's console and into an internal buffer in memory that can hold up to four megabytes of text. Messages that begin with a caret, '^', go to the console but not the internal buffer. Messages that begin with an exclamation point, '!', go to the buffer but not to the console.

The trace driver **tr** reads this internal buffer, and lets you copy its contents into a file for later perusal. This offers two major advantages to persons developing and debugging device drivers:

- First, copious diagnostic output will no longer scroll off the screen and be lost. The trace buffer holds every error message written through **cmn_err()** until you read the buffer. When you install **tr**, you can set the size of the buffer, up to four megabytes.
- Second, if messages are written to the trace buffer only and not to the console, system timing is affected much less than if the messages were written to the console. This makes it easier to catch subtle problems in timing.

To add **tr** to your kernel, do the following:

- Log in as the superuser **root**.
- **cd** to directory **/etc/conf**.
- Execute script **tr/mkdev**. This will walk you through the process of configuring your kernel to use this driver, and create the device **/dev/trace**.
- Execute script **bin/idmkcoh**, to generate a new kernel.
- Invoke the script **/etc/shutdown** to shutdown system, then boot the new kernel.

To read the contents of the trace buffer, simply use the command

```
cp /dev/trace file
```

where *file* is the file into which you wish to copy the contents of the trace buffer.

See Also

device drivers
COHERENT Device Driver Kit: **cmn_err()**

transports — System Administration

Describe mail transportation systems
/usr/lib/mail/transports

The program **smail** reads file **/usr/lib/mail/transports** for information on the commands it can use to deliver mail, either to your local system or to a remote system.

Each entry within **transports** names a transport and sets its attributes. Each entry consists of the following information:

- The name of the transport. This attribute begins the definition of a transport. The name must be unique, it must appear flush with the left margin, and must be followed by a single colon ':'.
- The name of the *driver*, or program that implements the transport. This can be a command that is part of **smail**'s suite of utilities (which are contained in directory **/usr/lib/mail**), or can be an ordinary COHERENT command. If the latter, then the full name of the command that implements the driver is given with a **cmd** attribute. This is demonstrated below.
- A set of generic attributes for the transport. These attributes are "generic" because they can come from a set that can be applied to any router.
- A set of driver-specific attributes. These can be applied only to entries that use this driver.

To extend an entry across multiple lines, begin successive lines with white space.

Attributes of a Transport

The following gives the generic attributes that a transport can have. Each attribute is followed by its type (Boolean, string, or number). To set a string or number attribute, its name should be followed by an '=', then the value to which you are setting it. To set a Boolean attribute, prefix it with a '+'; to unset a Boolean attribute, prefix it with a '-'.

bsmtp (Boolean)
This transport uses a batched SMTP format, in which the message is enclosed within an envelope of SMTP commands. You can use such a transport to send mail in SMTP format to remote hosts, even when direct two-way connections are not feasible. For example, this will work over UUCP and eliminates difficulties with sending arbitrary addresses as arguments to the command **uux**. Use of this attribute also turns on the attribute **dots**. When this attribute is also used with the attribute **uucp**, **smail** uses UUCP-style bang-path addresses in the SMTP envelope.

crlf (Boolean)
If set, each line of the header and message ends within the pair of characters CR:LF rather than a single newline character. In general, this is not a useful attribute, as the SMTP transport (which requires this as a part of the interactive protocol) always does this anyway.

debug (Boolean)
If set, this attribute replaces the body of the message with debugging information. You can use it, for example, as a shadow transport, to watch the flow of mail for debugging purposes. This lets you debug mail while avoid the problems that arise from saving other users' personal correspondence.

dots (Boolean)
If set, then **smail** uses the "hidden-dot" protocol. With this protocol, **smail** prefixes a period '.' onto every line that already begins with a period. All of the various SMTP modes imply this behavior.

driver (string)
This attribute names the specific entity that actually transports the mail. It is required.

error_transport (string)
This attribute names another transport that **smail** can use to send the message, should this transport fail.

from (Boolean)
If set, **smail** supplies a "From<space>" line before the message when it delivers mail via this transport. If this is a remote transport (i.e., the attribute **local** is not turned on), this line ends with the string

remote from *hostname*

where *hostname* is the UUCP name for your local host (as set in file **/etc/uucpname**). This is useful for delivery via UUCP and for delivering mail to standard mailbox files, which require this format.

hbsmtp (Boolean)
"Half-baked" batched SMTP. This is batched SMTP mode without an initial **HELO** command or an ending **QUIT** command. **smail** can use this transport to create files that it will later concatenate into a batch of SMTP commands and multiple messages. Use of this attribute also turns on attribute **dots**.

local_xform (Boolean)
If this attribute is set, **smail** uses the form of the header and envelope information appropriate for delivery to your local host. This changes no existing header field, except that it inserts commas into the fields that name the sender and recipient. This also affects the form of any generated **From** line and the form of envelope addresses used in SMTP commands.

You can also use this attribute when delivering mail to a remote site that is also running **smail** version 3.1. This is useful within a domain that maintains consistent user-forwarding information. This leaves a message in unqualified format until it leaves the domain via a gateway.

local (Boolean)
This implies that attribute **local_xform** is set, but implies that delivery really is the final delivery to a user, file, or program on your local host. This attribute disables generation of a bounce message that results should a message exceed its allowed hop-count.

max_addrs (number)
This attribute sets the maximum number of recipient addresses that can be given in one call to the transport. If this is turned off, then there is no maximum. The default number is one; typically, this attribute either is left at one or turned off.

max_chars (number)
This states the maximum number of characters in the addresses that can be given in one call to this transport. If this is turned off, there is no maximum number. The default number is about one third of the number of characters that can be passed as arguments to a program. When using SMTP transports, this should be turned off unless a remote host is known to be unable to handle a large number of

addresses. For delivery over UUCP to **rmail** ona remote system, this should be in the neighborhood of 200 to 250, to avoid buffer overruns at the remote site. UUCP generally has small buffers to hold argument information.

If **smail** is given an address whose length exceeds this number, then the address will be passed with one call to the transport. Thus, this limit is not strictly enforced.

max_hosts (number)
This states the maximum number of different hosts that can be given in one call to the transport. If this is turned off using the form **-max_hosts**, there is no maximum number. The default number is one and typically this is not changed.

received (Boolean)
If this attribute is set, **smail** inserts a **Received:** field into each message it delivers via this transport. The form of this field is taken from the attribute **received_field** in file **/usr/lib/mail/config**. This attribute is on by default.

return_path (Boolean)
If this attribute is set, **smail** inserts field **Return-Path:** into the header of each message it delivers via this transport. The form of this field is taken from the attribute **return_path_field** in file **/usr/lib/mail/config**. Use this attribute only with a transport that performs final delivery to a local destination.

shadow (string)
This names a second transport through which **smail** also sends the message. This second transport usually performs some task that is unrelated to the actual delivery of the message. For example, you could use a shadow transport to start a program that looked up the sender within a data base and displayed her picture in a window on your workstation. **smail** calls the shadow transport only if the primary transport successfully delivers the message.

strict (Boolean)
If this flag is set, then **smail** attempts to transform mail that does not conform to RFC822 standards. This may be useful for sites that gateway between the UUCP zone and the Internet. In general, it is not a good idea to turn on this attribute, as it changes the contents of headers fields. Turn on this attribute only when you know that some remote hosts understand only mail that conforms to the RFC822 standard.

unix_from_hack (Boolean)
If set, then **smail** inserts the character '>' before any line in the message that begins with the string "From". This is required for local delivery to mailbox files that are in the standard form expected by the System-V program **mailx** and the BSD program **Mail**.

uucp (Boolean)
If set, then **smail** converts outgoing recipient addresses into UUCP-style paths of the form *hosta***!***hostb***!***hostc***!***user*. An exception is that **smail** preserves any use of '%' as an address operator. Thus, **smail** would convert an envelope address of the form **user%hostb@hosta** to **hosta!user%hostb**. This only affects envelope addresses and does *not* affect the body of the message or its header.

inet (Boolean)
If you set this attribute, **smail** converts output-recipient addresses to Internet specifications. This is not the same as the attribute **strict**, because the transformations apply only to the envelope's address, and not to header's. If **inet** is defined, then when **smail** routes a message to a remote system, it generates a "route-addr" address rather than "bang-path" address. Thus, if **smail** is given the address **user%host@gateway** and **gateway** is reached through the path **hosta!hostb!hostc**, then **smail** generates the address **@hostb,@hostc:user%host@gateway**to be sent to the host **@hosta**.

retry_dir (string)
This attribute tells **smail** to use the subdirectory under directory **/usr/lib/mail/retry** for managing host retry intervals for this transport. By default, the directory is named after the transport. However, multiple transports can share a retry directory by using **retry_dir** to force each to use that directory. For example, by default the definition of each TCP/IP SMTP transport uses **retry_dir** to force that transport to use retry directory **smtp**.

remove_header (string)
Tell **smail** to remove the named header field from each message it sends via this transport. This is an expansion string, so header removal can be made dependent upon some condition. If expansion of the string results in an empty string, then no header is removed. You can specify any number of

remove_header attributes for a given transport.

insert_header (string)
append_header (string)
Add the given header field at the beginning (**insert_header**) or end (**append_header**) of the message header for transport. These are expansion strings, so the header (and the existence of the header) can be made to depend on some conditions. If expansion of the string results in an empty string, then **smail** does not add a header. You can specify any number of **insert_header** and **append_header** attributes for a given transport.

The Default Transports

The following describes the transports that are defined in the version of **/usr/lib/mail/transports** that is shipped with COHERENT.

The first transport, **local**, delivers mail to a user on your system:

```
# local - deliver mail to local users
#
# By default, smail will append directly to user mailbox files.
#
local:      driver=appendfile,      # append message to a file
      return_path,            # include a Return-Path: field
      from,                   # supply a From_ envelope line
      local;                  # use local forms for delivery

      file=/usr/spool/mail/${lc:user}, # location of mailbox files
      mode=0600,                  # For BSD: only the user can
                                  # read and write file
      notify_comsat,              # notify comsat daemon of delivery
      suffix="\1\1\1\10,           # MMDF mailbox format
      prefix="\1\1\1\10,           # MMDF mailbox format
```

The next transport, **pipe**, delivers mail to a shell command:

```
# pipe -deliver mail to shell commands
#
# This is used implicitly when smail encounters addresses which begin with
# a vertical bar character, such as "|/usr/lib/news/recnews talk.bizarre".
# The vertical bar is removed from the address before being given to the
# transport.
pipe: driver=pipe,            # pipe message to another program
      return_path,            # include a Return-Path: field
      from,                   # supply a From_ envelope line
      local;                  # use local forms for delivery

      cmd="/bin/sh -c $user", # send address to the Bourne Shell
      parent_env,             # environment info from parent addr
      pipe_as_user,           # use user-id associated with address
      ignore_status,          # ignore a non-zero exit status
      ignore_write_errors,    # ignore write errors, i.e., broken pipe
      umask=0022,             # umask for child process
      -log_output,            # do not log stdout/stderr
```

The next transport, **file**, delivers mail to a file:

```
# file - deliver mail to files
#
# This is used implicitly when smail encounters addresses which begin with
# a slash or squiggle character, such as "/usr/info/list_messages" or
# perhaps "~/Mail/inbox".
file: driver=appendfile,
      return_path,          # include a Return-Path: field
      from,                 # supply a From_ envelope line
      local;                # use local forms for delivery
```

```
        file=$user,             # file is taken from address
        append_as_user,         # use user-id associated with address
        expand_user,            # expand ~ and $ within address
        mode=0644,              # you may wish to change this
                                # mode, depending upon local
                                # conventions and preferences
        suffix="\1\1\1\10,        # MMDF mailbox format
        prefix="\1\1\1\10,        # MMDF mailbox format
```

The next transport , **uux**, invokes the UUCP command **uux** to deliver messages to a remote site via UUCP:

```
# uux - deliver to the rmail program on a remote UUCP site
#
# HDB UUCP users should comment out the first cmd= line below, and
# uncomment the second.
uux:  driver=pipe,
      uucp,               # use UUCP-style addressing forms
      from,               # supply a From_ envelope line
      max_addrs=5,        # at most 5 addresses per invocation
      max_chars=200;      # at most 200 chars of addresses

      # the -r flag prevents immediate delivery, parentheses around the
      # $user variable prevent special interpretation by uux.
      cmd="/usr/bin/uux - -r -a$sender -g$grade $host!rmail $(($user)$)",
      pipe_as_sender,     # have uucp logs contain caller
      log_output,         # save error output for bounce messages
```

Transport **demand** delivers mail to command **rmail** on a remote system:

```
# demand - deliver to a remote rmail program, polling immediately
#
# HDB UUCP users should comment out the first cmd= line below, and
# uncomment the second.
demand:     driver=pipe,
      uucp,                   # use UUCP-style addressing forms
      from,                   # supply a From_ envelope line
      max_addrs=5,            # at most 5 addresses per invocation
      max_chars=200;          # at most 200 chars of addresses

      cmd="/usr/bin/uux - -a$sender -g$grade $host!rmail $(($user)$)",
      pipe_as_sender,         # have uucp logs contain caller
      log_output,             # save error output for bounce messages
```

The final two transports are local versions of previously defined tranports. What a local transport is, and the advantages it offers, is described above.

Transport **local_uux** is a local version of transport **uux**:

```
local_uux:
      driver=pipe,
      local_xform,        # transfer using local message format
      uucp,               # use uucp-conformant addresses
      from,               # supply a From_ envelope line
      max_addrs=5,        # at most 5 addresses per invocation
      max_chars=200;      # at most 200 chars of addresses

      # the -r flag prevents immediate delivery, parentheses around the
      # $user variable prevent special interpretation by uux.
      cmd="/usr/bin/uux - -r -a$sender -g$grade $host!rmail $(($user)$)",
      pipe_as_sender,     # have uucp logs contain caller
      log_output,         # save error output for bounce messages
```

Finally, **local_demand** is a local form of transport **demand**:

```
local_demand:
      driver=pipe,
      local_xform,         # transfer using local formats
      uucp,                # use uucp-conformant addresses
      from,                # supply a From_ envelope line
      max_addrs=5,         # at most 5 addresses per invocation
      max_chars=200;       # at most 200 chars of addresses

      cmd="/usr/bin/uux - -a$sender -g$grade $host!rmail $(($user)$)",
      pipe_as_sender,      # have uucp logs contain caller
      log_output,          # save error output for bounce messages
```

See Also

Administering COHERENT, config [smail], directors, mail [overview], smail, routers

Notes

For information on how the configuration files **directors**, **routers**, and **transports** relate to each other, see the Lexicon entry for **directors**.

trap — Command

Execute command on receipt of signal
trap [*command***] [***n ...***]**

The command **trap** tells the shell to execute *command* when it receives signal *n*.

You can name more than one signal on the command line for **trap**. Each signal *n* is an integer, as defined in the header file **signal.h**. For information on the traps that COHERENT recognizes and what each one means, see the Lexicon entry for the system call **signal()**. If *n* is zero, the shell executes *command* when it exits.

If you name no *command* on the command line for **trap**, then **trap** resets the trap for signal *n* to its original value. If *command* is a null string (i.e., the string ""), the shell traps signal *n* but does nothing; in effect, this turns off signal *n*.

If you invoke **trap** with no arguments, it prints the signal number and associated command for each signal for which a trap has been set.

The shell executes **trap** directly.

Example

The following example takes two files and outputs only those lines which are the same.

```
# If input only one file-name then simply "cat".
if [ $# = 1 ]; then
      cat $1
      exit 0

# If input two file-names - Ok, else "Usage".
else
      if [ $# != 2 ]; then
            echo "Usage: cmn file1 [file2]"
            exit 1
      fi
fi

# TMP is original name of temporary file (/tmp/temp_(pid)
TMP=/tmp/temp_$$

# Temporary file has to be removed
trap 'rm $TMP; exit 1' 1 2 9
```

```
# Difference between "file1" and "difference between file1 and file2"
# is the common strings "file1" and "file2"
# The strings that are in "file1" and absent in "file2" print in TMP.
diff $1 $2 | sed -n -e "s/^< //p" > $TMP

# The strings that are in "file1" and absent in TMP print in stdout.
diff $1 $TMP | sed -n -e "s/^< //p"

# Remove temporary file
rm $TMP
```

See Also

commands, ksh, sh, signal()

trigraph — C Language

A *trigraph* is a set of three characters that represents one character in the C character set. The set of trigraph sequences was defined in the ANSI Standard to allow users to use the full range of C characters, even if their keyboards do not implement the full C character set. Trigraph sequences are also useful with input devices that reserve one or more members of the C character set for internal use; e.g., the Hazeltine family of terminals, which reserves the tilde '~' as its escape character.

Each trigraph sequence is introduced by two question marks. The third character in the sequence indicates which character is being represented. The following table gives the set of trigraph sequences:

Trigraph Sequence	*Character Represented*
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	|
??>	}
??-	~

The characters represented are the ones used in the C character set but not included in the ISO 646 character set. ISO 646 describes an invariant sub-set of the ASCII character set.

Trigraph sequences are interpreted even if they occur within a string literal or a character constant. Thus, strings that uses a literal "??" will not work the same as under a non-ANSI implementation of C. For example, the function call

```
printf("Feel lucky, punk??!\n");
```

would print:

```
Feel lucky, punk|
```

To print a pair of questions marks, use the escape sequence '\??'. For example:

```
printf("Feel lucky, punk\??!\n");
```

See Also

cc, C language
ANSI Standard, §5.2.1.1

Notes

By default, the COHERENT C compiler **cc** ignores trigraphs. To invoke interpretation of trigraphs, use the option **-V3GRAPH**.

troff — Command

Extended text-formatting language
troff [*option ...***] [***file ...***]**

The command **troff** is the COHERENT typesetter and text-formatting language. It performs typeset-quality text formatting, suitable for printing on either the Hewlett-Packard LaserJet II or III printers, or on any printer for

which the PostScript language has been implemented.

troff Input

troff processes each given *file*, or the standard input if none is specified, and prints the formatted result on the standard output. The input must consist of text with formatting commands embedded within it.

troff provides a full suite of commands that set line length, page length and page offset, generate vertical and horizontal motions, indentation, fill and adjust output lines, and center text. The great flexibility of **troff** lies in its acceptance of user-defined macros to control almost all higher-level formatting. For example, the formation of paragraphs, header and footer areas, and footnotes must all be implemented by the user via macros.

troff uses a superset of the commands and syntax used by **nroff**, the other COHERENT text-formatter: files prepared for the latter usually can be processed through the former without requiring any changes. **troff** differs from **nroff** in that **nroff** can perform only monospaced formatting, whereas **troff** can handle multiple fonts of type, both monospaced and proportionally spaced. It lets you load font-width tables dynamically, so you can use whatever fonts you have loaded into your printer at a given time. **troff** also lets you move about the page in increments other than sixths of an inch vertically or tenths of an inch horizontally.

troff produces output either in the Hewlett-Packard Printer Control Language (PCL) or PostScript, whichever you prefer. The former can be printed on the Hewlett-Packard LaserJet family of laser printer, and can use any PCL bitmapped "soft font". The latter can be printed on any printer that supports the PostScript language, and can use any font for which you have an Adobe Font Metric description. The default is PCL output; to obtain PostScript, use the **-p** command-line option. See below for information on how to manage downloadable fonts.

Command-line Options

Command-line options may be listed in any order. They are as follows:

-d Debug: print each request before execution. This option is very useful when you are writing and debugging new macros.

-D Display the available fonts. These are all the fonts that have been loaded into **troff** with the **.lf** primitive (described below).

-f *name* Write the temporary file into file *name*.

-i *files* Read from the standard input after reading the given *files*.

-k Keep: do not erase the temporary file.

-l Landscape mode: output is rotated 90 degrees, with default size 11 by 8.5 inches rather than 8.5 by 11 inches.

-m*name* Include the macro file **/usr/lib/tmac.***name* in the input stream.

-n*N* Number the first page of output *N*.

-p Produce output for a PostScript printer rather than for a HP-compatible printer.

-r*aN* Set number register *a* to the value *N*.

-r*abN* Set number register *ab* to value *N*. For obvious reasons, *ab* cannot contain a digit.

-v Return the number of your version.

-x Do not eject to the bottom of the last page when text ends. This option lets you use **troff** interactively, which is especially useful when debugging macros.

If the environmental variable **TROFF** is set when **troff** is invoked, its contents are prefixed to the list of command-line arguments. This allows the user to set commonly used options once in the environment rather than on each **troff** command line.

troff Primitives

As noted earlier, **troff**'s command set is a superset of that used by **nroff**: see the Lexicon entry on **nroff** for information on the commands and escape sequences shared by **troff** and **nroff**. This article describes the primitives that **troff** does *not* share with **nroff**.

Please note that the basic **troff** unit is one-tenth of a point. A printer's point is 1/12 of a pica, which is in turn one-sixth of an inch; therefore, there are 72 points and 720 **troff** units in an inch.

.co *endmark*
Copy input to output file directly, with no processing. If *endmark* argument is present, **troff** copies input until it finds a line containing *endmark* followed by **\n**. If no *endmark* is given, **troff** copies input until it finds a line containing **.co\n**. This directive is useful for embedding PostScript commands in an input file.

.cs *XX N M*
Set font *XX* to use constant character spacing. The width of each character is *N* divided by 36 ems. If *M* is present, it specifies the width of an em; otherwise, *N* assumes the point size em for the given font.

.fd Display the currently available fonts.

.fp *N XX*
Associate font name *XX* with numeric font position *N*. The given *N* should be a number between 1 and 9. Subsequently, the numeric font position can be used in an escape sequence **\f***N* to select the font. (This nomenclature comes from the days when phototypesetters used print wheels that were set in fixed positions on the device.) The **nroff** primitive **.rf** performs a similar task, and is more flexible in its syntax.

.fz *XX N*
Fix the point size of font *XX* at *N*. The point size of the font will not be affected by subsequent **.ps** commands or **\s***N* point size escapes.

.lf *XX file* **[***n***]**
Load font-width table from *file* and use it for font *XX*. If *file* is not found, **troff** looks for **/usr/lib/roff/troff_pcl/fwt/***file* or **/usr/lib/roff/troff_ps/fwt/***file* (depending on whether the **-p** option is used).

The optional third argument sets the default point size of the loaded font to *n*. Note that this argument takes effect only if **troff** is running in **-p** (PostScript) mode.

For example, to load the font-width table for the PCL bitmapped font **cn090rpn.usp** (which sets Century Roman, nine point, portrait mode) and name it font **RS**, use the command:

```
.lf RS cn090rpn.usp
```

To do the same thing under PostScript, use the command:

```
.lf RS Century_R.fwt 9
```

Thereafter, you can reference font **RS** with either **.ft RS** or **\f(RS**.

Note that the second argument to this primitive must name a font-width table generated by the COHERENT command **fwtable**, not the font itself, although both may have the same name.

Please note that **.lf** is unique to the COHERENT implementation of **troff**, and cannot be ported to other implementations.

.ps *N***p** Set point size to *N* points. The default point size is 10 point.

.rb *file* Read input from *file* and copy it to the output without processing. This directive is useful for including files containing PostScript routines in the output.

.ss *N* Set the minimum word spacing to *N* divided by 36 ems.

.vs *N***p** Set the vertical spacing to *N* points. The default vertical spacing for **troff** is 11 points.

Escape Sequences

troff recognizes the following escape sequences, in addition to those recognized by **nroff**:

\| Set a 1/6th-em half-narrow space character.

\^ Set a 1/12th-em half-narrow space character.

\s´*N*˝ Set the point-size escape sequence to *N*. Like the **.ps** primitive, it changes the point size to *N*. The specified *N* may have a leading plus or minus sign to make the new size relative to the current point size.

\X*dd* Output character *dd* where *dd* are two hexadecimal digits. This is useful for forcing **troff** to print characters outside the normal printable range, e.g., those with the high-order bit set. **troff** reserves the following values for its internal use:

LEXICON

<ctrl-space>	0X00	Ignored
<ctrl-A>	0X01	Leader dots, same as "\a"
<ctrl-I>	0X09	Tab, same as "\t"
<ctrl-J>	0X10	Newline

The hexadecimal values to which characters map depend upon the character set that you (or your printer) use. For example, to print the character 'ß' using the Hewlett-Packard Laser-Jet printer and the Pacific Page cartridge, use the escape sequence **\XFB**.

The escape sequence **\X** is unique to the COHERENT implementation of **nroff** and **troff**. Code that uses it will behave differently when ported to other implementations.

Number Registers

The basic unit of measure under **troff** is the decipoint, or one-tenth of a printer's point. A point is one-tenth of a pica, which in turn is one sixth of an inch; therefore, there are 72 points in an inch, or 720 decipoints. All **troff** number registers that hold information about page or type dimensions hold that information in decipoints. For this reason, the decipoint is sometimes called the "machine unit."

The following table shows how other units of measure translate into **troff** machine units:

inch:	1**i** = 720**u**
vertical line space:	1**v** = 110**u**
centimeter:	1**c** = 283**u**
em:	1**m** = 100**u**
en:	1**n** = 50**u**
pica:	1**P** = 120**u**
point:	1**p** = 10**u**

If you are working with PostScript, you must remember to divide the value of a **troff** number register by ten before you pass the value to PostScript, or you will see very strange results on your page — or likelier, no results at all.

Special Characters

troff includes a set of escape sequences for setting special characters. These escape sequences are defined in the files **/usr/lib/roff/troff_*/specials.r**. If you have additional fonts or an extended PostScript cartridge on your printer, you can modify these files to change the current definitions or add new ones.

The following shows the escape sequences currently defined in **specials.r**, and the character each prints:

\(em	—	\(hy	-	\(bu	•	\(sq	[]
\(ru	_	\(14	1/4	\(12	1/2	\(34	3/4
\(fi	fi	\(fl	fl	\(ff	ff	\(Fi	ffi
\(Fl	ffl	\(de	°	\(dg	†	\(fm	'
\(ct	¢	\(rg	®	\(co	©	\(tm	™
\(pl	+	\(mi	–	\(eq	=	\(**	*
\(sc	§	\(aa	´	\(ga	`	\(ul	_
\(sl	/	\(*a	α	\(*b	β	\(*g	γ
\(*d	δ	\(*e	ε	\(*z	ζ	\(*y	η
\(*h	θ	\(*i	ι	\(*k	κ	\(*l	λ
\(*m	μ	\(*n	ν	\(*c	ξ	\(*o	ο
\(*p	π	\(*r	ρ	\(*s	σ	\(ts	ς
\(*t	τ	\(*u	υ	\(*f	ϕ	\(*x	χ
\(*q	ψ	\(*w	ω	\(*A	A	\(*B	B
\(*G	Γ	\(*D	Δ	\(*E	E	\(*Z	Z
\(*Y	H	\(*H	Θ	\(*I	I	\(*K	K
\(*L	Λ	\(*M	M	\(*N	N	\(*C	Ξ
\(*O	O	\(*P	Π	\(*R	P	\(*S	Σ
\(*T	T	\(*U	Y	\(*F	Φ	\(*X	X_
\(*Q	Ψ	\(*W	Ω	\(sr	√	\(rn	
\(>=	≥	\(<=	≤	\(==	≡	\(~=	≈
\(ap	~	\(!=	≠	\(->	→	\(<-	←
\(ua	↑	\(da	↓	\(mu	×	\(di	/
\(+-	±	\(cu	∪	\(ca	∩	\(sb	⊂
\(sp	⊃	\(ib	⊆	\(ip	⊇	\(in	∞
\(pd	∂	\(gr	∇	\(no	¬	\(is	∫

\(pt	∝	\(es	∅	\(mo	∈	\(br	│
\(dd	‡	\(rh	☞	\(lh	<-	\(or	|
\(ci	O	\(lt	⎛	\(lb	⎝	\(rt	⎠
\(rb	⎠	\(lk	{	\(rk	}	\(bv	⎪
\(lf	⎣	\(rf	⎦	\(lc	⎡	\(rc	⎤

Printer Configuration

troff reads several files in directory **/usr/lib/roff/troff_pcl** (when generating PCL output) or **/usr/lib/roff/troff_ps** (when generating PostScript) to find printer-specific information. It reads special character definitions from file **specials.r**. It reads font loading requests from file **fonts.r**. It copies file **.pre** at the beginning of the output. It copies file **.post** at the end of the output. In landscape mode, **troff** looks for files **.pre_land** and **.post_land** instead. You can change these files as desired to include printer-specific commands in **troff** output.

Managing Fonts

As noted above, **troff** produces output in either of two page-description languages: the Hewlett-Packard Printer Control Language (PCL), which is the "native language" of Hewlett-Packard's LaserJet printers; or PostScript. The COHERENT system also comes with tools that lets you process fonts, so that you can use with **troff** either downloadable soft fonts or the fonts that are on board your printer.

The following two sections describe how to manage fonts under PCL and under PostScript. You should refer to the section that is appropriate to your type of printer.

PCL Fonts

Before **troff** can use a font, it must know the following information:

- What the width of every character of the font is, and
- How it can tell the printer to print that font.

Both pieces of information are stored in a file called a *font-width table*. Before **troff** can use a font, it must read the font-width table for that font.

To load a font-width table into **troff**, use the primitive **.lf**. Its syntax is as follows:

.lf *XX file*

XX gives the name by which you will call the font in your **troff** program. *file* is the font-width table for this font. If *file* is not a full path name, **troff** looks for it in directory **/usr/lib/roff/troff_pcl/fwt**.

COHERENT comes with font-width tables for a number of commonly used fonts. The following tables are for the fonts built into the Hewlett-Packard LaserJet III:

Table	*Description*
CGTimes_B.fwt	Times Bold, scalable, rotatable
CGTimes_BI.fwt	Times Bold Italic, scalable, rotatable
CGTimes_I.fwt	Times Italic, scalable, rotatable
CGTimes_R.fwt	Times Roman, scalable, rotatable
Cour10_B.fwt	Courier Bold, ten point, portrait
Cour10_I.fwt	Courier Italic, ten point, portrait
Cour10_R.fwt	Courier Roman, ten point, portrait
Cour12L_B.fwt	Courier Bold, 12 point, landscape
Cour12L_R.fwt	Courier Roman, 12 point, landscape
Cour12_B.fwt	Courier Bold, 12 point, portrait
Cour12_I.fwt	Courier Italic, 12 point, portrait
Cour12_R.fwt	Courier Roman, 12 point, portrait
LinepL_R.fwt	Line Printer, 8.5 point, landscape
Linep_R.fwt	Line Printer, 8.5 point, portrait
Univers_B.fwt	Univers Bold, scalable, rotatable
Univers_BI.fwt	Univers Bold Italic, scalable, rotatable
Univers_I.fwt	Univers Italic, scalable, rotatable
Univers_R.fwt	Univers Roman, scalable, rotatable

Note that the scalable Hewlett-Packard fonts are set by default at 250 points in size — that is, about 3.5 inches. Because you cannot scale PCL fonts when you load them, you must use the **.ps** primitive to size the font.

The following **troff** program demonstrates scalable fonts on the Hewlett-Packard LaserJet III:

LEXICON

```
.lf TR CGTimes_R.fwt
.lf TB CGTimes_B.fwt
.lf TI CGTimes_I.fwt
.lf UR Univers_R.fwt
.lf UB Univers_B.fwt
.lf UI Univers_I.fwt
.vs 14p
.ps 12p
\f(TRThis is Times Roman, 12 point.
.sp
\f(TBThis is Times Bold, 12 point.
.sp
\f(TIThis is Times Italic, 12 point.
.vs 26p
.ps 24p
\f(URThis is Univers Roman, 24 point.
.sp
\f(UBThis is Univers Bold, 24 point.
.sp
\f(UIThis is Univers Italic, 24 point.
.br
```

Note that this program does not run correctly if downloaded to a LaserJet II, or to any printer that is running PostScript.

The COHERENT command **fwtable** lets you build new font-width tables. It can build tables for PCL bit-mapped soft fonts, as well as for fonts that are built into the LaserJet III.

To manipulate PCL bit-mapped soft fonts, do the following:

- Use the command **fwtable** to build a font-width table from the font. The input to **fwtable** should be the soft font itself; and the output of **fwtable** should be redirected into an appropriately named file. See the lists of tables given above for an idea of how to name your font-width table.
- Move the newly created font-width table into directory **/usr/lib/roff/troff_pcl/fwt**.
- Move the font itself into directory **/usr/lib/roff/troff_pcl/fonts**. You may need to create this directory if this is the first time you are using soft fonts.
- Include the instruction **.lf** in your **troff** file to load the font-width table and name the font, as shown above. If you use the same fonts repeatedly, you may wish to put the **.lf** primitives into a separate file that you always include on your **troff** command line via the environmental variable **TROFF**.
- Before you print your document, load the soft font into your printer. If you are using the **hp** spooler to spool files to your printer, use the command **hpr -f**. If you are using the MLP spooler, then you must pre-process the font with the command **pclfont**, then spool the processed font to device **hpraw**. Both commands are described in detail in their Lexicon entries. Briefly, to load font **tr100bpn.usp** into your printer, use the command

  ```
  hpr -f /usr/lib/roff/troff_pcl/fonts/tr100bpn.usp
  ```

  or the command:

  ```
  pclfont /usr/lib/roff/troff_pcl/fonts/tr100bpn.usp | lp -d hpraw
  ```

  These commands also let you specify what “slot” to put the font; you can use this to help manage fonts in your printer. By placing the frequently used fonts in the lower slots, you can then load the less-frequently used fonts into the upper slots, and overwrite just those fonts when you change fonts for another printing job. You must do such font management by hand — COHERENT does not include a utility to do it for you.

  You may wish to write the font-loading commands into a script that you execute before you print a job. You must reload fonts every time you power up your printer or clear its memory.

To build a font-width table for a font built into your LaserJet III, do the following:

- Each font on your printer is described with a **.tfm** file, which comes on a disk with your printer. (If you did not receive such a disk, check with the dealer from which you purchased your printer, or write to Hewlett-Packard.) Use the COHERENT command **doscp** to copy the **.tfm** file for the font that interests you from the disk.

- Use the command **fwtable -t** to build the font-width table. Its input should be the **.tfm** file that you just uploaded. Redirect its output into an appropriate named file.
- Move the newly created font-width table into directory **/usr/lib/roff/troff_pcl/fwt**.
- Note that because the font is build into your printer, you do not need to download anything before you can use the font. When **troff** reads the font-width table, it will know how to invoke the font on your printer.

PostScript Fonts

Before **troff** can use a font, it must know the following information:

- What the width of every character of the font is, and
- How it can tell the printer to print that font.

Both pieces of information are stored in a file called a *font-width table*. Before **troff** can use a font, it must read the font-width table for that font.

To load a font into **troff**, use the primitive **.lf**. Its syntax is as follows:

.lf *XX file* **[***n***]**

XX gives the name by which you will call the font in your **troff** program. *file* is the font-width table for this font. If *file* is not a full path name, **troff** looks for it either in directory **/usr/lib/roff/troff_ps/fwt**.

The optional argument **n** lets you size the font. This applies only to PostScript scalable fonts. All fonts that are loaded with this option are *not* affected by the **.ps** primitive.

For example, the instruction

```
.lf HR      HelvNar_R.fwt  12
```

loads a font for PostScript output. The font is named **HR**. The font-width table is read from file **/usr/lib/roff/troff_ps/HelvNar_R.fwt**, which defines the font Helvetica Narrow Roman. Finally, it sizes the font to 12 points. Hereafter, the instructions **.ft HR** or **\f(HR** invoke this font.

COHERENT comes with font-width tables for a number of commonly used fonts. The following tables are for PostScript fonts. LaserJet III, and are kept in directory **/usr/lib/roff/troff_pcl/fwt**. All are, of course, scalable and rotatable:

Table	*Description*
Avant_B.fwt	Avant-Garde Roman (Gothic Book)
Avant_BI.fwt	Avant-Garde Bold Italic
Avant_I.fwt	Avant-Garde Italic
Avant_R.fwt	Avant-Garde Roman
Bookman_B.fwt	Bookman Bold
Bookman_BI.fwt	Bookman Bold Italic
Bookman_I.fwt	Bookman Italic
Bookman_R.fwt	Bookman Roman
Century_B.fwt	Century Bold
Century_BI.fwt	Century Bold Italic
Century_I.fwt	Century Italic
Century_R.fwt	Century Roman
Chancery_I.fwt	Zapf Chancery Italic
Courier_B.fwt	Courier Bold
Courier_BI.fwt	Courier Bold Italic
Courier_I.fwt	Courier Italic
Courier_R.fwt	Courier Roman
Dingbats.fwt	Zapf Dingbats
HelvNar_B.fwt	Helvetica Narrow Bold
HelvNar_BI.fwt	Helvetica Narrow Bold Italic
HelvNar_I.fwt	Helvetica Narrow Italic
HelvNar_R.fwt	Helvetica Narrow Roman
Helv_B.fwt	Helvetica Bold
Helv_BI.fwt	Helvetica Bold Italic
Helv_I.fwt	Helvetica Italic
Helv_R.fwt	Helvetica Narrow

Pala_B.fwt . . . . . Zapf Calligraphic Bold (Palatino)
Pala_BI.fwt . . . . . Zapf Calligraphic Bold Italic
Pala_I.fwt . . . . . Zapf Calligraphic Italic
Pala_R.fwt . . . . . Zapf Calligraphic Roman
Symbol.fwt . . . . . Symbols
Times_B.fwt. . . . . Times Bold
Times_BI.fwt . . . . . Times Bold Italic
Times_I.fwt . . . . . Times Italic
Times_R.fwt. . . . . Times Roman

Note that these tables are designed for the fonts used on the Pacific Page implementation of the PostScript language. They may not work correctly with genuine Adobe fonts.

The following gives an example program to demonstrate the PostScript fonts:

```
.lf HR      HelvNar_R.fwt  12
.lf HC      Avant_B.fwt    24
.lf DB      Dingbats.fwt    9
.vs 14
.sp
\f(HRThis is 12-point Helvetic Narrow Roman
.vs 26
.sp
\f(HCThis is 24-point Avant-Garde
.vs 11
.sp
\fRA row of dingbats:  \f(DBa row of dingbats
```

This program will not work unless you format using the **-p** option to **troff**, and print it on a PostScript printer. Please note that because PostScript is a portable language, you can print the PostScript output of **troff** on any printer that implements PostScript, not just the Hewlett-Packard LaserJet.

COHERENT comes with tools with which you can "cook" fonts so that you can use with with **troff**, whether the fonts are downloadable soft fonts or on board a cartridge. To cook fonts that are on-board a cartridge in your printer, do the following:

- First, the PostScript cartridge should come with a set of files that give font-width information. These have the suffix **.afm**; there should be one file for each font in your cartridge. If you did not receive such a cartridge, contact the dealer from which you purchased the cartridge, or contact the cartridge's manufacturer. Use the command **doscp** to copy the **.afm** files from the disk onto your COHERENT system.

- Use the command **fwtable -p** to cook each **.afm** file into **troff**'s font-width table format. Each font-width table that you create should have the suffix **.fwt**, and should be named so that it appropriate describes the font. See the above table of font-width tables for examples.

  Move the newly created font-width tables into directory **/usr/lib/roff/troff_ps/fwt**.

  Thereafter, when you write a **troff** program, use the **.lf** primitive to load the font-width table. You may wish to create a file called **fonts.r** that routinely loads all of the font-width tables that you use routinely. You do *not* need to load fonts into your printer; the font-width table includes the information needed so that **troff** can invoke them from your cartridge.

COHERENT comes with tools to help you manage download soft fonts under PostScript. Note that the fonts must be in the Adobe Font Metric (AFM) format. To manage downloadable AFM fonts, do the following:

- A downloadable AFM font comes in three files: a file of information about the font, which has the suffix **.inf**; a file that contains the font-width table, which has the suffix **.afm**; and a file that contains the font itself, which has the suffix **.pfb**. You can ignore the **.inf** file; it is not used in this process. You should use the COHERENT command **doscp -b** to copy the **.pfb** from the floppy disk; and use the command **doscp -a** to copy the **.afm** file from floppy disk. (The options **-b** and **-a** stand, respectively, for binary and ASCII modes.)

- Use the command

  `fwtable -p` *fontname*`.afm` *fontname*`.fwt`

  to generate the font-width table from the **.afm** file. Note that the font-width table should have the suffix **.fwt**. By convention, you should give the font-width table the same name as the font, to help you remember which table goes with which font; this, however, is not required. For example, to create the font-width table for the Adobe font Avant Garde bold, use the following command:

```
fwtable -p avgb____.afm avgb____.fwt
```

- Move the newly created font-width table into directory **/usr/lib/roff/troff_ps/fwt**.
- Next, use the command **PSfont** to "cook" the **.pfb** file into a form that can be downloaded to your printer. Note that a font can be cooked into either of two forms. The first form permits the font to stay resident in your printer, so that you can use it to print an indefinite number of documents. The second form does not permit the font to stay resident in your printer, but it does permit you to include the font directly within your **troff** output. The first form is the default output of **PSfont**; to create the second form, invoke **PSfont** with its option **-s**. For example, to cook the font Avant Garde bold into the first output format, use the command:

  ```
  PSfont avgb____.pfb avgb____.ps1
  ```

  To it into the second form, use the command:

  ```
  PSfont -s avgb____.pfb avgb____.ps2
  ```

  Note that the suffix **.ps1** indicates the first (stay-resident) form of the font, whereas the suffix **.ps2** indicates the second (includable) form of the font. These suffixes are simply conventions, and are not required.
- Move the newly created fonts into directory **/usr/lib/roff/troff_ps/ps**. Note that you may need to create this directory when you first begin to process fonts.
- When you create a **troff** program, use the primitive **.lf** to include the font-width table for this font and size the font, as described above.
- If you have processed the fonts into the first (stay-resident) form, you must load them into your printer before you can print any documents. To download the font, use either the command **hpr -B** or the command **lp -d***printer* (where *printer* names the printer to which the font is being downloaded). For example, to download the Avant Garde bold font to printer **hpraw**, use the command:

  ```
  lp -dhpraw /usr/lib/roff/troff_ps/ps/avgb____.ps1
  ```

  (For more information on the command **lp**, see its entry in the Lexicon, or see the entry for **printer**.) You may wish to create a script to download the fonts that you use commonly. Note that you must reload the fonts into your printer every time you either power up the printer or clear out its memory. Note, too, that downloading and processing stay-resident fonts may take several minutes, depending upon your printer's make.
- To use the "includable" form of a font, use the **troff** primitive **.rb** to load it into the **troff**. For example, to include Avant Garde bold directly within your **troff** output, include the following statement in your **troff** source:

  ```
  .rb /usr/lib/roff/troff_ps/ps/avgb____.ps2
  ```

  If you use some downloadable fonts commonly, you may wish to include a set of **.rb** statements for the fonts in file **fonts.r**. Note that files that include downloadable fonts will be *much* larger than those that do not use them.

Files

/tmp/rof* — Temporary files
/usr/lib/tmac.* — Standard macro packages
/usr/lib/roff/troff_pcl/ — Support files directory for PCL
/usr/lib/roff/troff_ps/ — Support files directory for PostScript
/usr/lib/roff/troff_*/.pre — Output prefix
/usr/lib/roff/troff_*/.pre_land — Output prefix, landscape mode
/usr/lib/roff/troff_*/.post — Output suffix
/usr/lib/roff/troff_*/.post_land — Output suffix, landscape mode
/usr/lib/roff/troff_*/fonts.r — Font definitions
/usr/lib/roff/troff_*/fwt/ — Directory for font width tables
/usr/lib/roff/troff_*/specials.r — Special character definitions

See Also

col, commands, deroff, fwtable, hpr, lp, man, ms, nroff, printer, PSfont
nroff, The Text-Formatting Language, tutorial

Adobe Systems Incorporated: *PostScript Language Reference Manual.* Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1988.

LEXICON

Adobe Systems Incorporated: *PostScript Language Tutorial and Cookbook.* Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1988.

Emerson, S.L., Paulsell, K.: *troff Typesetting for Unix Systems.* Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1987 (ISBN 0-13-930959-4).

Lawson, A.: *Printing Types: An Introduction.* Boston: Beacon Press, 1971.

Lawson, A.: *Anatomy of a Typeface.* Boston: David R. Godine, Publisher, 1990.

Diagnostics

For a list of the error messages that **troff** can produce, see the Lexicon entry for **nroff**.

Notes

Like **nroff**, **troff** should be used with the macro packages **ms**, which is found in the file **/usr/lib/tmac.s**, and **man**, which is found in the file **/usr/lib/tmac.an**.

troff output, unlike that of **nroff**, cannot be processed through a terminal driver. If you redirect the output of **troff** to a terminal, all you will see is the literal program it outputs.

Laser printers cannot print on an area near each edge of the output page. Output sent to the unprintable area will disappear. On some printers, the *logical page* does not correspond to the *physical page*, so printed **troff** output may be offset from the specified position on the physical page.

true — Command

Unconditional success
true

true does nothing, successfully. It always returns zero (i.e., true).

true is useful in shell scripts when you want to execute a condition indefinitely. For example, the following example

```
while true; do
      date
done
```

prints the current date and time on your screen forever (or at least until interrupted by typing **<ctrl-C>**).

See Also

commands, false, ksh, sh

Notes

Under the Korn shell, **true** is an alias for the partial-comment **:**.

trustme — System Administration

List of trusted users
/etc/trustme

The file **/etc/trustme** names users who are "trusted" — that is, who are permitted to log into the system even though the file **/etc/nologin** has been created to stop users from logging in.

See Also

Administering COHERENT, login, nologin

tsort — Command

Topological sort
tsort [*file***]**

tsort performs a topological sort of a set of input items. The input *file* (or the standard input, if no *file* is given) specifies an ordering on pairs of items. It consists of pairs of items separated by blanks, tabs or newlines. If a pair contains the same item twice, it simply indicates that the item is in the input set. Otherwise, the pair indicates that the first item precedes the second in the ordering.

tsort prints a sorted list of the input items on the standard output.

See Also

commands, sort

Diagnostics

tsort prints an error message on the standard error if its input contains an odd number of items or if the specified ordering includes a cycle.

ttt — Command

Play 3-D tic-tac-toe
/usr/games/ttt

The COHERENT game **ttt** plays three-dimensional tic-tac-toe. Each playing board is four-by-four, and four are stacked on top of each other. You play against the computer; each player selects to occupy one "square" on one of the boards. The first player to get four four squares in a row, in any direction, wins.

See Also

commands

tty — Command

Print the user's terminal name
tty

tty prints the name of the character-special file that manages your terminal.

Diagnostics

tty prints the message "Not a tty." if the user is not associated with any controlling terminal.

See Also

commands, who

tty.h — Header File

Define flags used with tty processing
#include <sys/tty.h>

tty.h defines manifest constants that are used by the routines that handle ttys.

See Also

header files, tty

ttyname() — General Function (libc)

Identify a terminal
#include <unistd.h>
char ***ttyname(***_fd_**)**
int _fd_**;**

Given a file descriptor *fd* attached to a terminal, **ttyname()** returns the complete pathname of the special file (normally found in the directory **/dev**).

Files

/dev/* — Terminal special files
/etc/ttys — Login terminals

See Also

ioctl(), isatty(), libc, tty(), ttyslot(), unistd.h
POSIX Standard, §4.7.2

Diagnostics

ttyname() returns NULL if it cannot find a special file corresponding to *fd*.

Notes

The string returned by **ttyname()** is kept in a static area, and is overwritten by each subsequent call.

ttys — System Administration

Describe terminal ports
/etc/ttys

File **/etc/ttys** describes the terminals in the COHERENT system. The process **init** reads this file when it brings up the system in multi-user mode.

/etc/ttys contains one line for each terminal. Each line consists of the following four fields:

1. The first field is one character long, and indicates if the device is enabled for logins: '0' indicates that the device is not enabled, and '1' (one) indicates that logins are enabled for the device.

2. The second field is one character long, and indicates whether the device is local (i.e., a terminal) or remote (i.e., a modem): 'r' indicates remote, and 'l' (lower-case **L**) indicates local.

If the port is named in file **/etc/dialups**, then the command **login** checks the file **/etc/d_passwd** to see if the program the user is invoking is protected by a password. If so, it prompts the user for that additional password before allowing her to log in. For details, see the Lexicon entries for **login**, **dialups**, and **d_passwd**.

3. The third field is one character long, and sets the baud rate for the device. Note that a device can have either a fixed baud rate, or a variable baud rate. The following table gives the codes for fixed baud rates:

C	110
G	300
I	1200
L	2400
N	4800
P	9600
Q	19200
S	38400

The common variable-speed codes terminal types are as follows:

0	300, 1200, 150, 110
3	2400, 1200, 300

When a user dials into a variable-speed line, a message is sent to the terminal using the first speed listed. If the message is unintelligible, the user hits the **<break>** key and the system tries the next speed; and so on, until the correct speed is selected.

4. The fourth field names the port that this device is plugged into. The following table names the ports that COHERENT recognizes:

console	The console device
color*N*	Virtual console device *N*, color console
mono*N*	Virtual console device *N*, monochrome console
com*N***l**	Serial port **com***N*, local device
com*N***r**	Serial port **com***N*, remote device
com*N***fl**	Serial port **com***N*, local device, flow control
com*N***fr**	Serial port **com***N*, remote device, flow control
com*N***pl**	Serial port **com***N*, local polled device
com*N***pr**	Serial port **com***N*, remote polled device

Note that if field 2 (described above) says that this is a local device, then you must use a port descriptor that ends in 'l'; likewise, if field 2 states that this is a remote device, the port descriptor must end in 'r'. Doing otherwise will result in trouble. See Lexicon entry **asy** for details. Note also that you must use a device with hardware flow control (i.e., a device whose suffix includes the letter 'f') if you wish to use a high-speed modem (e.g., 14.4*bis*).

Do not leave trailing spaces at the end of an entry in **/etc/ttys**. Leaving blanks at the end of a line usually results in errors that state that a device could not be found.

After you have edited **/etc/ttys**, the following command forces COHERENT to re-read the file and use the new descriptions:

```
kill quit 1
```

Examples

Consider the following **ttys** entry:

```
1lPconsole
```

Field 1 is the first character. Here it is set to '1' (one), which indicates that the device is enabled for logins. Field 2 is the second character. Here it is set to 'l' (lower-case **L**), which indicates that this is a local device. Field 3 is the third character. Here, it is set to 'P', which indicates that the device operates at the fixed baud rate of 9600 baud. This field is ignored by the console device driver since the console is not a serial device. Finally, field 4 is the remainder of the line. Here, it indicates that the device in question is the console.

Now, consider another example:

```
1r3com3r
```

Field 1 is the first character. Here it is set to '1' (one), which indicates that the device is enabled for logins. Field 2 is the second character. Here it is set to 'r', which indicates that this is a remote device, i.e., a modem. Field 3 is the third character. Here, it is set to '3', which indicates that the device operates at variable baud rates of 2400, 1200, and 300. By hitting the **<break>** key on the terminal, the user can select from among those three baud rates, in that order. Finally, field 4 is the remainder of the line. Here, it indicates that the device in question is plugged into port **com3**, and is accessed via special file **/dev/com3r**.

Files

/etc/ttys

See Also

Administering COHERENT, asy, d_passwd, dialups, getty, init, login, stty, terminal, tty

Notes

If you wish to enable logins on a COM port on which you will also be dialing out, you must edit file **/etc/ttys** and add a line for the raw device. For example, if you have a modem plugged into COM1 and you wish to dial out on that port, you must have an entry for both **com1l** and **com1r**. Note that the entry for **com1r** *must* precede the entry for **com1l**. If you do not do this, the commands **cu** and **uucico** cannot disable **com1r** before they dial out on **com1l**.

cu also requires that the device **/dev/console** appear last in file **/etc/ttys**. If this is not so, **cu** refuses to disable the enabled port or dial out.

ttyslot() — General Function (libc)

Return a terminal's line number
int ttyslot()

ttyslot() returns the number of the line in the file **/etc/ttys** that describes the controlling terminal (see **ttys**).

Files

/dev/* — Terminal special files
/etc/ttys — Login terminals

See Also

libc

Diagnostics

ttyslot() returns zero if an error occurs.

ttystat — Command

Get terminal status
/etc/ttystat [-d] *port*

ttystat checks the status of the specified asynchronous *port* in directory **/dev**. It normally just returns an exit status that indicates the status of the *port*. The option **-d** tells **ttystat** to print the status of the *port* on the standard output.

Example

The following example prints the status of port **/dev/com2r**:

```
/etc/ttystat -d com2
```

If **/dev/com2r** is enabled, **ttystat** prints:

```
com2r is enabled
```

ttystat finds the port status from the **/etc/ttys** file.

Files

/etc/ttys — Terminal characteristics file

See Also

commands, disable, enable, ttys

Diagnostics

ttystat returns one if the *port* is enabled, or zero if the *port* is disabled. It returns -2 if an error occurs.

ttytype — Command

Select a default terminal type for a port
ttytype

The command **ttytype** selects a default terminal type for a given port.

The default terminal types are recorded in file **/etc/ttytype**. You must edit this file to ensure that the default terminal types are described correctly. The following gives a example version of **/etc/ttytype**:

```
ansipc    console    The COHERENT console
adm3a     com1l      The old Kaypro II
vt100     com2r      Remote logins
```

The first string gives the type of terminal. This string must name a terminal that is recognized by **termcap** and **terminfo** The second string gives the device with which this terminal type is linked. The **console** device should always be linked to terminal type **ansipc**. Other devices can be linked to the type of terminal most often used on them; on the above example, the user has a Kaypro II that is wired into his COHERENT system via a local serial port. **ttytype** ignores all strings after the first two in each line, so you can add comments to each entry, as in the above example.

You can use **ttytype** to set a terminal type automatically at login time. To do so, edit the file **/etc/profile** and replace the line

```
export TERM=ansipc
```

with the command:

```
export TERM=`/usr/bin/ttytype`
```

Files

/etc/ttytype — File of default terminal types

See Also

commands, termcap, terminfo

type checking — Definition

Every expression has a *type*, such as **int**, **char**, or **double**. C is not strongly typed, which means that it allows different types to be mixed relatively freely, and be changed (or **cast**) from one type to another.

COHERENT checks types more strictly than the C standard implies. COHERENT's type checking can be enabled or disabled in degrees, using **-VSTRICT** and other "variant" options with the **cc** command.

See Also

cc, Programming COHERENT, type promotion

type promotion — Definition

In arithmetic expressions, COHERENT promotes one signed type to another signed type by sign extension, and promotes one unsigned type to another unsigned type by zero padding. For example, **char** promotes to **int** by sign extension, whereas **unsigned char** promotes to **unsigned int** by zero padding.

See Also

data formats, Programming COHERENT

typedef — C Keyword

Define a new data type

typedef is a C facility that lets you define new data types. Such definitions are always made in terms of existing data types; for example,

```
typedef long time_t;
```

establishes the data type **time_t**, and defines it to be equivalent to a **long**. By convention, programmer-defined data types are written in capital letters.

Judicious use of the **typedef** facility can make programs easier to maintain, and improve their portability.

See Also

C keyword, manifest constants, portability, storage class
ANSI Standard, §6.5.6

types.h — Header File

Define system-specific data types
#include <sys/types.h>

The header file **types.h** defines a number of data types that are used throughout the COHERENT system.

See Also

header_fi
POSIX Standard, §2.5

typeset — Command

Set/list variables and their attributes
typeset
typeset [+-]fr
typeset [irx] *variable*=*value*

The command **typeset** is built into the Korn shell **ksh**. It sets or lists all variables and their attributes.

When called with an argument of the form *variable*=*value*, it sets variable *variable* to *value*. The following options modify *variable* or *value*:

- **i** Store *value* as an integer
- **r** Make *variable read-only*
- **x** Export *variable* to the environment

When called without an argument, **typeset** lists all variables and their attributes. When called with one of the following options, it lists the variables of the appropriate type. When prefixed with a hyphen '-', it prints the variable plus its value; when prefixed with a plus sign '+', it prints the variable alone:

- **f** List functions instead of variables
- **r** List read-only variables

See Also

commands, ksh

typo — Command

Detect possible typographical and spelling errors
typo [-nrs][*file ...***]**

typo proofreads an English-language document for typographical errors. It conducts a statistical test of letter digrams and trigrams in each input word against digram and trigram frequencies throughout the entire document. From this test, **typo** computes an index of peculiarity for each word in the document. A high index indicates a word less like other words in the document than does a low index. Built-in frequency tables ensure reasonable results even for relatively short documents.

typo reads each input *file* (or the standard input if none), and removes punctuation and non-alphabetic characters to produce a list of the words in the document. To reduce the volume of the output, **typo** compares each word against a small dictionary of technical words and discards it if found. The output consists of a list of unique non-dictionary words with associated index of peculiarity, most peculiar first. An index higher than ten indicates that the word almost certainly occurs only once in the document.

typo recognizes the following arguments:

-n Inhibit use of the built-in English digram and trigram statistics, and inhibit dictionary screening of words. More words will be output and the indices of peculiarity will be less useful for short documents.

-r Inhibit the default stripping of **nroff** escape sequences. Normally, **typo** strips lines beginning with '.' and removes the **nroff** escape sequences '\'.

-s Produce output files **digrams** and **trigrams** that contain, respectively, the digram and trigram frequency statistics for the given document. No indices of peculiarity are calculated or printed. If desired, these files may be installed in directory **/usr/dict**.

Files

/tmp/typo* — Intermediate files
/usr/dict/dict — Limited dictionary
/usr/dict/digrams — Digram frequency statistics
/usr/dict/trigrams — Trigram frequency statistics

See Also

commands, nroff, sort, spell

tzset() — Time Function (libc)

Set the local time zone
#include <time.h>
#include <sys/types.h>
void tzset()
extern long timezone; char *tzname[2][16];

tzset() is one of the suite of COHERENT functions that control and display the system's time. It searches for the environmental parameter **TIMEZONE**, which gives information on the local time zone. For more information on **TIMEZONE**, see its Lexicon entry.

If **TIMEZONE** is set, **tzset()** initializes the external variables **timezone** and **tzname**. **timezone** contains the number of seconds to be subtracted from GMT to obtain local standard time. **tzname[0]** and **tzname[1]** are character arrays that hold, respectively, the names of the local standard time zone and the local daylight-saving time zone. If **TIMEZONE** is not set, **timezone** defaults to zero, **tzname[0]** to **GMT**, and **tzname[1]** to the empty string.

See Also

date, ftime(), libc, localization, time [overview], TIMEZONE

Notes

tzset() used to be named **settz()**. It has been renamed to conform to published standards.

ulimit() — System Call (libc)

Get/set limits for a process
#include <ulimit.h>
long ulimit (*command* **[,** *blocks^***])**
int *command***,** *blocks^***;**

The system call **ulimit()** retrieves or sets limits on what a process can do. *command* indicates what you want it to do, as follows:

UL_GETFSIZE
Return the maximum size, in blocks, of a file that the current process can create.

UL_SETFSIZE
Limit to *blocks* the size of any regular file that any process can create. A process may decrease this limit, but only a process owned by the superuser **root** can increase it.

UL_GMEMLIM
Return the current process's break value. For details on the break value, see the Lexicon entry for **brk()**.

UL_GDESLIM
Return the maximum number of files that this process can open.

Each of the above commands is defined in the header file **ulimit.h**. When called to execute the command **UL_SETFSIZE**, **ulimit()** requires a second integer argument; when called to execute any other command, **ulimit()** takes only one argument.

If all goes well, **ulimit()** returns a non-negative value. **ulimit()** fails if any of the following occur:

- A process owned by someone other than the superuser **root** attempted to increase its file-size limit. **ulimit()** returns -1 and sets **errno** to **EPERM**.
- The first argument to **ulimit()** was something other than one of the above-named values. **ulimit()** returns -1 and sets **errno** to **EINVAL**.

See Also

brk(), libc, ulimit.h

Notes

ulimit() does not fail *per se* if you invoke it with option **UL_SETFSIZE** and do not supply a second argument. However, doing so will (or should) crash the process. *Caveat utilitor*.

ulimit.h — Header File

Define manifest constants used by system call ulimit()
#include <ulimit.h>

The header file **ulimit.h** defines manifest constants used with the system call **ulimit()**.

See Also

header files, ulimit()

umask — Command

Set the file-creation mask
umask [*OOO***]**

The *file-creation mask* modifies the default mode assigned to each file upon creation. The mode sets the permissions granted by the file's owner, plus other important information about a file.

The command **umask** sets the default file-creation mask to *OOO*, which are three octal numerals. If invoked without an argument, **umask** prints the current file-creation mask in octal.

Note that zero bits in the mask correspond to permitted permission bits in the target, and that execute permission cannot be enabled via any setting of *mask*. See the Lexicon entries for **umask()** and **chmod** for further details on file mode. The shell executes **umask** directly.

Example

Setting *mask* to octal 022 (i.e., 000 010 010) causes a file created with mode octal 0666 to actually have permissions of

```
rw- r-- r--
```

Setting *mask* to zero (i.e., 000 000 000) causes a file created with mode octal 0666 to actually have permissions of

```
rw- rw- rw-
```

See Also

chmod, commands, ksh, sh, umask()

umask() — System Call (libc)

Set file-creation mask
#include <sys/stat.h>
int umask(*mask***)**
int *mask***;**

umask() allows a process to restrict the mode of files it creates. Commands that create files should specify the maximum reasonable mode. A parent (e.g. the shell **sh**) usually calls **umask()** to restrict access to files created by subsequent commands.

mask should be constructed from any of the permission bits found by **chmod()** (the low-order nine bits). When a file is created with **creat()** or **mknod()**, every bit set in the *mask* is zeroed in *mode*; thus, bits set in *mask* specify permissions that will be denied.

umask() returns the old value of the file-creation mask.

Example

Setting *mask* to octal 022 (i.e., 000 010 010) causes a file created with mode octal 0666 to actually have permissions of

```
rw- r-- r--
```

Setting *mask* to zero (i.e., 000 000 000) causes a file created with mode octal 0666 to actually have permissions of

```
rw- rw- rw-
```

See Also

creat(), libc, mknod(), sh, stat.h
POSIX Standard, §5.3.3

Notes

A file's default permission cannot be set to execute regardless of the value of *mask*.

umount — Command

Unmount file system
/etc/umount *special*

umount unmounts a file system *special* that was previously mounted with the **mount** command.

The script **/bin/umount** calls **/etc/umount**, and provides convenient abbreviations for commonly used devices. For example, typing

```
umount f0
```

executes the command

```
/etc/umount /dev/fha0
```

The system administrator should edit this script to reflect the devices used on your specific system.

Files

/etc/mtab — Mount table
/dev/*
/bin/umount — Script that calls **/etc/umount**

See Also

clri, commands, fsck, icheck, mount

Diagnostics

Errors can occur if *special* does not exist or is not a mounted file system.

umount() — System Call (libc)

Unmount a file system
#include <sys/mount.h>
umount(*filesystem***)**
char **filesystem***;*

umount() is the COHERENT system call that unmounts a file system. *filesystem* names the block-special file through which the file system is accessed. Note that this must have been previously mounted by a call to **mount()**, or the call will fail.

See Also

libc, mount()

unalias — Command

Remove an alias
unalias *alias ...*

The command **unalias** is built into the Korn shell **ksh**. It removes each *alias*.

See Also

alias, commands, ksh

uname — Command

Print information about COHERENT
uname [-amnrsv]
uname [-S *systemname* **]**

The command **uname** prints information about the current implementation of COHERENT. It recognizes the following options:

-a Print all information.

-m Print the machine on which this implementation of COHERENT is running. This always defaults to the Intel 80386.

-n Print the name of your system, as set in the file **/etc/uucpname**.

-r Print the release of your copy of COHERENT.

-s Print the system name.

-S Change the system name. *systemname* is restricted to eight characters.

-v Print the version of COHERENT.

Example

The following script uses **uname** to implement a version of the Sun OS command **hostname**. It is by Cy Schubert (cschuber@bcsc02.gov.bc.ca):

```
#!/bin/sh -
# hostname - display or change the name of the host system
case $# in
    0)    uname -n;;
    1)    uname -S $1;;
    *)    echo Usage: hostname [new_hostname]
          exit 1;;
esac
```

See Also

commands

uname() — System Call (libc)

Get the name and version of COHERENT
#include <sys/utsname.h>
int uname(*name***)**
struct utsname **name**;

The COHERENT system call **uname()** identifies the current release of the COHERENT operating system. It writes its output into the structure pointed to by *name*. This must be of type **utsname**, which has the following members:

```
char sysname[SYS_NMLN];             /* system name */
char nodename[SYS_NMLN];            /* UUCP node name */
char release[SYS_NMLN];             /* current release */
char version[SYS_NMLN];             /* current version */
char machine[SYS_NMLN];             /* hardware */
```

uname() returns a non-negative value upon success. If something went wrong, i.e., *name* points to an invalid address, **uname()** returns -1 and sets **errno** to an appropriate value.

See Also

libc, utsname.h
POSIX Standard, §4.4.1

Notes

The COHERENT implementation of **uname()** conforms to POSIX Standard, which states that **uname()** returns a "non-negative" value upon success. To write portable code, your code must check for a return value that is greater than or equal to zero. It is an error to check for return value equal to zero, because the test works on some systems that adhere to the Standard but not on others.

uncompress — Command

Uncompress a compressed file
uncompress [*file ...* **]**

uncompress uncompressses one or more *file*s that had been compressed by the command **compress**.

Each *file*'s name must have the suffix **.Z**, which was appended onto it by **compress**; otherwise, **uncompress** prints an error message and exits. When **uncompress** has uncompressed a *file*, it removes the **.Z** suffix from that file's name.

If no *file* is specified on the command line, **uncompress** uncompresses matter read from the standard input, and writes its output to the standard output.

Older versions of **uncompress** could only uncompress files that had been compressed with option **-b12** or lower, with **-b12** being the default. The edition of **uncompress** released with COHERENT version 3.1 (and subsequent versions) can handle values up to 16.

See Also

commands, compress, compression, ram, zcat

unctrl.h — Header File

Define macro unctrl()
#include <unctrl.h>

The header file **unctrl.h** defines the macro **unctrl()** which changes a character from a control character to a printable character.

See Also

header files

ungetc() — STDIO Function (libc)

Return character to input stream
#include <stdio.h>
int ungetc (*c***,** *fp***)**
int *c***; FILE** **fp***;*

ungetc() returns the character *c* to the stream *fp*. *c* can then be read by a subsequent call to **getc()**, **gets()**, **getw()**, **scanf()**, or **fread()**. No more than one character can be pushed back into any stream at once. A call to **fseek()** will nullify the effects of a previous **ungetc()**.

ungetc() normally returns *c*. It returns **EOF** if the character cannot be pushed back.

Example

For an example of this function, see **fgetc()**.

See Also

fgetc(), getc(), libc
ANSI Standard, §7.9.7.11
POSIX Standard, §8.1

union — C Keyword

Multiply declare a variable

A **union** defines an area of storage that can accept any one of several types of data. In effect, it is a multiple declaration of a variable. For example, a **union** may be declared to consist of an **int**, a **double**, and a **char** *. Any one of these three elements can be held by the **union** at a time, and will be handled appropriately by it. For example, the declaration

```
union {
      int number;
      double bignumber;
      char *stringptr;
} example;
```

allows **example** to hold either an **int**, a **double**, or a pointer to a **char**, whichever is needed at the time. All of these have the same address. The elements of a **union** are accessed like those of a **struct**: for example, to access **number** from the above example, type **example.number**.

unions are helpful in dealing with heterogeneous data, especially within structures; however, you are responsible for keeping track of what data type the **union** is holding at any given time. Passing a **double** to a **union** and then reading the **union** as though it held an **int** will yield results that are unpredictable, and probably unwelcome.

Example

For an example of how to use a **union** in a program, see the entry for **byte ordering**.

See Also

C keywords, initialization, struct, structure
ANSI Standard, §3.1.2.5, §3.5.2.1

uniq — Command

Remove/count repeated lines in a sorted file
uniq [-cdu] [-*n*] [+*n*] [*infile*[*outfile*]]

uniq reads input line by line from *infile* and writes all non-duplicated lines to *outfile*. The input file must be sorted. **uniq** uses the standard input or output if either *infile* or *outfile* is omitted. The following describes the available options:

-c Print each line once, discarding duplicate lines; before each line, print the number of times it appears within the file.

-d Print only lines that are duplicated within the file; print each line only once; do not print any counts.

-u Print only lines that are *not* duplicated within the file.

uniq by default behaves as if both **-u** and **-d** were specified, so it prints each unique line once.

Optional specifiers allow **uniq** to skip leading portions of the input lines when comparing for uniqueness.

-*n* Skip *n* fields of each input line, where a field is any number of non-white space characters surrounded by any number of white space characters (blank or tab).

+*n* Skip *n* characters in each input line, after skipping fields as above.

See Also

comm, commands, sort

unistd.h — Header File

Define constants for file-handling routines
#include <unistd.h>

The header file **unistd.h** defines standard routines used by the UNIX and UNIX-like operating systems. It prototypes many commonly used functions, and declares manifest constants used when checking file access, setting the seek pointer, and locking files.

See Also

access(), fseek(), header files, lockf()
POSIX Standard, §2.9

units — Command

Convert measurements
units [-u]

units is an interactive program that tells you how to convert one unit of measurement into another. It prompts you for two quantities with the same dimension (e.g., two measurements of weight, or two of size). It first prints the prompt "You have:" to ask for the unit you wish to convert from, and then prints the prompt "You want:" for the unit you wish to convert to.

Example

The following example returns the formula for convert fortnights into days:

```
You have: fortnight
You want: days
* 14
/ 0.071428
```

The following fundamental units are recognized: **meter**, **gram**, **second**, **coulomb**, **radian**, **bit**, **unitedstatesdollar**, **sheet**, **candle**, **kelvin**, and **copperpiece** (shillings and pence).

A quantity consists of an optional number (default, 1) and a dimension (default, none). Numbers are floating point with optional sign, decimal part and exponent. Dimensions may be specified by fundamental or derived units, with optional orders. A quantity is evaluated left to right: a factor preceded by a '/' is a divisor, otherwise it is a

multiplier. For example, the earth's gravitational acceleration may be entered as any of the following:

```
9.8e+0 m+1 sec-2
32 ft/sec/sec
32 ft/sec+2
```

British equivalents of US units are prefixed with **br,** e.g., **brpint**. Other units include **c** (speed of light), **G** (gravitational constant), **R** (gas-law constant), **phi** (golden ratio) % (1/100), **k** (1,024), and **buck** (United States dollar).

/usr/lib/units is the ASCII file that contains conversion tables. The binary file **/usr/lib/binunits** may be recreated by using the **-u** option.

Files

/usr/lib/units — Known units
/usr/lib/binunits — Binary encoding of units file

See Also

bc, commands, conv

Diagnostics

If the ASCII file **/usr/lib/units** has changed more recently than the binary file **/usr/lib/binunits**, **units** prints a message and regenerates the binary file before it continues; this can take up to a few minutes, depending upon the speed of your system.

The error message "conformability" means that the quantities are not dimensionally compatible, e.g., **m/sec** and **psi**. **units** prints each quantity and its dimensions in fundamental units.

Notes

There are the inevitable name collisions: **g** for gram versus **gee** for Earth's gravitational acceleration, **exp** for the base of natural logarithms versus **e** for the charge of an electron, **ms** for (plural) meters versus **millisecond,** and, of course, **batman** for the Persian measure of weight rather than the Turkish.

unlink() — System Call (libc)

Remove a file
#include <unistd.h>
int unlink(*name***) char** ***name*;**

unlink() removes the directory entry for the given file *name*, which in effect erases *name* from the disk. *name* cannot be opened once it has been **unlink()**'d. If *name* is the last link, **unlink()** frees the i-node and data blocks. Deallocation is delayed if the file is open. Other links to the file remain intact.

Example

This example removes the files named on the command line.

```
#include <unistd.h>
main(argc, argv)
int argc; char *argv[];
{
      int i;

      for (i = 1; i < argc; i++) {
            if (unlink(argv[i]) == -1) {
                  printf("Cannot unlink \"%s\"\n", argv[i]);
                  exit(EXIT_FAILURE);
            }
      }
      exit(EXIT_SUCCESS);
}
```

See Also

libc, link(), ln, remove(), rm, rmdir, unistd.h
POSIX Standard, §5.5.1

Diagnostics

unlink() returns zero when successful. It returns -1 if *file* does not exist, if the user does not have write and search permission in the directory containing *file*, or if *file* is a directory and the invoker is not the superuser.

unpack — Command

GNU utility to uncompress files
unpack [-cfhLrtvV] [*file ...* **]**

unpack uncompresses each *file* named on its command line. Each *file* must have been compressed by the COHERENT commands **gzip** or **compress**, or by the UNIX command **pack**. If no *file* appear on its command line **unpack** uncompresses what it reads from the standard input.

unpack is a link to the command **gunzip**. For details on its command-line options, see the Lexicon entry for **gunzip**.

See Also

commands, gzip, gunzip

unset — Command

Unset an environment variable or shell function
unset *environmental_variable*
unset -f *shell_function*

The command **unset** unsets an environmental variable or shell function.

When used with the option **-f**, **unset** unsets the shell function named on the command line. This option applies only to the Bourne shell **sh**.

When used without the option **-f**, **unset** unsets the environmental variable named on the command line. This version of the command applies to both the Bourne shell **sh** and the Korn shell **ksh**.

See Also

commands, environmental variables, ksh, sh

unsigned — C Keyword

Data type

unsigned tells the compiler to treat the variable as an unsigned value. In effect, this doubles the largest absolute value that that type can hold, and changes the lowest storage value to zero.

See Also

C keywords, data type
ANSI Standard, §6.2.1.2

until — Command

Execute commands repeatedly
until *sequence1* **[do** *sequence2* **] done**

The shell's **until** loop executes the commands in *sequence1*. If the exit status is nonzero, the shell then executes the commands in the optional *sequence2* and repeats the process until the exit status of *sequence1* is zero. Because the shell recognizes a reserved word only as the unquoted first word of a command, both **do** and **done** must occur either unquoted at the start of a line or preceded by ';'.

The shell commands **break** and **continue** may be used to alter control flow within an **until** loop. The contruct **while** has the same form as **until** but the sense of the test is reversed.

The shell executes **until** directly.

See Also

break, commands, continue, ksh, sh, test, while

unzip — Command

Un-zip a zipped archive
unzip *archive* **[-cfpux** *file ...***] [-ltvz] [-anojqUV]**

The command **unzip** extracts files from a zipped archive. It recognizes the following command-line options:

-c [*file ...***]**
Extract *file*, but write them to the standard output instead of to disk.

-f [*file ...***]**
"Freshen" files: Extract *file* from *archive* and write it to disk, but do so only if the file in the archive is newer than the file on disk. Do not create new files.

-l List the contents of the archive, short format.

-p [*file ...* **|** *command***]**
Extract each *file* and pipe it to *command*.

-t Test the integrity of *archive*.

-u [*file ...***]**
Update each *file* within the archive. Create the file if necessary.

-v List files, verbose format.

-x [*file ...***]**
Extract each *file* from default. If no *file* argument is given, extract all files. This is the default.

-z Display archive's comments, if any.

The following modify the behavior of the options:

-a Convert text from MS-DOS format to UUCP format.

-j Ignore ("junk") paths; do not make directories.

-n Never overwrite existing files.

-o Overwrite files without prompting.

-q Quiet mode.

-qq Quieter mode.

-U Do not convert file names to lower-case letters.

The following example extracts file **ReadMe** from archive **data1**:

```
unzip data1 ReadMe
```

The next example extracts all files from archive **foo.zip** and pipes them to the pager **more**:

```
unzip -p foo | more
```

The final example "freshens" files on disk from the contents of **foo.zip**. Files are overwritten without prompting:

```
unzip -fo foo
```

Notes

commands, compress, gunzip, gzip, uncompress, zip

Notes

Do not confuse this command with **gunzip**. Archives made by **gzip** may not be extractable by **unzip**.

upac — Command

De-fragment a file system without sorting
upac *raw_device*

Command **upac** uses de-fragments file system *raw_device* without sorting it by access date. Rather, it orders files by i-node number.

LEXICON

See Also

commands, dpac, fmap, fsck, qpac, spac

Notes

upac is a link to the command **dpac**.

upac was written by Randy Wright (rw@rwsys.wimsey.bc.ca).

update — System Administration

Update file systems periodically
/etc/update

update periodically calls **sync** to write to the disk all file system data that are in memory. It never exits.

The initialization command file **/etc/rc** normally executes **update**. It should not be executed directly.

See Also

Administering COHERENT, init, sync

uproc.h — Header File

Definitions used with user processes
#include <sys/uproc.h>

uproc.h defines the constants and structures used by routines that manage user processes.

See Also

header files

USER — Environmental Variable

Name user's identifier
USER=*user_identifier*

The environmental variable **USER** names your login identifier. For example, if your login identifier is **fwb**, then by typing **set** you will see the entry **USER=fwb**. **USER** is set by **login**.

See Also

environmental variables, ksh, login, LOGNAME, sh

Using COHERENT — Overview

For an ordinary user — that is, one who neither administers the COHERENT system nor writes programs for it — using COHERENT mainly involves issuing commands to the COHERENT system.

The Lexicon entry **commands** names every command that comes with the COHERENT system. The commands are grouped by function. You should look carefully at the shell commands — that is, the commands that work closely with the shell to help you control the execution of other commands. What other groups you study will depend on just what you want to do with your COHERENT system.

Pay particular attention to the Lexicon entries for the commands **sh**, **ksh**, and **vsh**. These introduce the *shells* — that is, the programs with which you can issue commands to COHERENT. Each has its own syntax; **ksh** and **sh** in fact implement fully flown programming languages on their own.

vsh is a visual shell, and is especially useful to beginners. It uses a visual interface and drop-down menus to make it easy for you to issue commands without having to remember convoluted command syntax. The Lexicon entry for **vsh** describes it, and how you can customize it for yourself.

The Lexicon entry for **MS-DOS** compares COHERENT with MS-DOS, and describes how they differ. It also gives a table of COHERENT equivalents to commonly used MS-DOS commands. If you are used to using MS-DOS, you should find this useful.

The follow commands help you to find information about your system:

apropos
: This command searches the description of each Lexicon article for a keyword that you enter. In this way, you can quickly find which articles discuss a given topic, such as "printer" or "modem".

help This command displays a brief summary of each Lexicon article, by name.

man This command displays Lexicon articles on your screen, by name.

The following three articles introduce files that are stored in your home directory. By modifying these files, you can customize your COHERENT account to suit your tastes:

.kshrc Script **$HOME/.kshrc** configures the Korn shell to suit your tastes. You will need to edit this file if you decide to use the Korn shell.

.lastlogin
File **$HOME/.lastlogin** records the date and time you last logged in to your COHERENT system.

.profile Script **$HOME/.profile** holds commands that are executed when a given user logs in to your COHERENT system.

The following Lexicon entries hold technical information that you probably will find useful:

block This defines the size of a "block" on a mass-storage device.

compression
This introduces the subject of compression, and the programs with which you can compress and de-compress files. It also gives a table that describes how to de-compress files based on their default suffices.

environmental variables
This article lists the commonly used environmental variables that are described in the Lexicon. These variables control many of the behaviors of the COHERENT system.

Lexicon
This describes the format of the printed COHERENT manual. It also summarizes changes made to on-line Lexicon pages (the ones that you view via the command **man**) since the manual was last printed.

man This summarizes the **man** macros that are used by the text-formatter **nroff**.

ms This summarizes the **ms** macros that are used by the text-formatter **nroff**.

Finally, the following Lexicon entries define technical terms that are used in this manual:

caveat utilitor
daemon
directory
file
filter
i-node
named pipe
pipe
process
root
sticky bit
superuser
wildcards

For pointers on where to look for information on how to install and modify peripheral devices on your system, such as the keyboard, the hard disk, or a CD-ROM drive, see the Lexicon entry **Administering COHERENT**.

See Also

Administering COHERENT, COHERENT, Programming COHERENT

usleep() — Sockets Function (libsocket)

Sleep briefly
long usleep (*t***)**
long *t***;**

The function **usleep()** sleeps for *t* milliseconds or until it receives a signal.

See Also

libsocket, nap()

Notes

usleep() is included for compatibility with Berkeley socket code. It is the equivalent of the System V Release 4 system call **nap()**.

usrtime — System Administration

Times each user is permitted to log in
/etc/usrtime

File **/etc/usrtime** holds the time, day of the week, and terminal line upon which a given user can log into your COHERENT system. Command **login** reads it to see if a user who is attempting to log in is doing at a permitted time and via a permitted line. If a user is not named in this file, **login** assumes that she can log in at any time, via any line.

usrtime consists of an indefinite number of lines, each with the following format:

users:*enable*:*tty*:*weekday*:*time*:*comment*

The following describes each field in detail.

user The login identifiers of the user or users to be restricted. Multiple identifiers must be separated by commas. Each identifier must be defined in **/etc/passwd**. If this field is empty, then the line is a default for every user not specifically named elsewhere in **usrtime**.

The keywords **ALL**, **UUCP**, **SLIP**, and **INTERACTIVE** can also be used in this field, to name categories of users. They are described in detail below.

enable Enable or disable the login (or logins). **NOLOGIN** disables the login; **LOGIN** or an empty field enables it. A range of dates of the form

```
yyyymmdd-yyyymmdd
```

enables logins only during those dates. This field can contain more than one range of dates; if it does, the ranges must be separated by a comma. Prefixing a range of dates with a '!' disables logins between those dates.

tty This field lists the devices via which the user (or users) may log in — usually a **tty** or **com** device. If this field names more than one device, they must be separated by commas. A device name can contain the wildcard character '?'; for details on how this is interpreted, see the Lexicon entry for **wildcards**. If a device is prefixed with a '!', the user cannot log in on that device. If this field is empty, then the user can log in on all devices.

weekday
This field lists the days of the week upon which the the user (or users) can log in. If more than one day is named, they must be separated by commas. Each day is identified by the first three letters of its name. If a weekday is prefixed with a '!', then the users cannot log in on that day. If this field is empty, the users can log in on any day of the week.

time This field gives range of time during which the user (or users) may log in. Time is given in the form:

```
hhmm-hhmm
```

If more than one range is named, they must be separated by commas. Prefixing a range with a '!' forbids the user to log in during between those times. If this field is empty, then the user can log in during any time of the day.

comment
This field holds some commentary, presumably helpful to others who must read this file. **login** ignores this field.

Scope of Entries

A user may be affected by more than entry in this file. The order in which the entries appear is significant.

At the top of the file should appear the entries that are being excluded from restriction. These should include such users as **bin** and **daemon**, plus any ordinary user you wish to exclude from being restricted. The entries for such a users should consist of her (its) name, followed by five colons. Any user named in such an entry is immune to any restrictions that may appear below in this file.

Next should come the global restrictions, that is, restrictions for entire categories of users. As mentioned above,

you can use the keywords **ALL**, **UUCP**, **SLIP**, or **INTERACTIVE** to describe users. These keywords have the following meaning:

ALL All users.
UUCP All "users" who are UUCP accounts — i.e., whose shell as set in **/etc/passwd** is **/usr/lib/uucp/uucico**.
SLIP All "users" who are SLIP accounts — i.e., whose shell is **sllogin**.
INTERACTIVE
Users who have an interactive the interactive shell **ksh** or **sh** set at login.

Last should come entries for individual users or clusters of users. These restrictions can be set in addition to those set for categories of users. An entry for an individual users that appears below the global entries will not loosen the restrictions set globally for that user; but it can tighten them.

Note that **login** ignores any restrictions set for the superuser **root**. Finally **login** ignores every line that begins with a '#'. You can use such lines to hold comments.

Example

The following gives an example **usrtime** file:

```
# <user>:<enable>:<tty>:<weekday>:<time>:<comment>
sys,bin,daemon:::::
INTERACTIVE::/dev/com??,/dev/color?:Mon,Tue,Wed,Thu,Fri:0630-1830:
UUCP::/dev/com2l:::UUCP accounts
::::0800-1700:default for anybody not mentioned below
fred,anne:LOGIN:/dev/color?::0830-1630:administration
ivan,marian:LOGIN:/dev/com??:::secretarial staff
catherine:19930401-19931130::::consultant programmer
```

See Also

Administering COHERENT, login

Notes

No line in **usrtime** can exceed 500 characters.

ustat() — System Call (libc)

Get statistics on a file system
#include <sys/types.h>
#include <ustat.h>
int ustat (*device***,** *buffer***)**
dev_t *device***;**
struct ustat **buffer***;*

The COHERENT system call **ustat()** returns information about a mounted file system. *device* names the device upon which the file system is mounted. *buffer* points to a structure of type **ustat**, which contains the following fields:

```
daddr_t        f_tfree;              /* number of free blocks */
ino_t          f_tinode;             /* number of free i-nodes */
char           f_fname[6];           /* name of the file system */
char           f_fpack[6];           /* pack name of the file system */
```

Useful information may not be available for fields **f_fname** and **f_fpack**; in that case, they are initialized to nuls.

ustat() returns zero if all goes well; otherwise, it returns -1 and sets **errno** to an appropriate value. **ustat()** can fail for any of the following reasons:

- *device* does not contain a mounted file system.
- *buffer* points to an illegal address.
- The kernel caught a signal while it was executing the call.

See Also

libc, mkfs, statfs()

Note

ustat() is largely superceded by **statfs()**.

utime() — System Call (libc)

Change file access and modification times
#include <sys/types.h>
#include <utime.h>
int utime(*file, times***)**
char ******file***;**
time_t *times***[2];**

utime() sets the access and modification times associated with the given *file* to times obtained from *times***[0]** and **times[1]**, respectively. The time of last change to the attributes is set to the time of the **utime()** call.

This call must be made by the owner of *file* or by the superuser.

Files

<sys/types.h>

See Also

libc, restor, stat(), utime.h
POSIX Standard, §5.6.6

Diagnostics

utime() returns -1 on errors, such as if *file* does not exist or the invoker not the owner.

utime.h — Header File

Declare system call utime()

The header file **<utime.h>** declares the COHERENT system call **utime()**. It also defines the structure **utimbuf**, which **utime()** uses.

See Also

header files, utime()

utmp — System Administration

File that notes login events that are active
/etc/utmp

File **/etc/utmp** notes every login event that is active — that is, when the user has logged in and has not yet logged out. It is read by the command **who** to display the users who are now logged into your system.

utmp records each active login event as a record of type **utmp**, which is defined in header file **<utmp>**. For details, see the Lexicon entry **utmp.h**.

File **/usr/adm/wtmp** records every login event that has concluded. You can comb this file to trace which user have logged onto your system, and when.

See Also

Adminstering COHERENT, utmp.h, wtmp

utmp.h — Header File

Login accounting information
#include <utmp.h>

Header file **<utmp.h>** defines the types and constants that are used to manipulate the system-adminstration files **/etc/utmp** and **/usr/adm/wtmp**. The former file describes every user who is currently logged into your system; the latter records when each user logged into your system and logged out again.

Each of these files consists of records, each of which has are objects of type **utmp**, which **<utmp.h>** defines as follows:

```
struct      utmp {
      char ut_user[8];
      char ut_id[4];
      char ut_line[12];
      short ut_pid;
      short ut_type;
      struct exit_status {
            short e_termination;
            short e_exit;
      } ut_exit;
      time_t ut_time;
};
```

The following describes each field in **utmp**:

ut_user
The login identifier of the user.

ut_id The user's identifier, as taken from **/etc/init**.

ut_line The device through which the user logged in.

ut_pid The process identifier of the user's shell.

ut_type
Type of entry in this file. This can be any of the following values:

EMPTY	An empty entry
RUN_LVL	Run level
BOOT_TIME	Boot time
OLD_TIME	
NEW_TIME	
INIT_PROCESS	Process spawned by **init**
LOGIN_PROCESS	A **getty** waiting for a login
USER_PROCESS	A user process
DEAD_PROCESS	
ACCOUNTING	

ut_exit The process's exit status. It consists of the following fields:

e_termination
Process's termination status.

e_exit Process's exit status.

ut_time
The time the user logged on.

The following functions use this header file:

endutent() . . . . . Close the logging file.
getutent() . . . . . Read the next entry from **/etc/utmp**.
getutid() . . . . . Find an entry in **/etc/utmp** by login identifier.
getutline() . . . . . Find an entry in **/etc/utmp** by login device.
pututline() . . . . . Write a record into **/etc/utmp**.
setutent() . . . . . Rewind the input stream that is reading **/etc/utmp**
utmpname() . . . . . Manipulate a file other than **/etc/utmp**.

Each function is described in its own Lexicon entry.

Files

/etc/utmp
/usr/adm/wtmp

See Also

ac, header files, login, utmp, who, wtmp

LEXICON

utmpname() — General Function (libc)

Manipulate a login logging file other than /etc/utmp
#include <utmp.h>
int utmpname(*file***)**
const char ******file***;**

The system files **/etc/utmp** and **/usr/adm/wtmp** record information about every login event on your system — that is, they record every time someone logs into your system, the line from which the user logged in, and how long he was logged in. COHERENT comes with a set of functions that manipulate these files: they let you open a logging file, reads records, and update them.

By default, these functions manipulate the file **/etc/utmp**, which records the login events that are active — that is, the user has logged in but not yet logged out. Function **utmpname()** lets you change the file being manipulated. *file* points to the name of the file you wish to manipulate. Usually, this is the file **/usr/adm/wtmp**, which records login events that have concluded; but you can name any file in which you or the system has recorded login events. **utmpname()** also closes the logging file that is already open.

See Also

libc, utmp.h

utsname.h — Header File

Define utsname structure
#include <sys/utsname.h>

utsname.h defines the structure **utsname**. This structure holds information that describes a given release of the COHERENT system.

See Also

header files, uname()
POSIX Standard, §4.4.1

uuchk — Command

Check UUCP configuation
/usr/lib/uucp/uuchk [-I*file***] [v] [--help]**

The command **uuchk** reads the UUCP configuration files **sys**, **port**, and **dial**, and generates a report on the configuration for each remote system listed in **sys**. You can use this report to repair problems in your configuration files.

The following gives sample output for system **mwcbbs**:

```
Call out using port intel.slow at speed 2400
The possible ports are:
 Port name intel.slow
  Port type modem
  Device /dev/com3fl
  Speed 2400
  Carrier available
  Hardware flow control available
  Dialer intel.slow
   Chat script "" AT\s&C1\s&D2\sE1\sM1\sQ0\sS0=0\sV1\sDP\D CONNECT\s2400
   Chat script timeout 60
   Chat failure strings BUSY NO\sCARRIER NO\sANSWER
   Chat script incoming bytes stripped to seven bits
   Wait for dialtone ,
   Pause while dialing ,
   Carrier available
   Wait 60 seconds for carrier
   When complete chat script "" \d+++\dAT\sH0\sE0\sV0\sQ1\sM0\sS0=1
   When complete chat script timeout 60
   When complete chat script incoming bytes stripped to seven bits
   When aborting chat script "" \d+++\dAT\sH0\sE0\sV0\sQ1\sM0\sS0=1
   When aborting chat script timeout 60
```

```
   When aborting chat script incoming bytes stripped to seven bits
Phone number 17085590445
Chat script "" \r\d\r in:--in: nuucp word: public word: 127417124
Chat script timeout 10
Chat script incoming bytes stripped to seven bits
At any time may call if any work
May make local requests when calling
May make local requests when called
May send by local request: /
May send by remote request: /usr/spool/uucppublic /tmp
May accept by local request: ~
May receive by remote request: /usr/spool/uucppublic /tmp
May execute rmail uucp
Execution path /bin /usr/bin /usr/local/bin
Will leave 50000 bytes available
Public directory is /usr/spool/uucppublic
Will use protocols g
For protocol g will use the following parameters
 window 3
 packet-size 64
```

uuchk recognizes the following command-line options:

-I*file*
--config*file*
: Use *file* instead of the standard configuration files. This option lets you sanity-check a new configuration file without having to install it.

-v
--version
: Print the version of **uuchk** and exit.

--help Print a help message, and exit.

See Also

commands, dial, port, sys, UUCP

uucico — Command

Communicate with a remote site
/usr/lib/uucp/uucico [-D] [-c*site***] [-I***file***] [-p***port***] [-r0] [-r1] [-s***site***] [-S***site***] [-x***level***]**

The UUCP daemon **uucico** is the program that communicates with a remote *site*. It either contacts another site and issues commands for execution by another **uucico** process on that remote system (*master* mode); or it receives a call from a remote system and executes the commands that that system issues (*slave* mode).

The commands **uucp** and **uux** invoke **uucico** automatically, usually in master mode. **uucico** can also be invoked directly from the shell, from within a script, or from with a **cron** file.

You can also name **uucico** in file **/etc/passwd** as the default process to run for a given login identifier. A system that logs in under that login ID (presumably, a version of **uucico** on a remote system) will interact with your system's **uucico**, instead of a shell. When invoked in this manner, **uucico** runs in slave mode by default.

After **uucico** has finished communicating with the remote system, it invokes the daemon **uuxqt** to execute the commands issued by the remote system. For information on **uuxqt**, see its Lexicon entry.

uucico recognizes the following command-line options:

-c*site* "Cron" mode: If a call is not permitted to *site* at the present time, then do not make the call; but also, do not log an error message or update the system status. Use this option if you wish to invoke **uucico** regularly through **cron**, and do not want to be bombarded with error messages should the entry in **cron** conflict with the legal calling times set in **sys**.

-D Do not detach from the device until the contact with the remote system concludes.

-e Force **uucico** to produce its own **login:** and **Password:** prompts. **uucico** checks the password it receives against its own, private list, rather than against the password kept in file **/etc/passwd**. This should be used with the options **-l** and **-p**. When used with this option, **uucico** does not terminate, but continues to issue prompts until you kill it explicitly. This option permits you to use **uucico** as a server on a network.

-f*site* Force option: call *site* immediately, regardless of whether the site's description in **sys** indicates that this is a legal time to call.

-I *file* Read configuration information from *file*, instead of from the default file **/usr/lib/uucp/sys**.

-l Force **uucico** to produce its own **login:** prompt. **uucico** checks the login it receives against its own, private list, rather than against the normal system password files. This should be used with the option **-e**.

-q Quiet: do not invoke daemon **uuxqt** on the remote system.

-p*port* Use *port*. When used with the options **-s** or **-S**, dial out on *port*; this overrides the default port used with the system being contacted. When **uucico** is in slave mode, this implies the option **-e**.

-r0 Act as slave in polling process; that is, carry out the orders of another **uucico** that has dialed into your system. This is the default.

-r1 Act as master in polling process; that is, dial out to another system and give it orders. This option is implied by options **-s** or **-S**. If the **uucico** command line does not name a site to call, this option tells **uucico** to call any system for which work is waiting to be performed.

-s*site* Call *site*. This must name one of the entries in **/usr/lib/uucp/sys**.

-S*site* Call *site* immediately, if the present time lies within the legal time set for *site*, as described in file **/usr/lib/uucp/sys**.

-w After contacting a system with the options **-r1**, **-s**, **-S**, begin an endless loop of login prompts, as with the option **-e**. In effect, UUCP calls a remote site; but instead of logging into that site, it lets that site log into it.

-x*activity***[,***activity***,...,***activity***^]**
-X*activity***[,***activity***,...,***activity***]**
Log a given *activity*. These logs can help you debug problems with UUCP. **uucico** recognizes the following activities:

abnormal	**chat**	**config**
execute	**handshake**	**incoming**
outgoing	**port**	**proto**
spooldir	**uucp-proto**	

One **-x** option can name multiple activities, separated by commas. A **uucico** command line can contain more than **-x** option. **uucico** writes its logging information into file **/usr/spool/uucp/.Admin/audit.local**.

Example

To poll the site **mwcbbs** (the Mark Williams bulletin board) five minutes after each hour, put the following entry into a **cron** file:

```
05 * * * * /usr/lib/uucp/uucico -smwcbbs -r1
```

Files

/usr/lib/uucp/sys — List of reachable systems
/usr/spool/uucp/.Log/uucico/*sitename*— **uucico** activities log file for *sitename*
/usr/spool/uucp/.Log/uucico/UUCICO— uucico debug log
/usr/spool/uucp/*sitename*— Spool directory for work

See Also

commands, cron, uucp, UUCP, uulog, uutouch, uuxqt

Notes

uucico was written by Ian Lance Taylor (ian@airs.com).

uuconv — Command

Convert UUCP configuration files to Taylor format
/usr/lib/uucp/uuconv -i *input* **-o** *output* **[-p** *program***] [-I** *file***]**

The command **uuconv** converts UUCP configuration files from one format to another. In all probability, you will have to run this program only once, when you convert from your previous UUCP implementation to Taylor UUCP.

uuconv recognizes the follow command-line options:

--help Print a help message and exit.

-I *file*
--config *file*
Read configuration information from *file* instead of from the standard UUCP configuration file.

-i *file*
--input *file*
Read input from *file*.

-o *file*
--output *file*
Write output into *file*.

-p *program*
--program *program*
Convert *program* (e.g., **uucp** or **cu**).

-v
--version
Print the version of **uuconv** that you are running, and exit.

See Also

commands, UUCP

Notes

uuconv was written by Ian Lance Taylor (ian@airs.com).

UUCP — Overview

Unattended communication with remote systems

UUCP stands for "UNIX to UNIX communications protocol". It is a system of commands that allows you to exchange files with other COHERENT or UNIX systems, in an unattended manner. With UUCP, you can send mail to other systems, upload files, and execute commands. When configured correctly, UUCP also lets other users upload files to your system, copy files from it, and execute commands. All this can be done without your having to sit at your console and type commands; thus, files can be transferred in the small hours, when telephone rates are lower and computers are relatively free.

UUCP gives you access to the Usenet, a nation-wide network of UNIX and COHERENT users. Access to the Usenet will let you exchange mail with any of the thousands of Usenet users, receive mail from them, download source code for many useful programs, and read the latest news on a host of subjects. For details on contacting UUNET, a commercially accessible Usenet site, enter the command:

```
phone uunet
```

Implementation of UUCP

Beginning with release 4.2, COHERENT implements the Taylor UUCP package. The current implementation is Taylor UUCP version 1.05. Taylor UUCP offers extraordinary flexibility, beyond that offered by standard implementations of UUCP. The following Lexicon entries describe UUCP:

config. . . . . Overall configuration file for UUCP
cu . . . . . Introduce the **cu** communications utility
dial . . . . . Describe how **uucico** and **cu** can dial a modem
domain. . . . . Describe the file that names your UUCP domain
port . . . . . File that describes ports through which UUCP dials

LEXICON

sys . . . . . File that describes systems contacted by UUCP
uuchk . . . . . Check UUCP configuation
uucico . . . . . Daemon that controls communication with a remote site
uuconv . . . . . Convert UUCP configuration files to Taylor format
uucp . . . . . Spool files for transmission to other systems
uucpname . . . . . File that names your system
uudecode . . . . . Decode a binary file sent from a remote system
uuencode . . . . . Encode a binary file for transmission
uuinstall . . . . . Install or modify UUCP
uulog . . . . . Read a UUCP log
uumkdir . . . . . Create a UUCP directory
uumvlog . . . . . Archive UUCP log files
uuname . . . . . List UUCP names of known systems
uupick . . . . . Pick up a file uploaded from a remote system
uurmlock . . . . . Remove a UUCP lockfile
uusched . . . . . Call all systems that have jobs waiting for them
uustat . . . . . Display and modify the status of a UUCP job
uuto . . . . . Send a file to a remote system
uutouch . . . . . Touch a file to trigger **uucico** poll
uutry . . . . . Debugging tool for UUCP
uux . . . . . Execute a command on a remote system
uuxqt . . . . . Execute commands requested by a remote system

Files and Directories

UUCP uses the following files and directories:

/usr/lib/uucp/sys
This file contains information about remote UUCP sites with which you can communicate. **uucico** uses its information to connect to remote systems; sets permissions for the directories that a given remote system can write into or read from; establishes the protocol (or protocols) that can be used when communicating with the given remote system to transfer files.

/usr/lib/uucp
This directory holds many of the UUCP executables. It also holds the following configuration files:

/usr/lib/uucp/config
Customize the configuratio of Taylor UUCP. Note that this file is not shipped with COHERENT, to ensure that the default configuration is used; however, you can write one yourself easily enough. For details, see the Lexicon entry **config**.

/usr/lib/uucp/dial
uucico uses the information in this file to communicate with modems.

/usr/lib/uucp/port
uucico uses the information in this file to communicate with a given port on your system.

/usr/spool/uucp
This directory holds log files and spool directories, as follows:

/usr/spool/uucp/.Admin
This directory holds the following administrative logging files:

/usr/spool/uucp/.Admin/xferstats
This file holds statistics about the rate at which data were transferred between your site and a remote site.

/usr/spool/uucp/.Admin/audit.local
This file holds auditing information, as generated using the option **-x** with any UUCP command.

/usr/lib/uucp/.Log
This directory holds information that detail the files transferred between your system and any remote system. It contains one sub-directory for each UUCP command — one each for **uucico**, **uucp**, **uux**, and **uuxqt**. Each sub-directory, in turn, contains one log file for each remote system with which your system exchanges files, plus the file **ANY**, which holds information about all remote systems. For example, file **/usr/spool/uucp/.Log/uucp/lepanto** logs every file that you

have exchange with remote site **lepanto** via the command **uucp**.

/usr/spool/uucp/.Received
This directory contains one sub-directory for each remote system with which your system exchanges files. It holds files received from that system that cannot be executed properly. If your system is configured correctly, this directory should be empty.

/usr/spool/uucp/.Sequence
This directory holds one file for each remote system with which you exchange files. The file holds a string from which the job most recently performed with that site was named. This sequence number is used to identify each job uniquely. This is discussed in more detail below.

/usr/spool/uucp/.Status
This directory holds one file for each system with your system communicates via UUCP. The file holds information about the status with which the last contact exited. For example, if your system communicated successfully with system **mwc**, then file **/usr/spool/uucp/.Status/mwc** will hold an entry that resembles the following:

```
0 0 778536664 0 SUCCESSFUL mwc
```

However, if your system communicates with system **sales** and the last session failed during handshake, then file **/usr/spool/uucp/.Status/sales** will hold something like the following:

```
4 7 769981110 4200 Handshake failed sales
```

Note that if a **.Status** file indicates that the last contact failed, **uucico** may silently refuse to dial out to that system; UUCP is designed this way, in order to spare you the expense of repeatedly calling a system whose connection is damaged in some way. The solution is simply to remove the file in question. For example, if **uucico** refuses to dial system **mwc** and you know that that system is working correctly, try removing file **/usr/spool/uucp/.Status/mwc**.

/usr/spool/uucp/.Temp
This directory holds one directory for each system with which your system has exchanged files. Each sub-directory holds temporary files used by the jobs being performed for that system.

/usr/spool/uucp/.Xqtdir
The command **uuxqt** executes from within this directory all commands that have been spooled onto your system for execution. It also copies into this directory all files on remote systems that a spooled command names. Note that files reside here only briefly.

/usr/spool/uucp/*sitename*
This directory holds all files being uploaded to site *sitename*. Each file is constructed as follows:

prefix This is either **D.** or **C.**. The former indicates a data file, and the latter a command file (that is, a file to be executed on the remote system by command **uux**).

site The name of the site to which the file is being uploaded.

sequence_number
This is a unique number, meant to ensure that no UUCP file clobbers another. When UUCP is spooling a file to be transmitted to a remote site, it looks in that site's **.Sequence** file, increases the sequence number by one, uses that number to name the file, and writes the incremented sequence number back into the site's **.Sequence** file.

/usr/spool/uucp/LCK..*
Finally, files that begin with the string **LCK..** are lock files. UUCP (and many other COHERENT programs) use them to lock devices, to ensure that only one program can access a device at a time. Each lock file contains the process identifier of the process that has locked that device, but different programs use different conventions in naming lock files.

Programs that log users into your system lock console and terminal devices. These programs use lock files whose names are built from the major-device number and the minor-device number of the device being locked For example, file **/usr/spool/uucppublic/LCK..2.1** locks the device with major number 2 and minor number 1 — that is, the color virtual-console device **/dev/color1**. Looking into file **LCK..2.1**, we see the number 6836; and when we use the command **ps -alx** to look for a process with this identifier, we see the following

```
color1   6836     1     fred  133 6001 w           ksh
color1   8923  6836     fred  204 6001 S  ttywait  me
```

That is, user **fred** has logged into this system via device **/dev/color1** and invoked a shell that has process identifier **6836**.

Second, when UUCP opens a port to dial out, it creates a lock file whose name includes the name of the port on which it is dialing. For example, if UUCP is dialing out via port **/dev/com3fl**, it creates file **LCK..com3fl** in **/usr/spool/uucp**. This helps to stop two UUCP process from each trying to open the same port at the same time.

Finally, when UUCP dials a given remote site, it creates a lock file for that site. For example, if UUCP dials site **mwc**, it creates lock file **LCK..mwc** in directory **/usr/spool/uucp**. This help to prevent two different UUCP processes from attempting to dial the same site at the same time.

This concludes our discussion of UUCP's files and directories. For more information, see the Lexicon entries **config**, **dial**, **port**, and **sys**.

Permissions

The following gives the correct permissions and ownership for the files that comprise the UUCP system:

```
-rw-------  uucp   uucp  /usr/lib/uucp/dial
-rw-------  uucp   uucp  /usr/lib/uucp/port
-rw-------  uucp   uucp  /usr/lib/uucp/sys
-r-sr-xr-x  uucp   root  /usr/lib/uucp/uucico
-rwxr-xr-x  uucp   root  /usr/lib/uucp/uuconv
-r-s--s--x  root   root  /usr/lib/uucp/uumkdir
-r-xr-xr-x  uucp   uucp  /usr/lib/uucp/uumvlog
-r-xr-xr-x  uucp   uucp  /usr/lib/uucp/uurmlock
-r-xr-xr-x  root   root  /usr/lib/uucp/uusched
-r-s--s--x  uucp   uucp  /usr/lib/uucp/uutouch
-r-x------  uucp   uucp  /usr/lib/uucp/uutry
-r-sr-xr-x  uucp   root  /usr/lib/uucp/uuxqt
-r-s--s--x  uucp   uucp  /usr/bin/uucheck
-r-sr-xr-x  uucp   root  /usr/bin/uucp
-r-x--x--x  bin    bin   /usr/bin/uudecode
-r-x--x--x  bin    bin   /usr/bin/uuencode
-r-s--s---  uucp   uucp  /usr/bin/uuinstall
-rwxr-xr-x  root   root  /usr/bin/uulog
-r-sr-xr-x  uucp   root  /usr/bin/uuname
-rwxr-xr-x  root   root  /usr/bin/uupick
-r-sr-xr-x  uucp   root  /usr/bin/uustat
-r-xr-xr-x  root   root  /usr/bin/uuto
-r-sr-xr-x  uucp   root  /usr/bin/uux
```

Permissions should be set properly by COHERENT when you installed it on your computer. However, if problems arise with UUCP, be sure to check that permissions are correct. If permissions have somehow been reset incorrectly, UUCP will not work because much of its work depends upon its being able to create and delete files in certain restricted directories.

Should a file's permissions be "stepped on" for whatever reason, use the command **chmod** to restore them. Likewise, should the group or user who "owns" a file or directory be changed for whatever reason, you (or, to be more exact, the superuser **root** can use the commands **chgrp** and **chown** to restore proper ownership. For details on how to use these commands, see their entry in the Lexicon.

Debugging UUCP Problems

For information how to debug and solve common problems with UUCP, see the tutorial on UUCP that appears in the front half of this manual.

See Also

asy, commands, config, cu, dial, domain, modem, mwcbbs, port, sys, terminal, uuchk, uucico, uuconv, uucp, uucpname, uudecode, uuencode, uuinstall, uulog, uumkdir, uumvlog, uuname, uupick, uurmlock, uusched, uustat, uuto, uutouch, uutry, uux, uuxqt
UUCP, Remote Communications Utility, tutorial

Notes

The Lexicon entry **mail** gives directions on how to send mail to users on popular commercial networks.

For information on how to hook up a Trailblazer modem to run UUCP, see the Lexicon entry for **modem**.

The COHERENT implementation of UUCP was written by Ian Lance Taylor (ian@airs.com). It was ported to COHERENT by Robert Chalmers (earth@nanguo.cstpl.com.au). For information on copyright and availablity of source code, see the documentation included in file **/usr/src/alien/uudoc.tar.Z**.

uucp — Command

Spool files for transmission to other systems
uucp [-cCdfmr] [-n*user***] [-x***activity***]** *source ... dest*

The command **uucp** spools every file *source* for copying to *dest*. *source* and *dest* can specify a remote system.

uucp recognizes the following options:

-C Copy the source file into spool directory; same as option **-p**. This is the default.

-c Do not copy the source file into spool directory; rather, use the file itself. The file must be readable both by yourself and by the daemon **uucico**. If the file is removed before **uucico** processes it, the transmission of the file will fail.

-d Create directories as required on the destination system. This is the default.

-f Do not create any directories on the remote system. If directories do not already exist, abort copying the file.

-g*grade*
Assign a grade (a single ASCII character, from '0' through 'z') to indicate the importance of the file being transmitted. The lower the ASCII value of *grade*, the more important the file; thus, '0' is the highest grade and 'z' the lowest.

-I *file*
Read the configuration for the remote system from *file* instead of from **/usr/lib/uucp/sys**, which is the default.

-j Report the job's process identifier. If you wish, you can use this identifier with the command **uustat** to kill the job.

-m Send mail to requester when the file is sent; report whether the job was executed successfully.

-n*user*
Send mail to *user* on destination system when the file is received. *user* can contain a path. Note that *user* is relative to the destination machine, not to originating machine or to any intervening machine. For example, consider the command:

```
uucp -nlepanto!fred myfile joe!/tmp
```

Here, you are copying **myfile** from your machine into directory **/tmp** on machine **joe**, and sending notification to user **fred** on machine **lepanto**. If, however, machine **joe** does not know how to address machine **lepanto**, then **fred** will never be notified of the transfer.

-p Copy the source file into spool directory; same as **-C**. This is the default.

-R Copy directories recursively.

-r Spool transfer request, but do not initiate **uucico**.

-s *file*
Write status upon completion of job into *file*.

-u *user*
Set user name to *user*.

-W Do not prefix the file's name with the name of the current directory.

-x*activity*
Log a given *activity*. These logs can help you debug problems with UUCP. **uucp** recognizes the following activities:

abnormal
Log abnormal events that occur while spooling a file for copying.

config Log problems that arise with reading or interpreting the configuration files **dial**, **port**, and **sys**.

execute
Log each step **uucp** takes as it executes.

spooldir
Log activity that involved the UUCP spooling directory **/usr/spool/uucp** and its subdirectories.

uucp writes its logging information into file **/usr/spool/uucp/.Admin/audit.local**.

Examples

The first example copies file **foo** to directory **/bar** on system **george**:

```
uucp foo george!/bar
```

The next example copies file **/foo** from system **george** into directory **/tmp** on your system:

```
uucp george!/foo /tmp
```

The next example copies file **/foo** from system **george** into file or directory **/bar** on system **ivan**:

```
uucp george!/foo ivan!/bar
```

Note that this assumes your system can talk to both **george** and **ivan** and that your system has permission to read file **/foo** on system **george** as well as to write file **/bar** on system **ivan**.

The next example downloads files **/foo** and **/bar** from remote systems **ivan** and **george** into directory **/tmp** on your system:

```
uucp ivan!/foo george!/bar /tmp
```

The last example downloads file **foo** from system **ivan** via system **george**:

```
uucp george!ivan!foo
```

For an example of using the command **find** with **uucp** to spool files automatically, see the entry for **find**.

Files

/usr/lib/uucp/sys — List of reachable systems
/usr/spool/uucp/.Log/*/*sitename*— **uucp** activities log files for *sitename*
/usr/spool/uucp/*sitename*— Spool directory for work

See Also

commands, mail, uucico, UUCP, uudecode, uuencode, uutouch, uuxqt

Notes

uucp was written by Ian Lance Taylor (ian@airs.com).

uucpname — System Administration

Set the system's UUCP name
/etc/uucpname

The file **/etc/uucpname** sets the name by which your system is known to all other system with which it communicates via UUCP. To rename your system, simply change the contents of this file.

The contents of **/etc/uucpname** is, in effect, your system's *nom de plume*. It should be unique (or as unique as possible), easily remembered, and preferably in good taste. Examples of existing systems include **lepanto**, **smiles**, and **stevesf**. You should avoid names taken from popular culture, such as **calvin**, **hobbes**, or **terminator**: many other people have already used them.

Note that system names must obey the following rules:

- UUCP names must be no more than 14 characters long.
- Names must consist of letters and numbers. No punctuation marks, white space, control characters, or diacritic marks are permitted.

- Each name must begin with a letter.

If you wish for your system to communicate with other systems in the world-wide UUCP network, you should follow the following restrictions as well:

- UUCP names should be contain no more than seven characters.
- They should use only lower-case letters and digits.

If you are connecting to other machines we recommend that you acquire a registered Fully Qualified Domain Name. Every person in the United States may register in the **.us** domain. Send mail to **us-domain-request@venera.isi.edu** for information on this. If you wish to create your own domain (e.g., **mwc.com)**, send mail to **info-request@uunet.uu.net** for information on this.

See Also

Administering COHERENT, domain, UUCP

Notes

Only the superuser **root** can edit **/etc/uucpname**.

uudecode — Command

Decode a binary file sent from a remote system
uudecode [*file* **]**

uudecode takes a **file** encoded by **uuencode** and translates it back to binary. Any leading and trailing lines added by **uucp** are discarded.

If the *file* is not specified, standard input is read.

Example

Consider the file **tmp** consisting of:

```
begin 644 sys
M5&AE('%U:6-K(&)R;W=N(&9O>"!J=6UP<R!O=F5R('1H92!L87IY(&1O9RX*

end
```

Note that the third line is a space followed by a newline. To decode it, type:

```
uudecode tmp
```

The output contained in file **sys** will be:

```
The quick brown fox jumps over the lazy dog.
```

See Also

commands, UUCP, uucp, uuencode

Notes

The user on the remote system must be able to write the file.

uuencode — Command

Encode a binary file for transmission
uuencode [*source* **]** *file_label* **[<** *source* **] >** *output*

uuencode prepares a file for transmission to a remote destination via **uucp.** It takes binary input and produces an encoded version, consisting of printable ASCII characters, on standard output, which may be redirected or piped to **uucp**.

If *source* is not specified, **uuencode** reads the standard input and writes to the standard output. If, however, *source* is specified, **uuencode** its permissions into the **uuencode**'d file. *file_label* is the name that **uudecode** gives to the file when it is decoded.

uuencode is chiefly used for mail. You cannot mail a binary file, but you can mail a **uuencode**'d binary. The standard way to mail a binary is to compress it, **uunecode** it, split it into pieces less than 50 kilobytes each, then mail each piece.

The format of the encoded file is as follows:

1. A *header* line starting with the characters **begin** followed by a space. This is followed by the mode of the file in octal and the name of the output file specified on the command line. (For details, see the Lexicon entry for **chmod**). These last two fields are also separated by a space. The mode and the system name can be changed by directing the output into a file and editing it.
2. The *body* of the file, consisting of a number of lines, each no more than 62 characters long, including a newline character. Each line starts with a character count written as a single ASCII character, representing an integer value from 0 (octal 40) to 63 (octal 135) giving the number of characters in the rest of the line. This is followed by the encoded characters and a newline. The last line of the body is a line consisting of an ASCII space (octal 40).
3. A *trailer*, which consists of the string **end** on a line by itself.

The encoding is done by taking three bytes and storing them in four characters, six bits per character. Each six bits is converted to a printable character by adding 0x20 to it.

Example

Consider the file **tmp**, which consists of the line:

```
The quick brown fox jumps over the lazy dog.
```

To record it in file **tmp.send**, type:

```
uuencode tmp < tmp > tmp.send
```

The output is:

```
begin 644 tmp
M5&AE('%U:6-K(&)R;W=N(&9O>"!J=6UP<R!O=F5R('1H92!L87IY(&1O9RX*

end
```

Note that the third line consists of a space followed by a newline.

See Also

commands, UUCP, uucp, uudecode

Notes

uuencode expands a file by more than one third, which thus increases transmission time. This can be a factor when sending large files.

uuinstall — Command

Install or modify UUCP
uuinstall

uuinstall help you to install UUCP on your COHERENT system. It uses screen templates, help lines, and prompts to walk you through the installation of devices, remote systems, site names, domains, and permissions. For a detailed description of its use, see the tutorial on UUCP in the front of this manual.

See Also

commands, UUCP

Notes

Only the superuser **root** can execute **uuinstall**.

On some terminals, the arrow keys do not move the cursor. In this case, you can use **vi**-style cursor-movement keys:

H Move the cursor left
K Move the cursor up
L Move the cursor right
J Move the cursor down

uulog — Command

Read a UUCP log
uulog [-f*system***] [-s***system***] [-n***number***] [-x]**

uulog copies the last part of the file **/usr/spool/uucp/.Log/uucico/***system* to see what **uucico** has done recently. *system* names the remote system whose logfile will be examined. If it is not specified, logfiles for all systems are displayed.

uulog recognizes the following options:

-f*system*
: Similar to the command **tail -f**: this forces **uulog** to display UUCP activity as it is written into the log file for *system*, until you interrupt it by typing **<ctrl-C>**.

-n*number*
: Display only *number* lines from the end of the log.

-s*system*
: Display the log file for *system*.

-x : Display the log file for the command **uuxqt** rather than **uucico**.

Files

/usr/spool/uucp/.Log/uucico/*system*— **uucico** log file for *system*
/usr/spool/uucp/.Log/uuxqt/*system*— **uuxqt** log file for *system*

See Also

commands, UUCP

uumkdir — Command

Create UUCP directories
/usr/lib/uucp/uumkdir [-m *mode* **] [-p]** *directory ...*

The command **uumkdir** creates each *directory* named on its command line. Option **-m** sets the permissions on the newly created directory to *mode*, which must be a three-numeral octal number.

See Also

commands, UUCP

uumvlog — Command

Archive UUCP log files
uumvlog *days*

uumvlog copies all UUCP log files into backup files, named for their respective commands and the date upon which the backup was performed. *days* gives the number of days for which backup files should be kept: if a backup file is more than *days* days old, then **uumvlog** will delete it.

This command should be run by **cron**, because the UUCP log files can threaten to exhaust available file space on a small system unless they are chopped back daily.

Files

/usr/spool/uucp/.Log/*command***/***system*— UUCP log files

See Also

commands, cron, UUCP

Notes

uumvlog manages the log files under directory **/usr/spool/uucp/.Log**. However, it does not touch the files in **/usr/spool/uucp/.Admin**. These can grow quite large if unattended. At present, you must manage these files by hand.

uuname — Command

List UUCP names of known systems
uuname [-l]

The command **uuname** lists the names of all systems reachable directly by UUCP. It does so by reading the names of the systems defined in file **/usr/lib/uucp/sys**, plus the name of your local system as set in file **/etc/uucpname**. Command-line option **-l** prints the name of your local system only.

Files

/etc/uucpname — Name of local system
/usr/lib/uucp/sys — Site and remote login data

See Also

commands, UUCP

uupick — Command

Pick up a file uploaded from a remote system
/usr/bin/uupick [-s *system***] [-I** *file***] [-x** *event***]** *file ...*

The command **uupick** lets you "pick up" each *file* that has been uploaded to your system via UUCP. It moves the file into your current directory from whence it was copied on your system. It usually used to acquire files that had been sent to your system via the script **uuto**.

uupick recognizes the following command-line options:

--help Print a help message, and exit.

-I *file*
--config *file*
Read configuration information from *file* instead of from the default configuration file.

-s *system*
--system *system*
"Pick up" only files uploaded from *system*.

-v
--version
Print the version of **uupick** that you are running, and exit.

-x *activity*

-x*type***[,***type***,...,type]**
-X*type***[,***type***,...,type]**
Log a given *activity*. **uupick** recognizes the following activities:

abnormal	**chat**	**config**
execute	**handshake**	**incoming**
outgoing	**port**	**proto**
spooldir	**uucp-proto**	

One **-x** option can name multiple activities, separated by commas. A **uupick** command line can contain more than **-x** option.

See Also

commands, UUCP, uuto

Notes

uupick was written by Ian Taylor (ian@airs.com).

uurmlock — Command

Remove UUCP lock files
uurmlock

UUCP uses a system of lock files to ensure that sites are polled in an orderly manner. It creates a lock file named after the site being polled, to prevent more than one invocation of **uucico** or another UUCP command from polling

the same site at the same time. On occasion, if UUCP fails or crashes, it will neglect to clean up its lock files, thus preventing itself from polling the locked sites.

The command **uurmlock** removes all lock files. You should run this if you suspect that UUCP has died in a disorderly manner and has left lock files lying around unattended.

Before you run **uurmlock**, examine the output of the command **ps** to ensure that no UUCP command is running at the moment (and so has legitimately locked a site).

Files

/usr/spool/uucp/LCK.* — UUCP lock files

See Also

commands, UUCP

Notes

Only the superuser **root** can run **uurmlock**.

Note that **uurmlock** removes all **.LCK** files from **/usr/spool/uucp**. Not all of these are used by UUCP; however, this behavior is necessary to remain compatible with UNIX, and is almost always benign.

uusched — Command

Call all systems that have jobs waiting for them
/usr/lib/uucp/uusched

The one-line script **uusched** invokes command **uucico** with its option **-r1**, which tells **uucico** to call all systems that have jobs queued up for them.

See Also

commands, uucico, UUCP

uustat — Command

UUCP status inquiry and control
uustat [-eKiMNQ] [-B *lines***] [-cC** *command***] [-o** *hours***] [-sS** *system***] [-uU** *user***] [-y** *hours***]**
uustat -a
uustat [-k *jobid***] [-r** *jobid***]**
uustat -m
uustat -p
uustat -q

The command **uustat** displays status information about the UUCP system. You can also use it to cancel or rejuvinate requests made by via commands **uucp** or **uux**.

By default, **uustat** displays every job queued by the user who invokes this command. If the command line includes any of the options **-a**, **-e**, **-s**, **-S**, **-u**, **-U**, **-c**, **-C**, **-o**, or **-y**, then **uustat** displays information about all of the jobs that match given specifications.

The option **-K** can be used to kill a selected group of jobs, such as all jobs more than seven days old.

Command-line Options

uustat recognizes the following command-line options:

-a List all queued requests to transfer files.

-C *command*
List all jobs except those that request the execution of *command*. If *command* is **ALL**, list all jobs that simply request a file transfer (as opposed to requesting the execution of some command). You cannot use this option with the option **-c**. A **uustat** command can hold more than one **-C** option.

-c *command*
List every job that requests the execution of *command*. If *command* is **ALL**, **uustat** lists all jobs that request the execution of a command (as opposed to simply requesting a file transfer). A **uustat** command can hold more than one **-c** option.

-e List queued requests to execute a program on a remote system, rather than queued requests to transfer files. Queued execution requests are processed by **uuxqt** rather than **uucico**. A queued execution request may await a file from a remote system. These requests are created by an invocation of the command **uux**.

-I *file* Read configuration information from *file* instead of from the default file **/usr/lib/uucp/sys**.

-i For each listed job, prompt whether to kill the job. If the first character of the input line is **y** or **Y**, the job will be killed.

-K Kill each listed job without prompting for permission. This can be used in a script to clean up obsolete jobs automatically.

-k *jobid* Kill the job with the identifier *jobid*. A job's identifier is shown by the default output format, as well as by the commands **uucp** or **uux** when invoked with option **-j**. A job may only be killed only by the user who created the job, the UUCP administrator, or the superuser **root**. You can use the option **-k** more than once on a **uustat** command line, to kill several jobs simultaneously.

-M For each listed job, send mail to the UUCP administrator. If the job is killed (due to **-K** or **-i** with an affirmative response), the mail will indicate that. A comment specified by the **-W** option may be included. If the job is an execution, the initial portion of its standard input will be included in the mail message; the number of lines to include may be set with the **-B** option (the default is 100). If the standard input contains null characters, it is assumed to be a binary file and is not included.

-m Display the status of conversations for all remote systems.

-N For each listed job, send mail to the user who requested the job. The mail is identical to that sent by the option **-M**.

-o *hours*
List all jobs that have been queued longer than *hours*.

-p Display the status of all processes holding UUCP locks on systems or ports.

-Q Work quietly: Do not list the job, just perform the actions indicated by the options **-i**, **-K**, **-M**, or **-N**.

-q Display the status of commands, executions, and conversations for all remote systems for which commands or executions are queued.

-r *jobid* Rejuvinate the job with job identifier *jobid*. This marks the job as having been invoked at the current time; which, in turn, affects the output of the options **-o** or **-y** and preserves the job from any automated cleanup daemon. The job identifier is shown by the default output format, as well as by the commands **uucp** and **uux** when invoked with option **-j**. A job may only be rejuvenated by the user who created the job, by the UUCP administrator, or the superuser **root**. You can use the option **-r** more than once on a **uustat** command line, to rejuvinate several jobs simultaneously.

-S *system*
List all jobs except the ones queued for *system*. You cannot use this option with the option **-s**. A **uustat** command can hold more than one **-S** option.

-s *system*
List every job queued for *system*. A **uustat** command can hold more than one **-s** option.

-U *user* List all jobs except the ones queued for *user*. You cannot use this option with the option **-u**. A **uustat** command can hold more than one **-U** option.

-u *user* List every job queued for *user*. A **uustat** command can hold more than one **-u** option.

-W Specify a comment to be included in mail sent with the **-M** or **-N** options.

-x *type* Turn on particular debugging types. The following types are recognized:

abnormal	**chat**	**config**
execute	**handshake**	**incoming**
outgoing	**port**	**proto**
spooldir	**uucp-proto**	

Only **abnormal**, **config**, **spooldir**, and **execute** are meaningful for **uustat**.

Multiple types may be given, separated by commas, and the **-x** option can appear multiple times on the **uustat** command line. A number may also be given, which will turn on that many types from the foregoing list; for example, **-x 2** is equivalent to **-x abnormal,chat**.

-y *hours*
List all jobs that have been queued less than *hours*.

Examples

The first example displays the status of all jobs:

```
uustat -a
```

The output has the format:

jobid system user queue-date command (size)

The job identifier may be passed to the options **-k** or **-r**. The size indicates how much data is to be transferred to the remote system, and is absent for a file-receive request. The options **-s**, **-S**, **-u**, **-U**, **-c**, **-C**, **-o**, and **-y** may be used to control which jobs are listed.

The next example displays the status of queued execution requests:

```
uustat -e
```

The output has the format:

system requestor queue-date command

The options **-s, -S, -u, -U, -c, -C, -o,** and **-y** can be used to control which requests are listed.

The next example displays the status for all systems with queued commands:

```
uustat -q
```

This displays the system, the number of commands queued for it, the age of the oldest queued command, the number of queued local executions, the age of the oldest queued execution, the date of the last conversation, and the status of that conversation.

The next example displays conversation status for all remote systems:

```
uustat -m
```

The output gives the system, the date of the last conversation, and the status of that conversation. If the last conversation failed, **uustat** indicates how many attempts have been made to call the system. If the retry period is preventing calls to that system, **uustat** also displays the time when the next call will be permitted.

The next example displays the status of all processes that hold UUCP locks:

```
uustat -p
```

The output is exactly the same as that of the command **ps** for each process that holds a lock.

The next example kills all **rmail** commands that have been queued up waiting for delivery for over one week (168 hours).

```
uustat -c rmail -o 168 -K -Q -M -N -W"Queued for over 1 week"
```

uustat sends mail both to the UUCP administrator and to the user who requested the **rmail** execution. The mail message includes the string given by the **-W** option. The option **-9** prevents any of the jobs from being listed on the terminal, so any output from the program will be error messages.

Files

/usr/lib/uucp/config — Configuration file.
/usr/spool/uucp — UUCP spool directory.

See Also

commands, ps, rmail, uucico, UUCP, uucp, uux, uuxqt

Notes

uustat was written by Ian Lance Taylor (ian@airs.com).

LEXICON

uuto — Command

Send a file to a remote system
/usr/bin/uuto *file ... file remote_system*

The one-line script **uuto** invokes the command **uucp** to send each *file* to *remote_system*.

See Also

commands, UUCP, uucp

uutouch — Command

Touch a file to trigger UUCP poll
uutouch *system*

The command **uutouch** creates an empty control file for *system* in the directory **/usr/spool/uucp/***system*. This forces UUCP to poll *system* when **uucico** is called with the option **-sall**. If the empty file for *system* aready exists, it is left alone.

There are three types of files in the spool directory **/usr/spool/uucp/***system*:

C. Command file.

D. Data file.

X. Execute file.

Example

A typical usage is to put the following line into the **cron** file **/usr/spool/cron/crontabs/uucp**:

```
0 7 * * * /usr/lib/uucp/uutouch george
```

This forces UUCP to schedule a poll to the remote system **george** at 7 AM local time. The actual poll take place when **uucico** is started.

Files

/usr/spool/uucp/*sitename* — Directory for uucp work files

See Also

commands, cron, uucico, UUCP

uutry — Command

Debugging script for UUCP
uutry *remotesystem* **[-x***debuglevel***]**"

The command **uutry** is a script that invokes **uucico** to contact *remotesystem*, and records all debugging information that **uucico** generates. **uutry** redirects the debugging information into file **audit.local** in directory **/usr/spool/uucp/.Admin**. If such a file already exists, **uutry** renames it **audit.OLD** before it invokes **uucico**.

The option argument **-x** sets the debugging level to *debuglevel*. This is a number from zero through nine; for information on what the debugging level means, see the Lexicon entry for the command **uucico**. The default level is five.

See Also

commands, UUCP

Notes

For security reasons **uutry** can be run only by the superuser **root**.

uux — Command

Execute a command on a remote system
uux [-a *user***] [-rnpz]** *command-string*

The command **uux** spools *command-string* for execution on a remote system. Usually, it is invoked by software systems, in particular the mail system, to request that work be performed on a remote system. However, you can also invoke **uux** by hand to execute a task on a remote system.

For security reasons, you can execute on the remote system only the commands that the remote system permits explicitly. These commands are named in the entry for your system in the remote system's copy of **/usr/lib/uucp/sys**.

If all permissions are in order, **uux** creates a file with the prefix **X.** in the remote system's directory **/usr/spool/uucp/***yoursystem*, where *yoursystem* is the name by which the remote system knows your system. This file is then executed by the remote system's copy of the command **uuxqt**.

command-string consists of a command name followed by zero or more arguments. Both the command's name and the arguments may be prefixed by a system name (sitename) and an exclamation mark. Note that all special characters must be escaped or enclosed in quotation marks to avoid being processed by your system's shell.

For example, the simplest form of the **uux** command is:

```
uux host!command arg0 ... argN
```

where *host* is the name of the remote system being contacted, as defined in file **/usr/lib/uucp/sys**, *command* is the name of the command to execute on the remote system, and *arg0* through *argN* are the arguments to *command*.

If an argument names a file, that file can reside on the remote system, on your system, or on some third system. For example, the command

```
uux widget!lp /usr/sally/herfile
```

asks site **widget** to print its own file **/usr/sally/herfile**. On the other hand, the command

```
uux widget!lp !$HOME/myfile
```

requests that site **widget** print on its line printer the file **myfile** from your home directory on your home system. Note that the '!' that prefixes **myfile** is shorthand for the name of your system. Finally, the command

```
uux widget!lp lepanto!/usr/fred/hisfile
```

requests that system **widget** print file **/usr/fred/hisfile**, which resides on the third site **lepanto**. If **widget** does not know how to contact site **lepanto**, the command fails.

If you wish, you can embed the shell operators '<', '>', ';', or '|' within a **uux** command. This lets you construct a more powerful command than you could do otherwise. Commands that contain these operators must be quoted, to ensure that your shell does not interpret them. For example, the command

```
uux "widget!pr /usr/sally/herfile > lepanto!~/fred/hisfile"
```

tells **uux** to use **pr** to format its file **/usr/sally/herfile**, and write the output into file **/usr/spool/uucppublic/fred/hisfile** on site **lepanto**. (Note that the tilde '~', as always, is a synonym for the home directory of the user that is executing the command; and a **uux** command is always executed by user **uucp** whose home directory is always **/usr/spool/uucppublic**.) Again, the command fails if you do not have appropriate permissions on **widget** or if **widget** does not have appropriate permissions on **lepanto**.

The operator '-' lets you use the standard input when constructing a **uux** command. For example, the command

```
who | uux - widget!lp
```

executes the **who** command on your system, pipes the output to **uux**, and tells **uux** to invoke the command **lp** on remote system **widget** to print the list of users on your system.

uux attempts to transfer any needed input files to the system that will be executing the requested command. You must enclose in parentheses any output files generated by *command*, to distinguish them from the names of input file.

Command-line Options

uux recognizes the following options:

-a *address*
: Report the status of the job to *address*.

-C
: Copy local files to the spool directory.

-c Do not copy local files to the spool directory. This is the default. If the files are removed from their local directory before **uucico** processes them, the copy fails. The files must be readable by the **uucico** as well as the by the user who invokes **uux**.

-g *grade*
Set the grade of the file-transfer command. *grade* is a single ASCII character, from '0' to 'z'; the lower the ASCII value of *grade*, the more important the files.

-I *file* Read configuration information from *file* instead of from the default file **/usr/lib/uucp/sys**.

-j Print job identifiers on the standard output. **uux** creates a job identifier for each file-copying operation required to perform the operation. To cancel the copying of a file, pass the job identifier to the **uustat** with its option **-k**.

-l Link local files into the spool directory. If a file cannot be linked because it is on a different device, it is copied unless the **-c** option also appears (in other words, use of **-l** switches the default from **-c** to **-C**). If the files are changed before **uucico** processes them, the changed versions will be used. The files must be readable by the **uucico** as well as by the user who invoked **uux**.

-n Do not send mail about the status of a job, even if it fails. The default is to send mail to the requester should the command fail.

-
-p Read the standard-input device and pipe what is read into the command to be executed.

-r Queue the **uux** request but do not invoke **uucico** to perform the transfer. The default is to initiate **uucico**.

-x *event*
Log each *event* in the execution of **uux**, where *event* is one of the following values: **abnormal**, **config**, **spooldir**, or **execute**. A **-x** option can hold multiple events, each separated by commas; and a **uux** command line can hold more than **-x** option.

-z Notify requester should *command-string* fail.

Examples

The following script prints files on a remote system. The files named on the command line are sent unprocessed to system **prnsrvr** to be printed through that system's version of command **lp**. Option **-r** tells **uux** not to invoke **uucico** immediately, but merely spool the request for execution later.

```
for i in $*
do
      uux -r prnsrvr!lp !$i
done
```

Please note that the '!' that prefixes string "!$i" indicates that the file to be printed resides in the current directory on your home system.

The next example copies file **/foo** from system **george** and file **/bar** from system **norm** to your system and then invokes command **cmp** to compare their contents. It writes the results of the comparison into file **/tmp/cmp.results** on your local system:

```
uux -z "!cmp -l george!/foo norm!/bar >/tmp/cmp.results"
```

This command assumes that your system can talk to both **george** and **norm**, and that your system has permission to read file **/foo** on system **george** and file **/bar** on system **norm**. Option **-z** tells **uux** to send you mail when it has successfully completed the job.

The last example compiles file **mycode.c** on system **cserver**. The command redirects all of the compiler's error messages into file **/tmp/errors** on your local system:

```
uux 'cserver!cc -O -o (!mycode) !mycode.c > !/tmp/errors'
```

Note that the name of the output file **!mycode** is enclosed within parentheses. This is to protect the '!' from being interpreted by **uux**; it will be interpreted by **uuxqt** on the remote system.

See Also

commands, UUCP, uuxqt

Notes

You cannot pipe the output from a command on one system into a command on another. If *command-line* consists of several commands that are connected by pipe characters '|', only the first can be prefixed by a system name and '!'; every other command within the pipeline will be executed on the system named by the first command. For example, consider the command:

```
uux "mwc!wrap -w80 -t4 < !myfile.c | prps | lp"
```

This command passes file **myfile.c** from the current directory on your current system to command **wrap** on system **mwc** for processing; then pipes the output of **wrap** into **prps** on system **mwc** for transformation into PostScript; and then pipes the output of **prps** into **lp**, again on system **mwc**, for printing. If you embed a '!' within the subsequent commands of a pipeline, **uux** will expand it into something quite unexpected (and probably unwelcome).

It is not a good idea to use the metacharacter '*' within *command-line*. The odds that it will be expanded into what you want are very small.

Every command that you spool with **uux** is executed within a special execution directory on the remote system. (Under COHERENT, this directory is **/usr/spool/uucp/.Xqtdir**; it may vary on other systems.) Before it executes the command, UUCP copies into that special directory each file that the command names, unless that file already resides on the system within which the command is being executed. For this reason, each file named in a **uux** command must be unique, regardless of its full path name. For example, the following command will not work:

```
uux  "marian!diff fred!/x/testfile ivan!/y/testfile > !xyz.diff"
```

It fails because **uux** (or, to be more accurate, **uuxqt**) copies file **testfile** from system **fred** into its execution directory, then copy **testfile** from system **ivan** into the test directory. The second copied **testfile** overwrites the first, and thus the command **diff** fails.

uux was written by Ian Lance Taylor (ian@airs.com).

uuxqt — Command

Execute commands requested by a remote system
uuxqt

uuxqt reads files from directory **/usr/spool/uucp/***sitename*, and executes them. It recognizes the files to execute (as opposed to the files that simply contain data, such as mail messages) because they are prefixed with the string "X.". **uuxqt** executes only the programs for which the remote system has permission.

uuxqt is invoked by either **uucp** or **uucico**. It is not generally considered a user-callable program.

Command-line Options

uuxqt recognizes the following command-line options:

-c *command*
Only execute *command*; ignore requests to execute any other command. For example:

```
uuxqt -c rmail
```

-s *system*
Only execute requests originating from *system*.

-I *file* Read configuration information from *file* instead of from **/usr/lib/uucp/sys**.

-x *activity*
Log each *activity*; *activity* must be one following: **abnormal**, **config**, **spooldir**, and **execute**. An **-x** option can name more than one activity, with activities being separated by commas; and a **uuxqt** command-line can have more than one **-x** option.

Files

/usr/lib/uucp/config — Configuration file
/usr/spool/uucp/*sitename* — Directory for execute files
/usr/spool/uucp/Log — UUCP log file
/usr/spool/uucp/Debug — Debugging file

See Also

commands, uucico, UUCP, uucp, uux

Notes

uuxqt was written by Ian Lance Taylor (ian@airs.com).

va_arg() — Variable Arguments

Return pointer to next argument in argument list
#include <stdarg.h>
typename ***va_arg(***listptr**,** *typename***)**
va_list *listptr***,** *typename***;**

#include <varargs.h>
typename ***va_arg(***listptr**,** *typename***)**
va_list *listptr***,** *typename***;**

va_arg() returns a pointer to the next argument in an argument list. It can be used with functions that take a variable number of arguments, such as **printf()** or **scanf()**, to help write such functions portably. It is always used with **va_end()** and **va_start()** within a function that takes a variable number of arguments.

listptr is of type **va_list**, which is defined in the headers **<stdarg.h>** and **<varargs.h>**. This object must first be initialized by the macro **va_start()**.

typename is the name of the type for which **va_arg()** is to return a pointer. For example, if you wish **va_arg()** to return a pointer to an integer, *typename* should be of type **int**.

va_arg() can only handle "standard" data types, i.e., those data types that can be transformed to pointers by appending an asterisk '*'.

Example

For an example of this macro, see the entry for **stdarg.h**.

See Also

stdarg.h, varargs.h
ANSI Standard, §7.8.1.2

Notes

There are two different versions of **va_arg()**: the ANSI version, which is defined in **<stdarg.h>**; and the UNIX version, which is defined in **<varargs.h>**. For a discussion of how these implementations differ, see the entry for **stdarg.h**.

If there is no next argument for **va_arg()** to handle, or if *typename* is incorrect, then the behavior of **va_arg()** is undefined.

The ANSI Standard demands that **va_arg()** be implemented only as a macro. If its macro definition is suppressed within a program, its behavior is undefined.

va_end() — Variable Arguments

Tidy up after traversal of argument list
#include <stdarg.h>
void va_end(*listptr***)**
va_list *listptr***;**

#include <varargs.h>
void va_end(*listptr***)**
va_list *listptr***;**

va_end() helps to tidy up a function after it has traversed the argument list for a function that takes a variable number of arguments. It can be used with functions that take a variable number of arguments, such as **printf()** or **scanf()**, to help write such functions portably. It should be used with the routines **va_arg()** and **va_start()** from within a function that takes a variable number of arguments.

listptr is of type **va_list()**, which is declared in header **stdarg.h**. *listptr* must first have been initialized by macro **va_start**.

Example

For an example of this function, see the entry for **stdarg.h**.

See Also

stdarg.h, varargs.h
ANSI Standard, §7.8.1.3

Notes

There are two different versions of **va_end()**: the ANSI version, which is defined in **<stdarg.h>**; and the UNIX version, which is defined in **<varargs.h>**. For a discussion of how these implementations differ, see the entry for **stdarg.h**.

If **va_list()** is not initialized by **va_start()**, or if **va_end()** is not called before a function with variable arguments exits, then the behavior of **va_end()** is undefined.

va_start() — Variable Arguments

Point to beginning of argument list
#include <varargs.h>
void va_start(*listptr***)**
va_list *listptr***;**

#include <stdarg.h>
void va_start(*listptr, rightparm***)**
va_list *listptr, type rightparm***;**

va_start() is a macro that points to the beginning of a list of arguments. It can be used with functions that take a variable number of arguments, such as **printf()** or **scanf()**, to help implement them portably. It is always used with **va_arg()** and **va_end()** from within a function that takes a variable number of arguments.

This macro is defined in two different header files, **<stdarg.h>** and **<varargs.h>**. The former header file is the creation of the ANSI C committee, whereas the latter originates with UNIX System V. In both implementations, the first argument is *listptr*, which is of type **va_list**.

The implementation in **<stdarg.h>** (ANSI) adds a second argument, *rightparm*, which is the rightmost parameter preceding the variable arguments in the function's parameter list. Undefined behavior results if any of the following conditions apply to *rightparm*: if it has storage class **register**; if it has a function type or an array type; or if its type is not compatible with the type that results from the default argument promotions.

Example

For an example of this macro, see the entry for **stdarg.h**.

See Also

stdarg.h, varargs.h
ANSI Standard, §7.8.1.1

Notes

For a discussion of how the **<stdarg.h>** and **<varargs.h>** implementations of the variable-argument routines differ, **stdarg.h**.

The ANSI Standard demands that **va_start()** be implemented only as a macro. If the macro definition of **va_start()** is suppressed within a program, the behavior is undefined (and probably unwelcome).

varargs.h — Header File

Declare/define routines for variable arguments
#include <varargs.h>

The header file **<varargs.h>** prototypes and defines the routines used to manage variable arguments. These routines are modelled after those used in UNIX System V. The routines in **varargs.h** were designed to give a C compiler a semi-rational way of dealing with functions (e.g., **printf()**) that can take a variable number of arguments. In brief, these routines consist of the variable-list **typedef va_list**, the parameter declaration **va_dcl**, and the three macros **va_start()**, **va_arg()**, and **va_end()**. The macros respectively start the argument list, fetch the

next member, and end the argument list.

See Also

header files, stdarg.h

Notes

These routines are also implemented in the header file **<stdarg.h>**, which is described in the ANSI Standard. For details on how these implementations differ, see the entry for **stdarg.h**.

vfprintf() — STDIO Function (libc)

Print formatted text into stream
#include <stdarg.h>
#include <stdio.h>
int
vfprintf(*fp***,** *format***,** *arguments***)**
FILE*****fp*****; char** ****format*****; va_list** *arguments***;**

vfprintf() constructs a formatted string and writes it into the stream pointed to by *fp*. It translates integers, floating-point numbers, and strings into a variety of text formats. **vfprintf()** can handle a variable list of arguments of various types. It is roughly equivalent to **fprintf()**'s conversion specifier **%r**.

format points to a string that can contain text, character constants, and one or more *conversion specifications*. A conversion specification describes how to convert a particular data type into text. Each conversion specification is introduced with the percent sign '%'. (To print a literal percent sign, use the escape sequence '%%'.) See **printf()** for further discussion of the conversion specification, and for a table of the type specifiers that can be used with **vfprintf()**.

After *format* comes *arguments*. This is of type **va_list**, which is defined in the header file **stdarg.h**. It has been initialized by the macro **va_start()** and points to the base of the list of arguments used by **vfprintf()**. For more information, see the Lexicon entry for **va_arg()**.

arguments should access one argument for each conversion specification in *format*, of the type appropriate to its conversion specification. For example, if *format* contains conversion specifications for an **int**, a **long**, and a string, then *arguments* access three arguments, being, respectively, an **int**, a **long**, and a **char** *. *arguments* can take only the data types acceptable to the macro **va_arg()**, namely, the basic types that can be converted to pointers simply by adding a '*' after the type name. See **va_arg()** for more information on this point.

If there are fewer arguments than conversion specifications, then **vfprintf()**'s behavior is undefined (and probably unwelcome). If there are more, **vfprintf()** evaluates and then ignores every argument without a corresponding conversion specification. If an argument is not of the same type as its corresponding conversion specifier, then the behavior of **vfprintf()** is undefined. Thus, presenting an **int** where **vfprintf()** expects a **char** * may generate unwelcome results.

If it wrote the formatted string correctly, **vfprintf()** returns the number of characters written. Otherwise, it returns a negative number.

See Also

fprintf(), libc, printf(), sprintf(), vprintf(), vsprintf()
ANSI Standard, §7.9.6.7

Notes

vfprintf() can construct a string up to at least 509 characters long.

vi — Command

Clone of Berkeley-style screen editor
vi [*options* **] [+***cmd* **] [** *file1 ... file27* **]**

vi is a link to the editor **elvis**, which is a clone of the UNIX editors **ex** and **vi**. For details on how to run **vi**, see the entry for **elvis** in the Lexicon.

See Also

commands, ed, ex, elvis, me, view

Notes

elvis is copyright © 1990 by Steve Kirkendall, and was written by Steve Kirkendall (kirkenda@cs.pdx.edu), assisted by numerous volunteers. It is freely redistributable, subject to the restrictions noted in included documentation. Source code for **elvis** is available through the Mark Williams bulletin board, USENET, and numerous other outlets.

elvis is distributed as a service to COHERENT customers, as is. It is not supported by Mark Williams Company. *Caveat utilitor.*

vidattr() — terminfo Function

Set the terminal's video attributes
#include <curses.h>
vidattr(*newmode***)**
int *newmode***;**

COHERENT comes with a set of functions that let you use **terminfo** descriptions to manipulate a terminal. **vidattr()** sends one or more video attributes to the terminal opened by a call to **setupterm()**. *newmode* is any combination of the macros **A_STANDOUT**, **A_UNDERLINE**, **A_REVERSE**, **A_BLINK**, **A_DIM**, **A_BOLD**, **A_INVIS**, **A_PROTECT**, and **A_ALTCHARSET**, OR'd together. Their names are self-explanatory; all are defined in the header file **curses.h**.

See Also

curses.h, setupterm(), terminfo, vidputs()

vidputs() — terminfo Function

Write video attributes into a function
#include <curses.h>
vidputs(*newmode***,***outc***)**
int *newmode***;**
int (*outc***)();**

COHERENT comes with a set of functions that let you use **terminfo** descriptions to manipulate a terminal. **vidputs()** resets the video attributes of the terminal that had been opened by a call to **setupterm()**.

newmode is any combination of the macros **A_STANDOUT**, **A_UNDERLINE**, **A_REVERSE**, **A_BLINK**, **A_DIM**, **A_BOLD**, **A_INVIS**, **A_PROTECT**, and **A_ALTCHARSET**, OR'd together. Their names are self-explanatory; all are defined in the header file **curses.h**.

outc points to a function that takes a single character as an argument, e.g., **putchar()**.

The related function **vidattr()** resets video attributes without requiring a pointer to a function.

See Also

curses.h, setupterm(), terminfo, vidattr()

view — Command

Screen-oriented viewing utility
view *file1 ... file27*

view is a link to **elvis**, which is a clone of the UNIX **vi/ex** set of editors. Invoking **elvis** through this link forces it to operate solely in read-only mode, just as the UNIX **view** utility operates.

For information on how to use this version of **view**, see the Lexicon page for **elvis**.

See Also

commands, ed, elvis, ex, me, vi

Notes

view is copyright © 1990 by Steve Kirkendall and was written by Steve Kirkendall (kirkenda@cs.pdx.edu), assisted by numerous volunteers. It is freely redistributable, subject to the restrictions noted in included documentation.

elvis is distributed as a service to COHERENT customers, as is. It is not supported by Mark Williams Company. *Caveat utilitor.*

virtual console — Technical Information

COHERENT system of multiple virtual consoles

The "virtual consoles" feature of COHERENT allows you to run multiple sessions from the system console. You can switch between sessions on the console screen using the appropriate keystrokes. If your computer has both monochrome and color video adapters and monitors, you can run multiple sessions on both screens simultaneously.

For this feature to be available, your system must be configured for virtual consoles. Normally, this configuration is done during installation. In addition, virtual console sessions must be *enabled* for logins prior to use. Virtual terminals are most useful when your system is running in multiuser mode.

COHERENT allows up to ten sessions at a given time. All you need to do to access multiple sessions is to hold down the **<Ctrl>** key on the system keyboard and press the digit on the numeric keypad corresponding to the desired active session number. Simultaneously pressing keys **<Ctrl>** and **<.>** (located on the numeric keypad) will take you to the next *open* virtual terminal session. Another means of switching sessions is to hold down the **<Alt>** key and press one of the "function keys". By default, function key **<F10>** takes you to the next *open* virtual terminal session, **<F11>** takes you to the previous *open* virtual terminal session, and **<F12>** toggles between the current and previously selected sessions.

Technical Features

It is not essential to know the following in order to use virtual terminals. We provide this information for advanced users, as well as persons wishing to customize their systems in ways not available under the default scheme used by the COHERENT installation procedure.

Different sessions are accessed by using different device names in directory **/dev.** Like any *character special* device, each virtual terminal screen has a *major* and *minor* number associated with it. The major number for all virtual terminal screens is 2. The device with minor number 0 is initially the console device — this is where output appears during startup and at other times when the system is in single-user mode. Virtual terminals are assigned successive minor numbers. When there are both color and monochrome display adapters on the system, the color sessions are given the lower minor numbers. For example, in a system configured for four color and four monochrome sessions, *logical* devices might be numbered as follows:

```
crwxr-xr-x 1 root   root     2   0 Mon Jun 15 14:51 /dev/console
crwxr-xr-x 1 root   root     2   1 Mon Jun 15 14:51 /dev/vcolor0
crwxr-xr-x 1 root   root     2   2 Mon Jun 15 14:51 /dev/vcolor1
crwxr-xr-x 1 root   root     2   3 Mon Jun 15 14:51 /dev/vcolor2
crwxr-xr-x 1 root   root     2   4 Mon Jun 15 14:51 /dev/vcolor3
crwxr-xr-x 1 root   root     2   5 Mon Jun 15 14:50 /dev/vmono0
crwxr-xr-x 1 root   root     2   6 Mon Jun 15 14:50 /dev/vmono1
crwxr-xr-x 1 root   root     2   7 Mon Jun 15 14:50 /dev/vmono2
crwxr-xr-x 1 root   root     2   8 Mon Jun 15 14:50 /dev/vmono3
```

Alternatively, using *physical* device numbering, successive color-only sessions can be accessed by using minor numbers 64-79, while successive monochrome-only sessions are selected with minor numbers 80-95. The configuration of four color plus four monochrome sessions described above could also be represented as:

```
crwxr-xr-x 1 root   root     2  64 Mon Jun 15 14:51 /dev/color0
crwxr-xr-x 1 root   root     2  65 Mon Jun 15 14:51 /dev/color1
crwxr-xr-x 1 root   root     2  66 Mon Jun 15 14:51 /dev/color2
crwxr-xr-x 1 root   root     2  67 Mon Jun 15 14:51 /dev/color3
crwxr-xr-x 1 root   root     2  80 Mon Jun 15 14:50 /dev/mono0
crwxr-xr-x 1 root   root     2  81 Mon Jun 15 14:50 /dev/mono1
crwxr-xr-x 1 root   root     2  82 Mon Jun 15 14:50 /dev/mono2
crwxr-xr-x 1 root   root     2  83 Mon Jun 15 14:50 /dev/mono3
```

The following diagram summarizes bit assignments in the virtual terminal minor number:

```
7654 3210
 |           1=physical device, 0=logical device
  ||         00=color, 01=mono, 1x=reserved
    ||||     terminal's index number
```

The system initially defaults to a maximum of four color and four monochrome sessions. This may be altered by patching *character* variables **VTVGA** and **VTMONO**. For example, to allow for six color and three monochrome sessions, enter the following command while running as root (note that this will not take effect until after the

system has been rebooted):

```
/conf/patch -v /coherent VTVGA=6:c VTMONO=3:c
```

Running multiple sessions on different virtual consoles requires that logins be enabled for each of the virtual consoles. Each session must have a corresponding entry in file **/etc/ttys**. For example, a system allowing four color and four monochrome sessions would have entries in **/etc/ttys** as follows:

```
0lPconsole
1lPcolor0
1lPcolor1
1lPcolor2
1lPcolor3
1lPmono0
1lPmono1
1lPmono2
1lPmono3
```

Device **/dev/console** must not be enabled when using virtual consoles! Additional lines would be present if logins are enabled for other devices such as serial ports. Commands **enable** and **disable** may be used, as usual, to allow or disallow logins on individual virtual consoles.

When virtual terminals are enabled, kernel output, such as messages about *user traps* or *system panics*, goes to the currently active session (i.e., the session with the cursor showing).

Altering Virtual Consoles

To add, delete, or alter the configuration of virtual consoles, log in as the superuser **root** and type the following commands:

```
cd /etc/conf
console/mkdev
bin/idmkcoh -o /kernel_name
```

where *kernel_name* is what you wish to name the newly built kernel. When you reboot, invoke *kernel_name* in the usual manner and your new configuration will have been implemented.

See Also

Administering COHERENT, console, device drivers, enable, kb.h

Notes

Some confusion can arise when you attempt to install COHERENT to use both color and monochrome consoles.

At installation time, you are asked if you want to install both color and monochrome screens. If you reply "yes," you must select only four multiscreens for each. Otherwise, you will find it difficult to address virtual consoles on both consoles: COHERENT uses the lower function keys for virtual consoles on the color monitor, and the upper function keys for those on the monochrome monitor.

If you have requested two consoles, COHERENT uses the color terminal by default. If you really have only a monochrome monitor plugged into your system, you must invoke the appropriate monochrome virtual console; otherwise, you will nothing on your monitor.

void — C Keyword

Data type

The keyword **void** indicates that the function does not return a value. Using **void** declarations makes programs clearer and is useful in error checking. For example, a function that prints an error message and calls **exit** to terminate a program should be declared **void** because it never returns. A function that performs a calculation and stores its result in a global variable (rather than **return**ing the result), or one that returns no value, should also be declared **void** to prevent the accidental use of the function in an expression.

See Also

C keywords
ANSI Standard, §6.1.2.5

volatile — C Keyword

Qualify an identifier as frequently changing

The type qualifier **volatile** marks an identifier as being frequently changed, either by other portions of the program, by the hardware, by other programs in the execution environment, or by any combination of these. This alerts the translator to re-fetch the given identifier whenever it encounters an expression that includes the identifier. In addition, an object marked as **volatile** must be stored at the point where an assignment to this object takes place.

See Also

C keyword, const
ANSI Standard, §6.5.3

Notes

Although COHERENT recognizes this keyword, the semantics are not implemented in this release. Thus, storage declared to be **volatile** might have references removed by optimizations that the compiler performs. The compiler will generate a warning if the peephole optimizer is enabled and the keyword **volatile** is detected.

vprintf() — STDIO Function (libc)

Print formatted text into standard output stream
#include <stdarg.h>
#include <stdio.h>
int
vprintf(*format***,** *arguments***)**
char **format***; **va_list** *arguments***;**

vprintf() constructs a formatted string and writes it into the standard output stream. It translates integers, floating-point numbers, and strings into a variety of text formats. **vprintf** can handle a variable list of arguments of various types. It is roughly equivalent to **printf()**'s conversion specifier **%r**.

format points to a string that can contain text, character constants, and one or more *conversion specifications*. A conversion specification defines how a particular data type is converted into a particular text format. Each conversion specification is introduced with the percent sign '%'. (To print a literal percent sign, use the escape sequence '%%'.) See **printf()** for further discussion of the conversion specification and for a table of the type specifiers that can be used with **vprintf()**.

After *format* comes *arguments*. This is of type **va_list**, which is defined in the header **stdarg.h**. It has been initialized by the macro **va_start()** and points to the base of the list of arguments used by **vprintf()**. Each argument must have basic type that can be converted to a pointer simply by adding an '*' after the type name. This is the same restriction that applies to the arguments to the macro **va_arg()**. For more information, see **va_arg()**.

arguments should access one argument for each conversion specification in *format* of the type appropriate to conversion specification. For example, if *format* contains conversion specifications for an **int**, a **long**, and a string, then *arguments* access three arguments, being, respectively, an **int**, a **long**, and a **char** *.

If there are fewer arguments than conversion specifications, then **vprintf**'s behavior is undefined (and probably unwelcome). If there are more, then **vprintf()** evaluates and then ignores every argument without a corresponding conversion specification. If an argument is not of the same type as its corresponding type specification, then the behavior of **vprintf()** is undefined; thus, accessing an **int** where **vprintf()** expects a **char *** may generate unwelcome results.

If it writes the formatted string correctly, **vprintf()** returns the number of characters written; otherwise, it returns a negative number.

See Also

fprintf(), libc, printf(), sprintf(), vfprintf(), vsprintf()
ANSI Standard, §7.9.6.8

Notes

vprintf() can construct a string up to at least 509 characters long.

vsh — Command

Interactive graphical shell
vsh [-d*directory***] [-eirt]**

vsh is the COHERENT system's visual shell. With it, users can use arrow keys or simple keystrokes to perform tasks under the COHERENT, such as change directories, edit files, and execute programs. Each user can program a bank of up to nine function keys to perform complex tasks with a single keystroke. With **vsh**, a naive user can access much of the power of the COHERENT system without having to learn the details of **sh** or **ksh**.

Unlike X or other windowing systems, **vsh** works on a character-based terminal and requires only a modest amount of memory. It does not require a mouse.

Graphics Interface

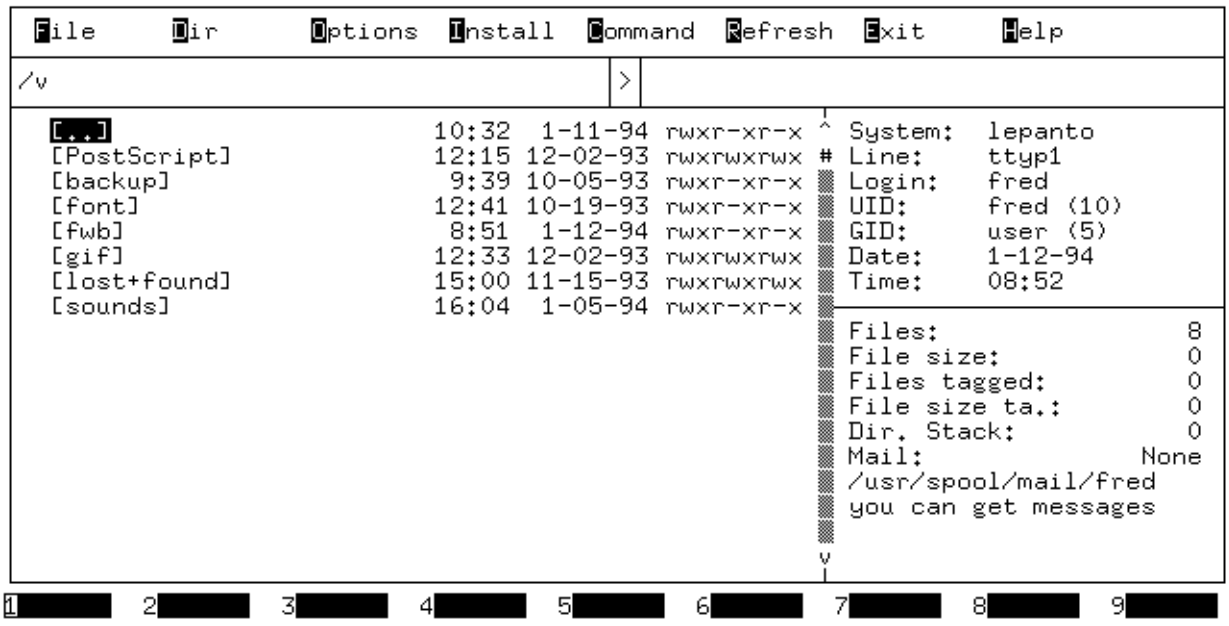
vsh uses the **curses** library and **terminfo** descriptions. To use **vsh**, you must have a **terminfo** description installed for the device upon which you wish to run it, and you must set the environmental variable **TERM** to point correctly to that description. For example, to run **vsh** from your console, you should set **TERM** to **ansipc**; while to run it from a PC that is plugged into a serial port, you should set **TERM** to **vt100**. You must have a **terminfo** description for the device to which you set **TERM**, or **vsh** will behave bizarrely. For more information on devices and how to set them, see the Lexicon entries for **TERM** and **terminfo**. For more information on terminals in general, see the entries for **terminal** and **console**.

To ensure that **TERM** set correctly, you may wish to embed the command **ttytype** in the file **/etc/profile**. For more details, see the Lexicon entry for **ttytype**.

If you have a non-standard terminal or have trouble displaying **vsh**, try invoking it with the options **-e** or **-t**. All of **vsh**'s command-line options are described below.

Main Screen

When you invoke **vsh**, you see a screen that appears as follows:



As you can see, the screen is divided the following six sections, or *windows*:

- The first window, the *Command* Window, is the narrow window that runs across the top of the screen. This window lists the commands that **vsh** can perform. You will enter this window frequently as you work with **vsh**.

- The second window, the *Current Directory* window, names the directory that you are currently in.
- The third window, the *Destination Directory* window, names the default destination directory.
- The fourth window, the *File* Window, extends down the left side of the screen. It lists the contents of the current directory. You will also work frequently in this window.
- The fifth window, the *System* Window, is the upper window on the right side of the screen. It gives information about the system, that is, who is running **vsh**, the device she is running it on, and the current date and time. Your cursor never enters this window.
- The last window, the *Status* Window, gives information about the work you have performed under **vsh**. Again, your cursor will never enter this window.

Across the bottom of the screen are nine "stubs," one each for function keys one through nine. The stub's text indicates the command that **vsh** executes when you press that key.

The following sections discuss each window in detail.

File Window

The file window lists all of the files and directories within the current directory. This is the default window for **vsh**; the cursor ordinarily rests in this window, and you will do most of your work in it.

The leftmost column in the File Window gives the name of each file and directory. Directories are given at the top of the list; they are enclosed within brackets '[ ]'. The other columns give, respectively the time the file or directory was last updated; the date it was last updated; and its permissions. For information on how to interpret the permissions string, see the Lexicon entry for the command **ls**.

The top listing in the File Window is always **[..]**, which represents the current directory's parent directory.

The top listing in this list is highlighted by being shown in reverse video. To move the highlighting bar up and down the list, use the arrow keys. If you press the arrow keys on your keyboard's number pad, be sure to turn the **<NumLock>** key *off*, or the keys will not work as you expect. If you press the (°) key, the bar shifts down one row on the list. Pressing the (ª) key moves the bar up one row.

You can page up or page down by pressing, respectively, the keys **<PgUp>** and **<PgDn>**. The key **<Home>** moves the cursor to the top of the list, and **<End>** moves it to the bottom. If your terminal does not implement these keys, you can use the following control characters:

<ctrl-N>	Next page (like **<PgDn>**)
<ctrl-P>	Previous page (like **<PgUp>**)
<ctrl-A>	Beginning (top) of list (like **<Home>**)
<ctrl-E>	End (bottom) of list (like **<End>**)

Note that if the list of files and directories is too large to fit into the window, moving the bar to the bottom of the window and pressing (°) will scroll the list. If you press the **<End>** key, the row moves to the last row in the list; and if you press **<Home>**, it moves to the top of the list.

A scroll bar runs down the right side of the File Window. As you scroll up and down this window, the scroll bar moves. Note that the position of the scroll bar is proportional to the highlighting bar's position in relation to the entire list of files, not just to its current position within the File Window. This gives you an easy way to see just where you are in the entire file list.

If you position the highlighting bar over the name of a directory and press (¢), **vsh** names that directory in the Current Directory Window and displays its contents in the File Window. For example, if you position the highlighting bar over the entry for directory **[letters]** and press (¢), **vsh** displays the contents of directory **letters** in the File Window. (If you are familiar with the Bourne or Korn shell, this has the same effect as typing the command **cd letters**.) To return to the directory you had just been displaying (that is, the parent directory of **letters**), use the arrow keys to move the highlighting bar to the entry **[..]**; then press (¢). **vsh** changes the contents of the Current Directory Window, and in the File Window erases the contents of **letters** and displays the contents of its parent directory.

If you press (¢) while a file is highlighted instead of a directory, **vsh** does the following:

1. If the file is executable, **vsh** executes it.

2. If the file matches a pattern from the file-action list, **vsh** executes the action from the list with the file as input. The file-action list is in file **$HOME/.vsh**; it looks like:

```
[Mm]akefile:make
*.mk:make -f %F
*.sh:sh %F
*.c:cc -c -O %F
*.sc:sc %F
*.a:ar tv %F | more
*.[1-9]:nroff -man %F | more
*.tar.F:fcat %F | tar xvf -
*.F:fcat %F | more
*.tar.Z:zcat %F | tar xvf -
*.Z:zcat %F | more
```

vsh recognizes most common wildcard characters; for a table of these and their meaning, see the Lexicon entries for **wildcards**. The token **%F** stands for the file that is currently highlighted. For example, in the above example the entry ***.Z:zcat %F** means that if you select a file with the suffix **.Z** (which usually means that a file has been compressed), it passes that file to **zcat** to uncompress and display it. **vsh** defines many defaults for you when it creates this file, which you can use as a model. To change the file-action list, use the **File actions** sub-command of the **Install** command, which is described below.

3. If the file appears to be ASCII **vsh** displays it with the default viewer.

While **vsh** is working, it displays a large letter 'X' in reverse video in the lower right corner of the screen. This shows that **vsh** is doing some internal task. **vsh** cannot accept any commands while the 'X' is displayed, so please be patient.

Also, note that **vsh** cannot handle more than 1,000 files in any given directory. If a directory contains more than 1,000 files, only the first thousand will be available for use.

System Window

The system window is the upper of the two windows on the right side of the screen. The cursor never enters this window; rather, this window simply displays information about your COHERENT system, and how you are currently using it. It contains the following entries:

System: This gives the name of your system, as you (or your COHERENT system administrator) has set it in file **/etc/uucpname**. See the Lexicon's entry for **uucpname** for more details on proper naming conventions for COHERENT systems.

Line: This gives the device by which you are accessing your COHERENT system. If you are working on your system's console device, then you should see **console** on this line; whereas if you are accessing your COHERENT via a PC plugged into serial port **com1l**, you should see **com1l** here. If you are using virtual consoles, the line is shown as **mono[0-8]** or **color[0-8]**. See the Lexicon entries for **console** and **asy** for more information about the devices through which you can access a COHERENT system.

Login: This gives the name under which you logged into COHERENT. For example, if your login identifier is **fred**, then you should see **fred** on this line.

UID: This shows your user-identification number (or UID). This is the unique number by which your COHERENT system knows you, as set in file **/etc/passwd**. For information on the UID and how to set it, see the Lexicon entries for **passwd** and **setuid**.

GID: This gives the number and name of the user group to which you belong. Users on a COHERENT system can be organized into groups; permissions on files can be set to include the members of your group, but exclude all others. For information on groups, see the Lexicon entries for **group** and **setgid**.

Date: This gives today's date (or rather, what your COHERENT system thinks today's date is).

Time: This gives what your system thinks the current time is. If your system's time is not set correctly, then the time shown here will not be correct. For information on how to set the system time, see the Lexicon entries for the commands **ATclock** and **date**.

The time can also vary depending upon what time zone your COHERENT system thinks it's located in. For information on timezones and how to set them correctly, see the Lexicon entry for **TIMEZONE**.

Command Window

The Command Window is the top window, and stretches across the width of the screen. This window gives you access to **vsh**'s commands. Some commands in the command window actually open an entire menu of commands, with which you can perform all manner of work.

The command window contains the following entries. For convenience, the following displays the entries vertically; the actual window displays them horizontally:

File
Directory
Options
Install
Command
Refresh
Exit
Help

When the cursor is in the File Window (which is the default) and you wish to execute one of the commands in the Command Window, press its initial letter. For example, to execute the **Refresh** command, press **R**.

Note that the commands on this window are in two groups. A command's behavior differs, depending upon which group it belongs to.

The commands **File**, **Directory**, **Options**, and **Install** display a drop-down menu when you invoke it. That is because they have more than one option available under it. If you do not wish to invoke any of the sub-commands on that menu, you can do either of the following: You can press the **<Esc>** key, which erases the drop down-menu and returns you to the File Window; or you can press the (æ) or (Æ) keys, which move you to the command in this group that lies, respectively, to the left or to the right of the current command. For example, suppose that you were in the File Window, and you pressed **F**, to invoke the **File** command. **vsh** would move the cursor into the Command Window, and display the File Command's drop-down window, which displays its sub-commands. If you then pressed the **<Esc>** key, **vsh** would return you to the File Window. If you pressed the (Æ) key, **vsh** would erase the **File** command's drop-down window and display, instead, the drop-down window for the **Directory** command. If, however, you pressed the (æ) key, **vsh** would erase the **File** command's drop-down window and display, instead, the drop-down window for the **Help** command. As you can see, **vsh** "wraps-around" the cursor — it considers the command at the far right to be to the left of the command to the far left, and vice versa. This concept is a little difficult to grasp when you read about it, but once you try it, it will quickly become clear.

Please note that **vsh** delays for one second its reaction to the **<esc>** key. The **curses** function **wgetch()**, which is used to read the keyboard, needs this delay so it can distinguish between the **<esc>** key and the other function keys, which all of which start with an **<esc>**. So, please be patient.

The other group of commands are the commands **Command**, **Refresh**, **Exit**, and **Help** each have only option, so when you invoke one of them, it immediately begins to execute that option. When you access one of these commands through the (Æ) and (æ) keys, each displays a drop-down menu that shows its one option.

The following describes each command in detail.

File Pressing **F** invokes the **File** command. This displays a drop-down menu that lists a set of sub-commands. These sub-commands let you manipulate files; with them, you can edit a file, create a file, change its permissions, rename it, erase it, print it, or do other common tasks.

To invoke a sub-command, you can do either of the following: Press the letter in the sub-command that is underlined (each sub-command has its own unique letter with which you can invoke it); or use the (ª) and (°) keys to move the highlighting bar to that command, and then press (¢).

The following discusses each sub-command in detail:

Copy This sub-command copies a file. Please note that the behavior of this subcommand depend upon whether you have tagged files.

If you have tagged one or more files, **vsh** opens a pop-up window that requests the path name of a directory. By default, **vsh** displays the destination directory, if you have set one. When you enter the path name, **vsh** copies every tagged file into that directory.

If you have not tagged any files, **vsh** opens a pop-up window that requests that you enter a file name or a path name. Again, if you have set a destination directory, the window displays it by default. If you enter only a file name into this window, **vsh** copies the highlighted file into the

newly named file in the current directory; if you have named an existing file, **vsh** prompts you before it overwrites that file. If you enter a path name, **vsh** copies the highlighted file into the directory you have named; the copied file retains its current name. If, however, you enter both a file name and a path name, then **vsh** copies the highlighted file into the directory you named, and gives it the file name that you entered.

Note that this command will not overwrite a file that you do not own; nor will it create a new file in a directory in which you do not have write permission, or copy a file on which you do not have read permission. For more information on copying files under COHERENT, see the Lexicon entry for the command **cp**.

Move This sub-command prompts you for the name of a directory; if you have set a destination directory, **vsh** displays it by default. When you confirm the destination, **vsh** then moves all tagged files into it. (If no files are tagged, **vsh** moves only the highlighted file. For more information on tagging, see the entry for the sub-command **Tag**, below.) The files retain their names in the new directory.

This command does not move a file for which you do not have read permission, or move a file into a directory into which you do not have write permission; nor will it move a file into a non-existent directory (of course). For details on moving files, see the Lexicon entry for the command **mv**.

Delete This sub-command deletes the tagged files. (If no files are tagged, then it deletes only the highlighted file. For more information on tagging, see the entry for the sub-command **Tag**, below.) It will prompt you to confirm that you really do want to delete the file or files in question. With regard to the mass deletion of tagged files, this sub-command lets you choose whether to do a mass deletion or delete files one at a time.

Note that this sub-command will not delete a file that you do not own. For details on deleting files, see the Lexicon entry for the command **rm**.

Rename
This sub-command lets you rename the highlighted file. It opens a pop-up window that shows the current name of the file, and prompts you to type the new name. Press **<Esc>** to abort this sub-command, or type the new name and press (¢).

It does not work with directories. It will not let you rename a file that you do not own. For details on renaming a file, see the Lexicon entry for the command **mv**.

Execute
This sub-command executes the highlighted file. **vsh** prompts you to type the arguments you wish to pass this file, then invokes the file with those arguments.

Note that **vsh** will not execute a file for which you do not have execute permission.

Access This sub-command lets you change the manner in which every tagged file can be accessed. (If no files are tagged, the default is the highlighted file.) When you invoke it, **vsh** displays the following pop-up window for each tagged file:

```
Change access f file filename

Owner
Read [x]       Write[x]        Execute[ ]

Group
Read [x]       Write[ ]        Execute[ ]

World
Read [x]       Write[ ]        Execute[ ]

Special
Set UID [ ]    Set GID [ ]     Set sticky[ ]
```

An 'x' in a field means that that permission is turned on; a blank means that it is turned off. Use the arrow keys to move to the cursor the field whose status you wish to change, then enter a space or 'x' to, respectively, turn off or turn on that given permission. To abort this command, press **<Esc>**.

For information what permission fields mean, see the Lexicon entry for **ls**. Note that you can reset permissions only on the files you own.

Owner This lets you change the owner and group that owns each tagged file. If no files are tagged, then this applies only to the highlighted file. When you invoke this sub-command, **vsh** opens a pop-up window that shows the user and group that own a file: type the name of the user or group you want to own the file. **vsh** repeats this step for each tagged file. To abort this command, press **<Esc>**.

For details on changing ownership of a file, see the Lexicon entries for the command **chown** and **chgrp**. Note that only the superuser **root** can run this command.

Print This passes every tagged file to the print spooler for printing. To change the default print spooler, use the **Install** command's **Print spooler** sub-command.

Note that **vsh** simply passes the file to the spooler for printing; you cannot use this to process a file before printing it. If you try to use this feature of **vsh** to print a file on a PostScript printer, the printer will hang. We suggest that you use the **Command** feature to print a file on a specialized printer; it's a little more difficult, but it works. Another approach is to use the spooler **lp** and prepare a special backend script to do the processing automatically. For details on how to do that, see the Lexicon entries for **lp** and **printer**.

View This sub-command invokes the default viewer to display the contents of every tagged file. If you try to view the contents of a binary file, the results may not be what you expect.

Note that **vsh** will not display a file for which you do not have read permission. To change the default viewer, use the **Install** command's **File viewer** sub-command.

Edit This sub-command invokes the text editor to edit every tagged file. If no files are tagged, then edit only the highlighted file.

The default text editor is **vi**, which can create problems for persons who do not know how to exit from that editor. For a quick brush-up on **vi**, see the Lexicon entry for **elvis**. To change the default text editor, use the **Install** command's **Editor** sub-command. Note that COHERENT will not let you edit a file for which you do not have read permission.

Edit new
This sub-command prompts you to type the name of a file, then invokes the editor for that file. This can be a new file (that is, one that does not yet exist in the current directory), or a file that already exists.

Note that if you do try to edit a binary file, you may find yourself running into difficulties.

Touch This "touches" every tagged file — that is, it changes the date and time that the file was last modified, just the same as if you had just edited it.

Note that you cannot touch a file for which you do not have write permission. For more information on touching files, see the Lexicon entry for the command **touch**.

Tag all This sub-command "tags" every file in the current directory. This lets you do mass moves or deletions of files. When you tag a file, **vsh** updates the entries **Files tagged** and **File size ta.** in the Status Window, to reflect the number and total size of the files you have just tagged. It also prints an asterisk next to the tagged file.

When the cursor is in the File Window, you can toggle tagging on the highlighted file by pressing the space bar. Note that the highlighted file is implicitly tagged, whether an as asterisk appears next to it or not. For details, see the section on the Status Window, below.

Untag all
This sub-command untags all files that are tagged in the current directory. As noted above, you can toggle the tagging of the highlighted file by pressing the space bar. This command updates the Status Window to reflect your changes.

Select This sub-command opens a pop-up window and lets you enter a regular expression; it then tags all files that match the expression. For example, if you enter ***.c**, then this sub-command tags all files that end in the string **.c**.

File type
This sub-command prints a summary of information about the type of the highlighted file.

File info
This sub-command opens a pop-up window that displays the following information about the highlighted file or directory:

```
Filename
Filetype
I-Node
Links
Owner UID
Owner GID
access
modification
status changed
```

Filename is the name of the file. *Filetype* is its type, e.g., directory or regular file. *I-Node* gives the number of this file's i-node; for information on what an i-node is, see its entry in the Lexicon. *Links* gives the number of links to the file. For information on what a link is, see the Lexicon entries for **ln** and **link()**. *Owner UID* and *Owner GID* identify the owner and group that own this file. For information on what the UID and GID are, see the Lexicon entries for **setuid** and **setgid**. *access*, *modification*, and *status changed* give, respectively, the date and time the file was last accessed, last modified, or last had its status changed.

Directory
Pressing **D** invokes the **Directory** command. This displays a drop-down menu that lists a set of sub-commands. These sub-commands let you manipulate directories; with them, you can create a directory, remove a directory, change permissions, and other common tasks. You can also manipulate a "directory stack," which lets you jump quickly from one directory to another without having to retype its name.

The following discusses each sub-command in detail:

Change
This lets you change the current directory. When you invoke this subcommand, **vsh** displays the following pop-up window:

```
Enter destination path
```

Type the full path name of the directory you wish to enter. If this directory does not exist, or if you cannot access it, **vsh** leaves you in the current directory; otherwise, it moves you to the requested directory.

Home This moves you to your home directory.

User's Home
This moves you to the home directory of another user. When you invoke this sub-command, **vsh** asks you to name the user whose home directory you wish to enter. To abort, press **<Esc>**. If the user you enter does not exist or if you do not have permission to read her home directory, **vsh** leaves you in the current directory; otherwise, **vsh** moves you into that user's home directory.

Set dest
Set the destination directory. This directory is saved in your **.vsh** file, and is restored the next time you invoke **vsh**.

Push The next three sub-commands makes it easy for you to maneuver your way around the COHERENT file system. The work by using what is called a "directory stack". In effect, you can tell **vsh** to remember the directory you are in (this is termed "pushing" the directory onto the stack); then, when you have switched to another directory, you can returned to this directory simply by "popping" this pushed directory from the directory stack. This lets you move around among directories without having to retype them continually.

The **Push** sub-command pushes the current directory onto the directory stack. When you push a directory, **vsh** increments the number next to the entry **Dir. Stack** in the Status Window. This tells you how many directories you have pushed onto the directory stack.

Pop & cd
This sub-command moves you to the last directory you pushed onto the directory stack. It also removes that directory from directory stack. When you pop a directory from the directory stack, **vsh** decrements the number next to the entry **Dir. Stack** in the Status Window. This tells you how many directories remain on the directory stack.

Note that directories are popped in the order opposite from that in which they were entered. For example, if you pushed directory **/usr/bin/sys** onto the directory stack, then directory **/usr/lib/mail**, then **/bin**, invoking the **Pop** sub-command will return you to directory **/bin**, then to **/usr/lib/mail**, and finally to directory **/usr/include/sys**.

Switch This command switches the current directory and the top entry in the directory stack.

Copy This copies the highlighted directory plus all of its contents into another directory whose name you type into a pop-up window. It behaves much like the command **cpdir**.

Delete This deletes the highlighted directory. It does not work with files. If the directory has files in it, **vsh** will prompt you and ask if you want the directory to vanish. If you answer 'Y', then **vsh** removes it, files and all — just as if you had executed the command **rm -rf**.

vsh will not delete a directory that you do not own.

Rename
This sub-command renames the highlighted directory. **vsh** opens a pop-up window and prompts you to type the new name of the directory. Press **<Esc>** to abort this sub-command. Note that you can rename only directories that you own. This sub-command does not work with files.

Create This sub-command creates a new directory in the current directory. **vsh** prompts you for the name of the new directory, and then creates it. Note that you can create a directory only if you have write permission in the current directory.

Access This lets you reset the access permission on the highlighted directory. This is the directory equivalent of the **File** command's **Access** sub-command.

Owner This lets you reset the user and group that own a given directory. This is the directory equivalent of the **File** command's **Owner** sub-command. Note that only the superuser **root** can run this command.

Read new
This tells **vsh** to re-read the current directory. **vsh** copies the contents of the current directory into memory for its own use; thus, if other people manipulated the directory and its contents after **vsh** read its contents, what you see in the File Window will not reflect the true state of affairs in that directory. If you are working with a directory that is being manipulated by one or more other people, you should issue this command from time to time, to ensure that you are working with an accurate image of the directory's contents.

Switch CWD
This command switches the current working directory with the destination directory.

Switch TOS
This switches the destination directory with the directory on top of the stack.

Info This is the same as the **File info** sub-command under the **File** command, described above.

Options
Pressing **O** invokes the **Options** command. Its sub-commands let you perform common system tasks. The following discusses each sub-command in detail:

Shell This command invokes an interactive shell. When you exit from the shell (either by typing **exit** or **<ctrl-D>**), you will be returned to **vsh**.

By default, **vsh** invokes the Bourne shell **sh**; to change the default shell, use the **Shell** sub-command under the **Install** command, which is described below.

Lock terminal
This command locks your terminal. When the terminal is locked, no command can entered into it; this lets you walk away from your terminal briefly without worrying whether anyone (e.g., your cat) will do anything untoward under your login. The terminal remains locked until you retype the

secret password that you entered when you invoked this sub-command

When you invoke this sub-command, a pop-up window appears with the following:

```
Lock Enter Password
```

vsh prints a '#' to echo each character that you type. If you wish to abort the **Lock** sub-command, press **<Esc>**. When you have finished entering your password, press (¢). When you have entered the password, the following window appears:

```
This Terminal is locked!

Enter Password to unlock
or hit return to logoff
```

Type the password to return to **vsh**. If you (or someone else) presses (¢), you will be logged out of COHERENT.

Messages

This sub-command lets you receive or ignore messages. A message can be sent to your terminal by another user or another process; for example, the **mail** command may send a prompt to your screen when new mail is received.

When you invoke this sub-command, **vsh** displays the following pop-up window:

```
Do you want to receive messages ?

       Yes         No
```

Use the (Æ) and (æ) keys to select the option you want, then press (¢). When you change your message status, the information in the Status Window changes. For example, when you turn off messaging, the following appears at the bottom of the Status Window:

```
You can't get messages
```

For information on how COHERENT sends messages to your terminal, see the Lexicon entry for **mesg**. Also, see the description of the Status Window, below.

Online manual

This lets you select an entry from the COHERENT system's on-line manual pages. When you invoke this sub-command, **vsh** displays the following pop-up window:

```
Enter topic, chapter is optional :

Topic:

Chapter:
```

Type the title of the Lexicon entry that interests you; for example, to see the Lexicon entry for the command **vsh**, enter **vsh** in the **Topic** slot, then type (¢). Do not enter anything into the **Chapter** slot; this does not apply to the COHERENT system. You will see on your screen the Lexicon entry that you are now reading. If you change your mind, press **<Esc>** to abort this command.

Note that if you did not install or uncompress the manual pages when you installed your COHERENT system, this sub-command will not work. For more information on the COHERENT manual pages, see the Lexicon entries for the commands **help** and **man**.

System news

Display news about your current system. By default, this invokes the COHERENT command **msgs**.

Internet news

Invoke a reader for Internet news. By default, this command invokes **rn**, should you have it installed.

Electronic mail
Invoke your mail reader. By default, this invokes **mail**.

Install Pressing **I** invokes the **Install** command. Its sub-commands let you modify some of **vsh**'s default behaviors; in particular, it lets you program your function keys to execute some tasks you select with one keystroke. The following discusses each sub-command in detail:

Display
This command lets you customize appearance of **vsh**. When you invoke this sub-command, **vsh** displays the following pop-up window:

```
Display Attributes

Menubar
Menu color
Menu attribute
Dialog box
```

The entry **Menubar** lets you select the display attribute for the menu bar, which can be one of **bold**, **underline**, or **reverse**.

The entry **Menu color** lets you set the menu color, which can be either **normal** or **reverse**. (This may vary, depending on the type of terminal you are using.)

The entry **Menu attribute** lets you set the display attribute for pulldown menus, which can be one of **bold**, **underline**, **bold**, or **normal**.

Finally, the entry **Dialog box** lets you set the display attribute for dialogue boxes, which can be one of **bold**, **underline**, or **both**.

The best way to see what these commands do is to try them out. As mentioned above, the behavior may change from device to device, depending upon the type of terminal that you are using.

Function keys
This lets you "program" up to nine function keys, so you can invoke selected commands easily. Each user can have her own list of programmed function keys.

When you invoke this sub-command, **vsh** displays the following drop-down menu:

```
Function keys

Function key 1
Function key 2
   ...
Function key 9
```

Press 1 through 9 to program the corresponding function key (or use the (ª) and (°) keys to move then highlighting bar, then press (¢)). **vsh** asks you to enter the label for the function key and the command you want that function key to invoke. When you have finished, the new label will appear in the corresponding function-key tag at the bottom of the screen; and when you press that function key, **vsh** executes the corresponding command.

For example, to make the game **chase** one of your function key entries, do the following: First, press **I** to invoke the **Install** command. The press **k** to invoke the **Function keys** sub-command. When the function-keys drop-down menu appears, press **2**, for function-key **F2**. When the label pop-window appears, type **chase** into the first slot, which holds the label Press **<Tab>** to jump to the second slot, which holds the command to execute, then type **/usr/games/chase**. When you have done typing, press (¢).

As you can see, the **F2** stub at the bottom of the screen shows **chase**; and when you press **F2**, **vsh** launches you into **chase**. You can program the first nine function keys to work in the same way.

You can embed the token **%F** as a placeholder for the current file. For example, to count the number of lines in the current file, put the following command into a function-key definition:

```
wc -l %F
```

Because some computers still do not have function keys (e.g., the NeXT machine), you can also use the number keys to execute commands installed on the function keys.

By the way, for information on the highly amusing game **chase**, see its entry in the Lexicon.

Shell This sub-command lets you set the default shell that **vsh** runs when you invoke its **Shell** command. When you invoke this sub-command, **vsh** displays the following pop-up menu:

```
Enter command to run a shell
(Coherent default is '/bin/sh')

      /bin/sh
```

Type the shell that you want, either **/bin/sh** or **/bin/ksh**, and press (¢). (You can enter another program if you like, but you may get some strange results if you do.) For information on each shell, see its entry in the Lexicon.

Editor This lets you set the editor that **vsh** invokes when you select the **Edit** sub-command under the **File** command. When you invoke the **Editor** sub-command, **vsh** displays the following pop-up window:

```
Enter command to run an editor
(Coherent default is 'vi')

      vi
```

Type the editor that you want, one of **ed**, **me**, or **vi**; then press (¢). For information on each editor, see its entry in the Lexicon.

Print spooler

This lets you set the spooler that **vsh** invokes when you select the **Print** sub-command under the **File** command. When you invoke the **Print spooler** sub-command, **vsh** displays the following pop-up window:

```
Enter command to run a print-spooler
(Coherent default is 'lpr -B')

      lpr -B
```

Enter the spooler that you want. For more information on the spooling commands available under COHERENT, see the Lexicon entry **printer**.

Beginning with release 2.7 of **vsh**, this feature works with pipes. **vsh** understands that the token **%F** represents the current file. For example, if you have a PostScript printer, you will want every file to be processed by the command **prps** before you print it. Thus, enter the command:

```
prps %F | hpr -B
```

This tells **vsh** to filter each file through **prps** and pipe the output to the laser-printer spooler **hpr**.

Some of this functionality may not be necessary under COHERENT release 4.2, which implements the System-V **lp** print spooler. See the Lexicon article **printer** for details.

File viewer

This lets you set the viewer that **vsh** invokes when you select the **View** sub-command under the **File** command. When you invoke the **File viewer** sub-command, **vsh** displays the following pop-up window:

```
Enter command to run a file view utility
(Coherent default is 'more')

      more
```

Enter one of **more** or **scat**. For information on how these commands differ, see their entries in the Lexicon.

File action

As noted above, **vsh** has a list of default actions that it takes when you select a file of a given type. For example, if you invoke the **File** command, move the highlighting bar to a file with the suffix **.c** and press (¢), **vsh** by default invokes the C compiler **cc** to compile that file.

vsh stores in the file **$HOME/.vsh** the list of its default actions. The **File actions** sub-command invokes a special editor so you can edit this list.

When you invoke this option, **vsh** displays the following pop-up window:

```
Edit actions list
Configure action
```

Use the (ª) and (°) keys to move the highlighting bar to the item you want; then press (¢).

When you select **Edit actions list**, **vsh** displays a pop-up window that contains all of the default actions. The syntax of the default actions is described above. Use the (ª) and (°) keys to move the highlighting bar to the action you wish to edit. To erase the current line, press **<ctrl-D>**; to open a new line, press **<ctrl-I>**.

To modify the line that is currently highlighted, press (¢). When you do so, the highlighting bar disappears and a cursor appears. Use the (æ) and (Æ) keys to move the cursor to the point you wish to change; typing inserts new text into the command, whereas pressing **<Backspace>** erases text. When you have finished modifying the current line, press (¢). To abort modifying the current line, press **<Esc>**.

When you have finished modifying the action list, press **<Esc>**. **vsh** records your changes into file **$HOME/.vsh**, and returns you to the File window.

When you select the option **Configure action**, **vsh** displays a window with the prompt

```
Show file actions before execution ?
```

The cursor is under the response **y**, for **yes**. If you accept this option, **vsh** will prompt you for your confirmation before it performs a default action. If you want **vsh** simply to go ahead and perform its default without asking for your approval, press the (Æ) key to move the cursor to the option **n**, for **no**, and press (¢).

Sys. news reader

Tell **vsh** what system news program you want it to invoke by default.

Internet news

Tell **vsh** what Internet news reader you want it to invoke by default.

Electronic mail

Tell **vsh** what mail reader you want it to invoke by default.

Command

The command **Command** lets you send a command directly to a COHERENT shell. This lets you invoke commands that ordinarily are not available through **vsh**.

Suppose, for example, that you decided you wanted to play a session of the game **tetris**, and that you have not yet programmed **tetris** as one of your function keys. Press **C** to invoke **Command**. **vsh** moves the cursor moves to the bottom of the screen, and erases the row of boxes that describe the function keys. You can now type the command you want, in this case **/usr/games/tetris**. To run the command, press (¢); to abort entering a command and return to **vsh**, type **<Esc>**.

When you press (¢), **vsh** runs the command you typed. When you have finished playing **tetris** and have exited from it, **vsh** clears the screen and displays the message:

```
Hit any key to continue ...
```

When you press a key, **vsh** redraws itself on your screen and returns the cursor to the File Window.

(By the way, the COHERENT version of **tetris** is available as part of COHware volumes 2 and 3. For information on obtaining COHware, see the release notes that came with your copy of COHERENT.)

Command remembers the last 40 commands that you have issued. To invoke a command that you previous issued through **Command**, press the (ª) key. The last command you issued will appear in the command slot. If you continue to press the (ª) key, others commands appear, in reverse order from when you issued them. If you overshoot the command that you want to re-run, press the (º) key to walk back down the list of previously issued commands. When you find the previously issued command that you wish to rerun, just press (¢) and **vsh** runs it again.

You can also edit a previously issued command. The following gives lists the available editing commands:

←	Move the cursor one character to the left
→	Move the cursor one character to the right
<del>	Delete the character to the left
<backspace>	Delete the character to the left
<ctrl-D>	Delete the character over the cursor
<ctrl-P>	Go to last character of the command
<ctrl-N>	Go to first character of the command

A command can use environmental variables, such as **$HOME**. **vsh** will expand all environmental variables correctly before it tries to execute the command.

You can also embed the following tokens in a command:

%F	Represent the currently highlighted file
%T	Represent all tagged files
%D	Represent the destination directory

For example, the command

```
cp %T %D
```

copies all tagged files into the destination directory.

Refresh

The command **Refresh** redraws the screen. It does no other work. This is helpful if your screen has become jumbled or scrambled for any reason — such as a message being written onto your screen by another user.

To invoke this command, type **R**. **vsh** pauses very briefly, then the screen flickers as **vsh** redraws. If the screen had been confused for any reason, invoking this command should restore to its proper state. If you need to refresh the screen while a pop-up menu or a pop-up window is active, press **<cntl-L>**.

Exit The command **Exit** exits you from **vsh**. To exit from **vsh**, press **E**. In response, **vsh** pops the following window onto your screen:

```
Do you really want to quit?

       Yes       No
```

The window is in reverse video, for emphasis. The option **Yes** is underlined, to show that it is the default choice. If you really do wish to exit, press (¢); and **vsh** returns to the COHERENT shell.

If you changed your mind, however, and do not wish to exit, press the (Æ) key to change the option; this will shift the underlining from option **Yes** to option **No**. Pressing enter at this point selects the **No** option; **vsh** in response removes the pop-up window from the screen and returns you to the File Window.

If you change your mind again, though, and really do wish to exit, then press the (æ) key. The underlining shifts back to the **Yes** option; and when you press (¢) you exit from **vsh** and return to the shell.

Status Window

The Status Window is the lower window on the right side of the screen. The cursor never enters this window; rather, this window gives information about how **vsh** is functioning, and in particular about the files that are currently displayed in the File Window.

The Status Window contains the following entries:

Files This gives the number of files being shown in File Window. Note that this is all files that can be scrolled through that window, *not* the files that are shown in that window at this moment.

File size
This gives the total size, in bytes, of all files available through in the File Window.

Files tagged
This gives the number of files that you have tagged. See the description of the **File** command, above, for details.

File size ta.
This gives the total size of all tagged files. See the description of the **File** command, above, for details.

Dir. Stack
This gives the number of directories that currently reside on the directory stack. As noted above, you can "push" directories onto the directory stack or "pop" them from it. By doing so, you have an easy way to jump about from one directory to another, without having to type directory names repeatedly. See the above description of the **Directory** command for more details.

You can have a maximum of ten directories on the stack.

Mail This line indicates whether you have mail waiting to be read. If you don't, this line will say

```
None
```

whereas if you do, the line will say

```
Avail
```

and flash at you. If new mail arrives, **vsh** flashes

```
New
```

in that slot.

mailbox
This line gives the name of your mailbox — that is, the file that **mail** reads.

messages
This indicates whether your terminal can receive messages — e.g., whether a message will pop up on your screen if someone wishes to communicate with you via the **write** command. For more information on how to change the message status of your terminal, see the Lexicon entry for the command **mesg**.

Function Keys

The bottom of the screen show nine small boxes in reverse video. These are labelled **F1** through **F9**. If you have defined the key using the **Function Key** command, **vsh** displays the box the tag that you gave that key.

For example, in our above example we set key **F1** to run the command **ps -a**, and gave it the tag **ps**. At the bottom of the screen, the box labelled **F1** should show **ps**.

For more details, see the description of the **Function Key** command, above.

Configuration File

vsh reads the file **$HOME/.vsh** to configure itself.

A typical **.vsh** file reads as follows:

```
cwd=/v/fwb
shell=/bin/ksh
editor=me
print-spooler=hpr -B
view=more
make=make
me-disp-attr=reverse
pd-disp-color=normal
pd-disp-attr=bold
se-disp-attr=underline
pfkey1=  mail mail
pfkey9=tetris /usr/games/tetris
cmd=
      tetris
      tetris
      echo foo
```

cwd points to the current working directory, that is, the directory in which you have last worked with **vsh**. **vsh** returns you to that directory when you next invoke the shell.

shell, **editor**, **print-spooler**, **view**, and **make** give, respectively, the shell, editor, print-spooler, viewer, and make utility that you selected with the **Install** command. If you change one of these values, the behavior of **vsh** changes to reflect the change. For example, if you change the line

```
editor=me
```

to

```
editor=ed
```

then **vsh** will invoke **ed** the next time you request the **File** command' **Edit** sub-command.

me-disp-attr, **pd-disp-color**, **pd-disp-attr**, and **se-disp-attr** give the display features for, respectively, the menu bar, the menu color, the menu attribute, and the dialogue box.

The lines **pfkey1** through **pfkey9** set the behavior of the function keys. The first seven characters after the equal sign '=' give the text that appears in stub at the bottom of the screen. Everything after the first seven characters describes the command to be executed when you press that function key.

The text that follows the line **cmd=** lists the commands that you have executed with the command **Command**. You can embed the following tokens in a command:

%F Represent the currently highlighted file
%T Represent all tagged files
%D Represent the destination directory

These are used just as they are with the **Command** command, described above.

Command-line Options

vsh recognizes the following options:

-d*directory*
Enter **vsh** and begin to work in *directory*. If no *directory* is named, then begin work in the current directory **vsh** normally begins in the last directory used in your last **vsh** session.

-e Do not use the graphic character set. This option coarsens the appearance of **vsh**, but gives it a fighting chance to run on cheap terminals that do not implement the full alternate character set of the DEC VT-100 terminal.

-i Restrict the user's ability to run the **Install** command. In this mode, **vsh** can be used as a restricted shell, especially if it is embedded in **/etc/passwd**.

-r Restrict the shell. This option turns off the following:

- The command **Command**
- No interactive shell can be called from the **Options** menu
- Most options from the **Directory** menu
- Most options from the **Install** menu

This lets the system administrator restrict the activity of users fairly strongly.

-t This command-line option tells **vsh** to assume the entire VT-100 mapping. This is useful with terminals whose system definitions are incomplete, or the alternate character set is ignored.

Files

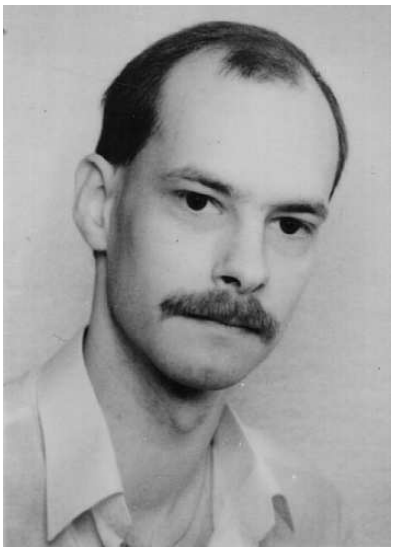
$HOME/.vsh — Configuration file

See Also

commands, ksh, sh, terminfo, ttytype, Using COHERENT

Notes

vsh was written by Udo Munk:



To reach Udo, send e-mail to udo@mwc.com.

vsprintf() — STDIO Function (libc)

Print formatted text into string

```
#include <stdarg.h>
#include <stdio.h>
int
vsprintf(string, format, arguments)
char *string, *format; va_list arguments;
```

vsprintf() constructs a formatted string in the area pointed to by *string*. It translates integers, floating-point numbers, and strings into a variety of text formats. **vsprintf()** can handle a variable list of arguments of various types. It is roughly equivalent to the '%r' conversion specifier to **sprintf()**.

format points to a string that can contain text, character constants, and one or more *conversion specifications*. A conversion specification describes how to convert a particular data type into a particular text format. Each conversion specification is introduced with the percent sign '%'. (To print a literal percent sign, use the escape sequence '%%'.) See **printf()** for further discussion of the conversion specification and for a table of the type specifiers that can be used with **vsprintf()**.

After *format* comes *arguments*. This is of type **va_list**, which is defined in the header file **stdarg.h**. It has been initialized by the macro **va_start()** and points to the base of the list of arguments used by **vsprintf()**. For more information, see **va_arg()**.

arguments should access one argument for each conversion specification in *format* of the type appropriate to the conversion specification. For example, if *format* contains conversion specifications for an **int**, a **long**, and a string, then *arguments* access three arguments, being, respectively, an **int**, a **long**, and a **char** *. If there are fewer arguments than conversion specifications, then **vsprintf()**'s behavior is undefined (and probably unwelcome). If

there are more, **vsprintf()** evaluates and then ignores every argument without a corresponding conversion specification. If an argument is not of the same type as its corresponding type specification, then the behavior of **vsprintf()** is undefined; thus, accessing an **int** where **vsprintf** expects a **char** * may generate unwelcome results.

If it writes the formatted string correctly, **vsprintf()** returns the number of characters written; otherwise, it returns a negative number.

See Also

fprintf(), libc, printf(), sprintf(), vprintf(), vsprintf()
ANSI Standard, §7.9.6.9

Notes

vsprintf() can construct a string up to at least 509 characters long.

vtkb — Device Driver

Non-configurable keyboard driver, virtual consoles

The device driver **vtkb** drives the keyboard on your system's console — that is, the keyboard that is plugged directly into your computer.

This driver recognizes the standard 83-, 101-, and 102-key AT-protocol keyboards, using the keyboard layout used in the United States. These codes are "hard-wired" into the driver. Unlike the other COHERENT keyboard driver, **vtnkb**, you cannot modify these settings.

vtkb is, in general, more robust than **vtnkb** in handling inexpensive keyboards that do not conform fully to accepted standards.

For details on how to select a given keyboard driver, see the Lexicon entry for **keyboard**.

See Also

device drivers, keyboard, vtnkb

vtnkb — Device Driver

Configurable keyboard driver, virtual consoles

The device driver **vtnkb** drives the keyboard on your system's console — that is, the keyboard that is plugged directly into your computer.

Unlike the related driver **vtkb**, **vtnkb** uses a loadable translation table to interpret keystrokes. This permits you to use any number of national keyboard mappings on your COHERENT system without changing the kernel in any way. You can select among any number of configuration programs stored in directory **/conf/kbd**, or you can create your own keymapping table to suit your preferences.

To change the layout and function-key bindings, run one of the keyboard-mapping programs kept in directory **/conf/kbd**. This directory contains the C source code for the mapping tables, as well as a **Makefile** that helps you rebuild the mapping programs. This rest of this article describes the structure of the driver **vtnkb**, and describes how you can write or modify a keyboard-mapping program.

Internal Structure of the Driver

vtnkb understands the following "shift" and "lock" keys:

scroll	Scroll lock
num	Keypad **<num>** lock
caps	**<shift>** or **<caps>** lock
lalt	Left **<alt>** key
ralt	Right **<alt>** key
lshift	Left **<shift>** key
rshift	Right **<shift>** key
lctrl	Left **<ctrl>** key
rctrl	Right **<ctrl>** key
altgr	**<alt graphic>** key (non-U.S. keyboards)

vtnkb records the internal shift state, as defined by the current positions of the shift and lock keys. The shift state is a logical combination of internal states **SHIFT**, **CTRL**, **ALT**, and **ALT_GR**. The **<lshift>** and **<rshift>** keys combine to form the current **SHIFT** state for non-alphabetic keys. Alphabetic keys generally use the current state

of the **<caps-lock>** key plus the left and right **<shift>** keys. Numeric keys on the keypad generally use the state of the **<num lock>** key plus the left and right **<shift>** keys. The left and right **<ctrl>** keys form the internal **CTRL** state. Likewise, the left and right **<alt>** keys form the internal **ALT** state. Note that 102-key keyboards generally replace the right **<alt>** key with the **<altgr>** (alt graphics) key, to allow access to the alternate graphics characters found on some keyboards.

vtnkb lets you configure or read the internal mapping tables via the following requests to **ioctl()**, as defined in header file **<sgtty.h>**:

TIOCGETF	Get function key bindings
TIOCSETF	Set function key bindings
TIOCGETKBT	Get keyboard table bindings
TIOCSETKBT	Set keyboard table bindings

Requests **TIOCGETF** and **TIOCSETF** reference a data structure of type **FNKEY**, which is defined in header file **<sys/kb.h>**. Structure member **k_fnval** is a character array that contains a series of contiguous function key/value bindings; the end of the bindings is marked by manifest constant **DELIM**. You can use any value other than **DELIM** as part of a function-key binding. Structure member **k_nfkeys** indicates how many function keys have associated entries in **k_fnval**. Function keys are numbered from zero through **k_nfkeys**-1.

How To Write a Keyboard Table

The main difference between the keyboard drivers **vtnkb** and **vtkb** is that **vtnkb** uses a "supplemental" process to interpret keystrokes. You can re-construct the interface to the **vtnkb** driver by modifying a keyboard-mapping file and then using a support module to link that file to the driver.

As noted above, directory **/conf/kbd** contains the source code for a series of such supplemental programs. These programs differ from each other only in the keyboard binding or mapping tables each uses; by selecting one such program and linking it into **vtnkb**, you can switch easily from the standard keyboard layout used in the U.S. to any of a number of layouts used in other countries. **/conf/kbd** also contains compiled executables, and a **Makefile** that you use to construct the executables from the corresponding source files.

The keyboard-mapping file is a C program that consists of initialized tables and strings. In addition, several header files provide the scan codes and other constants required for the key tables. This format makes the file easy to edit, and also lets you enter characters in several different formats.

The support module, in turn, performs several tasks. These include scanning the keyboard-mapping file for errors, reformatting the table for use by the device driver, and passing the reformatted table to the driver.

A keyboard-mapping source file consists primarily of three data structures that you must modify to support a given keyboard mapping. The first, and simplest, of the structures is **tbl_name**. This is a character string that describes the keyboard. For example, the stock 101-key U.S. AT keyboard mapping file **/conf/kbd/us.c** initializes this string to:

```
"U.S. AT keyboard table"
```

The second data structure, **kbtbl**, is an array of key-mapping entries. It has one entry (or row) for each possible key location. Each entry in this structure consists of 11 fields, which hold, respectively, the key number, nine possible mapping values, and a mode field. The following example is for physical key location 3 from key-mapping source file **/conf/kbd/belgian.c**:

```
{ K_3, 0x82, '2', none, none, 0x82, '2', '~', none, '~', O|T },
```

Field 1 contains the *scan code set 3* code value for the desired key. Header file **<sys/kbscan.h>** contains manifest constants of the form **K_***nnn* that map the AT keyboard's *physical* key number *nnn* to the corresponding scan code set 3 value generated by the keyboard. In the above example, **K_3** corresponds to key location three.

Fields 2 through 10 contain the key mappings corresponding to the following shift states, as follows:

2 base or unshifted
3 **SHIFT**
4 **CONTROL**
5 **CONTROL+SHIFT**
6 **ALT**
7 **ALT+SHIFT**
8 **ALT+CONTROL**
9 **ALT+CONTROL+SHIFT**
10 **ALT_GRAPHIC**

For "regular" keys, the values for these nine fields are eight-bit characters; for "function" or "shift" keys, they are special values. The symbolic constant **none** indicates that you want no output when the key is pressed in the specified shift state.

In the case of a function key, the value specified is the number of the desired function key. Header file **<sys/kb.h>** defines a set of symbolic constants of the form **f***n*, where *n* is the number of the desired function key. You should use these constants; they will improve the readability of your code, and they will protect your keyboard mapping source files from any future changes in the structure of the keyboard driver.

In the case of a "shift" key, all nine entries must be identical and must consist of one of the following symbolic constants: **scroll**, **num**, **caps**, **lalt**, **ralt**, **lshift**, **rshift**, **lctrl**, **rctrl**, or **altgr**. These are defined in the header file **<sys/kb.h>**. Note that 83-key XT-layout keyboards only have one "control" and "alt" key, so not every shift-key combination may be possible on your target keyboard.

The last (11th) field in the key entry is the "mode" field. The following symbolic constants specify the mode of the current key:

S The specified key is a "shift" or "lock" key. Note that all entries in array **k_val** must be identical for a "shift" or "lock" key to work correctly.

F The specified key is a "function" or special key. The value of all elements of array **k_val** must specify a function key number.

O The specified key is "regular" and requires no special processing.

C The **<caps-lock>** key affects this key.

M Make: generate an interrupt only upon key "make" (i.e., when the key is depressed). This mode is useful for keys that do not repeat. Note that using this mode with a "shift" key stops you from unshifting upon release of the key!

T Typematic: generate an interrupt when the key is depressed, and generate subsequent key-depression interrupts while the key is depressed. The rate at which interrupts are generated is specified by the typematic rate of the keyboard. This type is usually associated with a "regular" key.

MB Make/Break: generate an interrupt when the key is depressed and when it is released. No additional interrupts are generated no matter how long the key is depressed. This mode is used for "shift" keys.

TMB Typematic/Make/Break: generate an interrupt when the key is first depressed; generate subsequent key depression interrupts while the key remains depressed; and generate an interrupt when the key is released.

The above example specifies a mode field of **O|T**, which corresponds to a "regular" key with typematic repeat, and no special handling of the "lock" keys.

The last data structure, **funkey**, consists of an array of function-key initializers, one per function key. The initializers are simple, quoted character strings delimited by either hexadecimal value **0xFF**, octal value **\377**, or symbolic constant **DELIM**. Note that any other value can be used as part of a function-key binding. Function keys are numbered starting at zero.

Function keys are useful not only in the classic sense of the programmable function keys on the keyboard, but also as a general purpose mechanism for binding arbitrary length character sequences to a given key. For example, physical key location 16 is usually associated with the **<tab>** and **<back tab>** on the AT keyboard; and **/conf/kbd/us.c** sets the key mapping table entry for key 16 as follows:

```
{ K_16, f42, f43, none, none, f42, f43, none, none, none, F|T },
```

For traditional reasons, the **<back tab>** key outputs the sequence **<esc>[Z** whereas the **<tab>** key simply outputs the horizontal-tab character **<ctrl-I>**. Because at least one of the mapping values for this key is more than one character long, the key must be defined as a "function" key and all entries for the the key must correspond to function-key numbers. In this example, function key number 42 was chosen for **<tab>**, and function key number 43 was chosen for **<back tab>**. The constant **none** indicates that you want no output when the key is pressed in the specified shift state. The corresponding **funkey** initialization entries for function keys **f42** and **f43** are as follows:

```
/* 42 */      "\t\377",          /* Tab */
/* 43 */      "\033[Z\377",      /* Back Tab */
```

We strongly recommend that you comment your function-key bindings.

You can also change function-key bindings via the command **fnkey**. This command lets you temporarily alter one or more function-key mappings without changing your key-mapping sources.

Examples

Prior to the release of the 101- and 102-key, enhanced-layout AT keyboards, the **<ctrl>** key was positioned to the left of 'A' key. Most terminals also locate the **<ctrl>** key there. The first example shows how to swap the left **<ctrl>** key and the **<caps-lock>** key on a 101- and 102-key keyboard. The **<caps-lock>** key is physical key 30, whereas the left **<ctrl>** key is physical key 58. Their respective entries in file **/conf/kbd/us.c** source file are as follows:

```
{ K_30, caps, caps, caps, caps, caps, caps, caps, caps, caps,  S|M },
{ K_58, lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl, S|MB },
```

Note that the **<caps-lock>** key is defined with mode **M** as it is a "lock" key. The keyboard will interrupt only on key depressions, because releasing a "lock" key has no effect. The left **<ctrl>** key is defined with mode **MB** as it is a "shift" key. The keyboard generates an interrupt on both key depression and key release, because the driver must track the state of this key.

To swap the aforementioned keys, simply change all occurrences of **caps** to **lctrl** and vice-versa, as well as swapping the mode fields. After making the changes, the entries now appear as:

```
{ K_30, lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl, S|MB },
{ K_58, caps, caps, caps, caps, caps, caps, caps, caps, caps,  S|M  },
```

The second example converts a 101- or 102-key keyboard table to support an XT-style 83-key keyboard layout. The following section summarizes the "typical" differences found when comparing the two keyboard layouts. Needless to say, given the extreme variety in keyboard designs, your mileage may vary:

Location	*101/102 Value*	*83-key Value*	*Comments*
14	**none**	Various	Keyboard-specific
30	**caps**	**lctrl**	
58	**lctrl**	**lalt**	
64	**rctrl**	**caps**	
65	**none**	**F2**	Function Key
66	**none**	**F4**	Function Key
67	**none**	**F6**	Function Key
68	**none**	**F8**	Function Key
69	**none**	**F10**	Function Key
70	**none**	**F1**	Function Key
71	**none**	**F3**	Function Key
72	**none**	**F5**	Function Key
73	**none**	**F7**	Function Key
74	**none**	**F9**	Function Key
90	**num**	**Esc**	
95	**/**	**num**	
100	*	**scroll**	
105	**-**	**none**	**<SysReq>** not used
106	**+**	*	
107	**none**	**-**	
108	**<Enter>**	**+**	
110	**esc**	**none**	Not on XT layout
112-123	**F1-F12**	**none**	Not on XT layout
124	**none**	**none**	**<PrtScr>** not used
125	**scroll**	**none**	Not on XT layout
126	**none**	**none**	**<Pause>** not used

Building New Binaries

After you have modified an existing keyboard-mapping table, use the following commands to rebuild the corresponding executables:

```
cd /conf/kbd
su root
make
```

If you have created a new keyboard mapping table, you must edit **/conf/kbd/Makefile**. Duplicate an existing entry from the **Makefile**, and change the duplicated name to match the name of your new keyboard-mapping table. After you have finished your editing, build an executable from your source file by simply executing the above series of commands.

To load your new keyboard table, simply type the name of the executable that corresponds to your keyboard-mapping file. For example, if you just built executable **french** from source file **french.c**, type the following command:

```
/conf/kbd/french
```

If the keyboard-support module finds an error, it will print an appropriate message. If it finds no errors, it will update the internal tables of the **vtnkb** keyboard driver, reprogram the keyboard, and print a message of the form:

```
Loaded French AT keyboard table
```

To ensure that the keyboard-support module is loaded automatically when you boot your COHERENT system, edit file **drvld.all** to name the module you wish to use. For example, to ensure that the French keyboard table is loaded automatically when you boot your system, insert the following command into **/etc/drvld.all**:

```
/conf/kbd/french
```

Disabling <Ctrl><Alt><Del>

By convention, function-key 0, when enabled, causes the computer system to reboot. This function key is usually bound to the key sequence **<ctrl><alt><del>**, but you can disable it by setting the value of driver-variable **KBBOOT** to zero. The installation script for configuring your console asks you if you want to turn off this feature; and if so, it sets **KBBOOT** to the correct value.

Problems With Incompatible Keyboards

If you are experiencing problems with respect to key mappings, and you installed one of the loadable keyboard mapping tables from the keyboard selection menu, you may have an incompatible keyboard. Please note that the COHERENT **vtnkb** driver (and loadable tables) only work with well-engineered keyboards, such as those built by IBM, Cherry, MicroSwitch, or Keytronics; it may not work correctly with a poorly engineered "clone" keyboard.

The preferred action is to replace your keyboard with one made by one of the above-named manufacturers. If, however, you wish to use a "clone" keyboard with COHERENT, your best bet is to re-install COHERENT and select the **vtkb** driver instead of **vtnkb**. **vtkb** is not loadable and supports only the U.S., German, and French keyboard layouts. For details on how to replace **vtnkb** with **vtkb**, see the Lexicon entry for **keyboard**.

See Also

device drivers, keyboard, vtkb

wait — Command

Await completion of background process
wait [*pid***]**

Typing the character '&' after a command tells the shell **sh** to execute it as a *background* (or *detached*) process; otherwise, it is executed as a *foreground* process. You can perform other tasks while a background process is being executed. The shell prints the process id number of each background process when it is invoked. **ps** reports on currently active processes.

The command **wait** tells the shell to suspend execution until the child process with the given *pid* is completed. If no *pid* is given, **wait** suspends execution until all background processes are completed. If the process with the given *pid* is not a child process of the current shell, **wait** returns immediately.

The shell executes **wait** directly.

See Also

commands, ksh, ps, sh

Notes

If a subshell invokes a background process and then terminates, **wait** returns immediately rather than waiting for the termination of the grandchild process.

wait.h — Header File

Define wait routines
#include <sys/wait.h>

Header file **wait.h** declares prototypes for the functions **wait()** and **waitpid()**. It also defines manifest constants used with those functions.

See Also

header files, wait(), waitpid()

wait() — System Call (libc)

Await completion of a child process
#include <sys/wait.h>
wait(*statp***)**
int ******statp***;**

wait() suspends execution of the invoking process until a child process (created with **fork()**) terminates. It returns the process identifier of the terminating child process. If there are no children or if an interrupt occurs, it returns -1.

If it is successful, **wait()** returns the process identifier of the terminated child process. In addition, **wait()** fills in the integer pointed to by *statp* with exit-status information about the completed process. If *statp* is NULL, **wait()** discards the exit-status information.

wait() fills in the low byte of the status-information word with the termination status of the child process. A child process may have terminated because of a signal, because of an exit call, or have stopped execution during **ptrace()**. Termination with **exit()**, which is normal completion, gives status 0. Other terminations give signal values as status (as defined in the article on **signal()**). The **0200** bit of the status code indicates that a core dump was produced. A status of **0177** indicates that the process is waiting for further action from **ptrace()**.

The high byte of the returned status is the low byte of the argument to the **exit()** system call.

If a parent process does not remain in existence long enough to **wait()** on a child process, the child process is adopted by process 1 (the initialization process).

Example

For an example of this system call, see the entry for **msgget()**.

See Also

_exit(), fork(), ksh, libc, ptrace(), signal(), sh, waitpid(), wait.h
POSIX Standard, §3.2.1

waitpid() — System Call (libc)

Wait for a process to terminate
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(*pid***,** *status***,** *flags***)**
pid_t *pid***; int** **status***, flags***;**

waitpid() waits until a given process terminates. *pid* identifies the child process whose termination is awaited. The value of *pid* sets the behavior of **waitpid()**, as follows:

pid>0 Wait for the termination of the child process whose identifier is *pid*.

pid=0 Wait for the termination of any child in the current process group.

pid=-1 Wait for the termination of any child process. In this mode, **waitpid()** behaves the same as the system call **wait()**.

pid<-1 Wait for termination of any child in the group given by *-pid*.

status points to the place where you want **waitpid()** to write the termination status of *pid*.

flags is the logical OR of the following values:

WNOHANG
If *pid* has already terminated, write its termination status into *status*; but if *pid* has not yet terminated, do not wait for it to do so.

WUNTRACED
Report the status of every child process of *pid* that is stopped, and whose status has not been returned since it stopped.

By default, **waitpid()** returns the process identifier of the chid process whose status is being reported, or -1 if something went wrong. If *flags* includes **WNOHANG**, **waitpid()** returns zero if no status information is available.

See Also

libc, wait(), wait.h
POSIX Standard, §3.2.1

wall — Command

Send a message to all logged-in users
/etc/wall

wall types a message to every user currently logged into the COHERENT system, with the exception of the sender. It can be used to inform users of information of general interest; for example, that man has landed on the moon, or that the system is going down in 15 minutes.

wall reads the message to be broadcast from the standard input until end of file. When it sends the message, it prefaces it with the herald "Broadcast message ...", which includes an audible warning. Only the superuser should invoke **/etc/wall** (to override access protections of the target terminals).

Files

/etc/utmp — Current users file
/dev/tty*

See Also

commands, msg, who, write

Diagnostics

The message "Cannot send to *user* on *ttyname*" indicates that **wall** cannot write to the given *user*.

wc — Command

Count words, lines, and characters in text files
wc [-clw] [*file…***]**

wc counts words, lines, and characters in each *file*. If no *file* is given, **wc** uses the standard input. If more than one *file* is given, **wc** also prints a total for all of the files.

wc defines a *word* to be a string of characters surrounded by white space (blanks, tabs, or newlines). It defines the number of lines to be the number of newline characters in the file, plus one.

wc recognizes the following options:

-c Count only characters.

-l Count only lines.

-w Count only words.

The default action is to print all counts.

See Also

commands

welcome — System Administration

Welcome a new user
/etc/default/welcome

The command **login** normally displays the contents of file **$HOME/.lastlogin** when you log in. This file holds the date and time that you last logged into your COHERENT system.

If this file does not exist, **login** assumes that you are logging in for the first time, and executes the script **/etc/default/welcome**. This script displays information about COHERENT, to help welcome you to it and provide you with a "friendly" environment.

For information on what this file does, you should read it. If you wish, you can modify this file to suit the layout and special features of your system.

See Also

Administering COHERENT, login, newusr

whence — Command

List a command's type
whence [-v] *command ...*

The command **whence** is built into the Korn shell **ksh**. It lists the type for each *command*. Option **-v** lists function and alias values as well.

See Also

commands, ksh

whereis — Command

Locate source, binary, and manual files
whereis [-bmrsu] [-BMS *dir ...* **-f]** *name ...*

The command **whereis** locates source files, binary files (executables), and manual pages (documentation) that match a given *name*. Prior to searching, **whereis** strips *name* of any path information, extensions, and the **s.** prefix.

By default, **whereis** searches the following directories:

Sources	*Binaries*	*Manual Pages*
/usr/src/cmd	/bin	/usr/man/*
/usr/src/games	/usr/bin	
/usr/src/local	/usr/games	
/usr/src/alien	/usr/local	
/usr/include	/etc	
/usr/include/sys	/lib	
	/usr/lib	

Options

whereis recognizes the following command-line options:

-b Search only for binary files.

-B Use the directory list specified by *dir* instead of the default directory list for binary files.

-f Terminate the directory list introduced by options **-B**, **-M**, or **-S**, and treat any additional command-line arguments as file names to be searched for.

-m Search only for manual pages (documentation files).

-M Use the directory list specified by *dir* instead of the default directory list for manual pages.

-r Search recursively downward from the directories specified by *dir* or from the default directories. This option is useful when the searched directories contain sub-directories. By default, **whereis** searches only the directories specified or the default directories.

-s Search only for source files.

-S Use the directory list specified by *dir* instead of the default directory list for source files.

-u Search for "unusual" files. A file is said to be unusual if it does not have one entry for each of the three search categories.

Please note that if you use options **-B**, **-S**, or **-M**, you must use the **-f** option to terminate the directory list specified by *dir*.

Example

The following example finds all commands in directory **bin** that have either no corresponding source code in directory **src** or no corresponding documentation in directory **doc**:

```
whereis -u -M doc -S src -B bin -f bin/*
```

See Also

commands, find, qfind, which

Notes

whereis is distributed as a service to COHERENT customers, as is. It is not supported by Mark Williams Company. *Caveat utilitor.*

which — Command

Locate executable files
which *command ...*

which displays the full path name associated with *command*. It searches the directories named by environment variable **PATH** for the first executable that matches *command* and that you have permission to execute. If **which** can find no executable that matches your request, an error message is displayed.

Example

The following example displays the path names that correspond to commands **write**, **vi**, **myprog**, and **fsck**:

```
which write vi myprog fsck
```

See Also

commands, find, PATH, qfind, whereis

while — Command

Execute commands repeatedly
while *sequence1* **[do** *sequence2***] done**

The shell construct **while** controls a loop. It first executes the commands in *sequence1*. If the exit status is zero, the shell executes the commands in the optional *sequence2* and repeats the process until the exit status of *sequence1* is nonzero. Because the shell recognizes a reserved word only as the unquoted first word of a command, both **do** and **done** must occur unquoted at the start of a line or preceded by ';'.

The shell commands **break** and **continue** may be used to alter control flow within a **while** loop. The **until** construct has the same form as **while**, but the sense of the test is reversed.

The shell executes **while** directly.

See Also

break, commands, continue, ksh, sh, test, until

while — C Keyword

Introduce a loop
while(*condition***)**

while is a C keyword that introduces a conditional loop. *condition* is tested on reiteration of the loop, and the loop ends when *condition* is no longer satisfied. For example,

```
while (foo < 10)
```

introduces a loop that will continue until the variable **foo** is reset to ten or greater. Note that the statement

```
while (1)
```

will loop forever, unless interrupted by a **break**, **goto**, or **return** statement.

See Also

break, C keywords, continue, do, for
ANSI Standard, §6.6.5.1

who — Command

Print who is logged in
who [*file***] [am i]**

The command **who** prints the names of the users who are logged in to the system. For each user, **who** prints her name, terminal name, login date, and login time. The form **who am i** prints this information only about yourself.

If *file* is specified, **who** uses it instead of **/etc/utmp** to obtain information about who is logged in. This is useful, for example, with the file **/usr/adm/wtmp**, which contains a continuous record of logins, logouts and reboots. When *file* is specified, **who** displays logouts; otherwise, they are suppressed.

Files

/etc/utmp — To get user information

See Also

ac, commands, sa

wildcards — Definition

Wildcards are characters that, in some circumstances, can represent a range of ASCII characters. Another name for them is "metacharacters". The wildcards available under COHERENT are as follows:

? Match any one character.

* Match any number of characters, or no characters at all.

[ ] A set of characters enclosed between '[' and ']' match any one character of the set. Sets of characters may include ranges, such as **[a-z]** for all lower-case letters or **[0-9]** for all numerals.

[!] A set of characters enclosed between '[!' and ']' match any one character *except* one of the set. Sets of characters may include ranges, such as **[a-z]** for all lower-case letters or **[0-9]** for all numerals. For example, the command

```
ls [!abc]*
```

prints the names of all files *except those that begin with* **a**, **b**, *or* **c**.

**** Ignore the special meaning of a wildcard.

See Also

egrep, pattern, pnmatch(), Using COHERENT

write — Command

Converse with another user
write *user* **[** *tty* **]**

The COHERENT system provides several commands that allow users to communicate with each other. **write** allows two logged-in users to have an extended, interactive conversation.

write initiates a conversation with *user*. If *tty* is given, **write** looks for the *user* on that terminal; this is useful if a user is marked as being logged in on more than one device. Otherwise, **write** holds the conversation with the first instance of *user* found on any tty.

If found, **write** notifies *user* that you are beginning a conversation with him. All subsequent lines typed into **write** are forwarded to the *user*'s terminal, except lines beginning with '!', which are sent to the shell **sh**. Typing end of file (usually **<ctrl-D>**) terminates **write** and sends *user* the message "EOT" to tell him that communication has ended.

Two users typing lines to **write** at about the same time can cause extreme confusion, so users should agree on a protocol to limit when each is typing. The following protocol is suggested. One user initiates a **write** to another, and waits until the other user replies before beginning. The first user then types until he wishes a reply and suffixes "o" (over) to indicate he is through. The other user does the same, and the conversation alternates until one user wishes to terminate it. This user types "oo" (over and out). The other user replies in the same way, indicating he too is finished. Finally each of the users leave **write** by typing end-of-file (usually **<ctrl-D>**).

Any user may deny others the permission to **write** to his terminal by using the command **mesg**.

Files

/etc/utmp
/dev/*

See Also

commands, mail, mesg, msg, sh, wall, who

Notes

You should use **write** only for extended conversations. Use **msg** to send brief communications to a logged in user, and **mail** to communicate with a user who is not logged in. **wall** broadcasts a message to all logged in users.

write() — System Call (libc)

Write to a file
#include <unistd.h>
int write(*fd***,** *buffer***,** *n***)**
int *fd***; char** **buffer***; **int** n***;**

write() writes *n* bytes of data, beginning at address *buffer*, into the file associated with the file descriptor *fd*. Writing begins at the current write position, as set by the last call to either **write()** or **lseek()**. **write()** advances the position of the file pointer by the number of characters written.

Example

For an example of how to use this function, see the entry for **open()**.

See Also

libc, unistd.h
POSIX Standard, §6.4.2

Diagnostics

write() returns -1 if an error occurred before the **write()** operation commenced, such as a bad file descriptor *fd* or invalid *buffer* pointer. Otherwise, it returns the number of bytes written. It should be considered an error if this number is not the same as *n*.

Notes

write() is a low-level call that passes data directly to COHERENT. Do not use it with the STDIO routines **fread()**, **fwrite()**, **fputs()**, or **fprintf()**.

wtmp — System Administration

File that records past login events
/usr/adm/wtmp

File **/usr/adm/wtmp** records every login event that has concluded — that is, the user has logged in and logged out again. You can comb this file to trace which user have logged onto your system, and when.

wtmp records each active login event as a record of type **utmp**, which is defined in header file **<utmp>**. For details, see the Lexicon entry **utmp.h**.

See Also

Adminstering COHERENT, utmp utmp.h

xargs — Command

Execute a command with many arguments
xargs *command argument ... argument*

COHERENT limits the amount of memory available to hold a command's arguments; therefore, a command will fail if its list of arguments exceeds this limit. This limit is set by the constant **BUFSIZ**, which is defined in the header file **stdio.h**.

To avoid this problem, COHERENT offers the command **xargs**. This command executes *command* and passes to it every *argument*. An *argument* can be an option to *command*, the name of a file, or anything else that *commands* expects. **xargs** then redirects the standard input into *command*. **xargs** is careful not to exceed the system-imposed limit, which is expected to be greater than **BUFSIZ**. It continues to execute *command* with the read-in arguments until it reaches end-of-file.

See Also

commands, exec, execution

Notes

The COHERENT implementation of **xargs** performs only the most basic — and most important — behaviors of **xargs**. You must rewrite all scripts that depend upon the more exotic behaviors of the System-V implementation of **xargs**.

xgcd() — Multiple-Precision Mathematics (libmp)

Extended greatest-common-divisor function
#include <mprec.h>
void xgcd(*a, b, r, s, g***)**
mint **a, *b, *r, *s, *g;*

xgcd() is an extended version of the greatest-common-division function. It sets the multiple-precision integer (or **mint**) pointed to by *g* to the greatest common divisor of the **mint** pointed to by *a* and that pointed to by *b*. It also sets the **mint**s pointed to by *r* and *s* so the following relation holds:

$$g = a \times r + b \times s$$

r, *s*, and *g* must all be distinct.

See Also

libmp

yacc — Command

Parser generator
yacc [*option ...***]** *file*
cc y.tab.c [-ly]

Many programs process highly structured input according to given rules. Compilers are a familiar example. Two of the most complicated parts of such programs are *lexical analysis* and *parsing* (sometimes called *syntax analysis*). The COHERENT system includes two powerful tools called **lex** and **yacc** to assist you in performing these tasks. **lex** takes a set of lexical rules and writes a lexical analyzer, whereas **yacc** takes a set of parsing rules and writes a parser; both output C source code that can be compiled into a full program.

The term *yacc* is an acronym for "yet another compiler-compiler". In brief, the **yacc** input *file* describes a context free grammar using a BNF-like syntax. The output is a file **y.tab.c**; it contains the definition of a C function **yyparse**(), which parses the language described in *file*. The output is ready for processing by the C compiler **cc**. Ambiguities in the grammar are reported to the user, but resolved automatically by precedence rules. The user must provide a lexical scanner **yylex**(), which you may generate with **lex**. The **yacc** library includes default definitions of **main**, **yylex**, and **yyerror**, and may be included with the option **-ly** on the **cc** command line.

yacc recognizes the following options:

-d Enable debugging output; implies **-v.**

-hdr *headerfile*
Put the header output in *headerfile* instead of **y.tab.h**.

-items *N*
Allow *N* items per state. This option is designed to help **yacc** users deal with the ANSI C grammar.

-l *listfile*
Place a description of the state machine, tokens, parsing actions, and statistics in file *listfile*.

-sprod *N*
Allow *N* symbols per production; default, 20. This option is designed to help **yacc** users deal with the ANSI C grammar.

-st Print statistics on the standard output.

-v Verbose option. Like **-l**, but places the listing in file **y.output** by default.

The following options are useful if table overflow messages appear:

-nterms *N*
Allow for *N* nonterminals; default, 100.

-prods *N*
Allow for *N* productions (rules); default, 350.

-states *N*
Allow for *N* states; default, 300.

-terms *N*
Allow for *N* terminal symbols; default 100.

-types *N*
Allow for *N* types; default, ten.

Files

y.tab.c — C source output
y.tab.h — Default C header output
y.output — Default listing output
/lib/yyparse.c — Protoparser
/tmp/y[ao]* — Temporaries
/usr/lib/liby.a — Library

See Also

cc, commands, lex, Programming COHERENT
Introduction to yacc, Yet Another Compiler-Compiler

Diagnostics

yacc writes onto the standard error the number of R/R (reduce/reduce) and S/R (shift/reduce) conflicts (ambiguities).

Notes

The version of **yacc** shipped prior to release 4.2 of COHERENT included the header file **<action.h>** in its output. This file's data are now built into parser skeleton in **/lib/yyparse**, thus obviating **<action.h>**. This header has been dropped from COHERENT. You should re-run **yacc** to update the source files generated by previous versions of **yacc**.

yes — Command

Print infinitely many responses
yes [*string* **]**

With no argument, **yes** prints the string **y\n** forever. If a *string* is named on the command line, then **yes** prints it forever.

Example

The following example scribbles the string **foo\n** over a high-density, 5.25-inch floppy disk in drive 0 (drive A):

```
yes foo >/dev/fha0
```

See Also

commands

zcat — Command

Concatenate a compressed file
zcat [*file***[.Z|.gz]** ... **]**

zcat uncompresses each *file* "on the fly," and prints the uncompressed text onto the standard output. Each *file* must have been compressed by the command **compress** and have the suffix **.Z**, or by the command **gzip** and have the suffix **.gz**.

If the command line names no *file*, **zcat** uncompresses matter read from the standard input.

Example

zcat is useful for extracting selected items from archives; it spares you the overhead of having to uncompress the entire archive just to get at one or two files. For example, to extract **myfile** from the compressed archive **backup.tar.Z**, use the following command line:

```
zcat backup.tar.Z | tar xvf - myfile
```

See Also

commands, compress, gzip, ram, uncompress

zcmp — Command

Compare compressed files
zcmp [-ls] *file1***[.gz]** *file2***[.gz] [***skip1 skip2***]**

zcmp compares two compressed files in a byte-by-byte fashion. It behaves exactly the same as **cmp**, except that it de-compresses compressed files "on the fly." For details on the options to **zcmp** see the Lexicon entry for **cmp**.

See Also

cmp, commands, gzip, zdiff

zdiff — Command

Compare two compressed files
zdiff [-bdefh] [-c *symbol***]** *file1 file2*

zdiff compares two compressed text files, and outputs a summary of their differences. It behaves exactly the same as **diff**, except that it de-compresses compressed files "on the fly." For details on the options to **zdiff** see the Lexicon entry for **diff**.

See Also

commands, diff, gzip, zcmp

zerop() — Multiple-Precision Mathematics (libmp)

Indicate if multi-precision integer is zero
#include <mprec.h>
int zerop(*a***)**
mint **a***;*

zerop() returns true if the multiple-precision integer (or **mint**) pointed to by *a* is zero; otherwise, it returns false.

See Also

libmp

zforce — Command

Force the suffix .gz onto every gzip file
zforce [*file* **...]**

The command **zforce** examines each *file*, and adds the suffix **.gz** to it if it had been compressed with **gzip**. If adding **.gz** would make the file's name longer than 14 characters, **zforce** truncates the file's original name to make room for the suffix.

You should use **zforce** to prompt name compressed files, to ensure that **gzip** does not compress a file twice. You can also **zforce** can be used to examine files whose names were truncated during file transfer, and properly stamp those that were compressed.

See Also

gzip, commands

zgrep — Command

Search compressed files for a regular expression
zgrep [-abcefhilnsvxy] [*pattern***] [***file ...***]**

The command **zgrep** searches for a string within a file that had been compressed by **gzip**. It behaves exactly like **grep**, except that it de-compresses compressed files "on the fly." For details on the options to **zgrep** see the Lexicon entry for **grep**.

See Also

commands, grep, gzip

zip — Command

Zip files into a compressed archive
zip [-*options***] [-b** *path***] [-t** *mmddyy***]** *zipfile file ...* **[-x** *file ...***]**

The command **zip** compresses and archives one or more files. It resembles the program **pkzip** which is widely used under MS-DOS.

zip recognizes the following command-line options:

-b *pathname*
: Write temporary files into directory *pathname*.

-c Add one-line comments to the archive.

-d Delete each *file* from *zipfile*.

-e Encrypt the zipfile. **zip** prompts you for the encryption key.

-ee Verify the encryption key.

-f "Freshen" the contents of *zipfile*: replace the files with the files on disk, but only if the file on disk is newer than that in *zipfile*.

-g "Grow" *zipfile*: that is, append files onto it.

-h Display a help message.

-i Only implode the files.

-j "Junk" (that is, do not record) directory names.

-k Mimic a PKZIP-made zip file.

-l Show the software license.

-m Delete each *file* from *zipfile*.

-n Do not compress special suffixes.

-o Make *zipfile* as old as latest entry.

-q Operate quietly.

-r Recurse — that is, if a *file* is a directory, manipulate its files and those in all of its subdirectories.

-s Only compress the files — do not archive them.

-t Manipulate only the files updated since *mmddyy*.

-u Update: manipulate only changed or new files.

-x Exclude each *file* from those manipulated.

-z Add a zipfile comment.

-0 Use level-0 compression. This compress faster.

-9 Use level-9 compression. This compresses smaller.

The default action is to add or replace each *file*. The file '—' names the standard input.

See Also

commands, compress, gunzip, gzip, unzip

Notes

Do not confuse this command with **gzip**.

zmore — Command

Display compressed text one page at a time
zmore [-cdflsu] [-*window_size* **] [+***line_number* **] [+/***pattern* **] [** *file ...* **] [-]**

The command **zmore** is a filter for paging through text one screenful at a time. *file* is a text file; the operator **-** tells **more** to read and display the standard input.

Unlike the command **more**, **zmore** can display the contents of compressed files. It works on files compressed with the commands **compress** or **gzip**, as well as on files that are uncompressed. If it cannot find *file*, **zmore** looks for a file of the same name that has any of the suffices **.gz**, **.z**, or **.Z**.

zmore recognizes the same command-line options as **more**, and recognizes the same commands. For details, see the Lexicon entry for **more**.

See Also

commands, gzip, more

znew — Command

Recompress .Z files to .gz files
znew [-ftv9PK] [*file***.Z ...]**

The command **znew** recompresses files from **.Z** (**compress**) format to **.gz** (**gzip**) format.

znew recognizes the following command-line options:

-9 Use the slowest, most thorough compression method.

-f Force recompression of *file* even if *file***.gz** already exists.

-K Keep a **.Z** file when it is smaller than the **.gz** file.

-P Use pipes for the conversion to reduce disk space usage.

-t Test the new files before deleting the originals.

-v Verbose mode: display the name and percent by which the size of each recompressed is reduced.

See Also

commands, gzip

Notes

To recompress a file already in **gzip** format, rename the file to replace the suffix **.gz** with the suffix **.Z**, and then invoke **znew**.

znew does not maintain the time stamp if you invoke it with command-line option **-P**.

Index

 # to _

B

C

D

F

G

J

K

L

M

O

P

Q

R

S

T

U

V

W

X

Y

Z

User Evaluation Report

To keep this manual free of errors and to help us improve COHERENT, please sent us your reactions. Please fill in the form below, detach it, and mail it to:

Mark Williams Company
60 Revere Drive
Northbrook, IL 60062

Name: _____

Company: _____

Address: _____

City/State/Zip: _____

Phone: _____

Date: _____

Version and hardware used:

Did you find any errors in the manual?

Can you suggest any improvements to the manual?

Did you find any bugs in the software?

Can you suggest any improvements or enhancements to the software?

Additional comments: