# A/UX Command Reference

Section 1(G–P)

Release 3.0

### Limited Warranty on Media and Replacement

If you discover physical defects in the manuals distributed with an Apple product or in the media on which a software product is distributed, Apple will replace the media or manuals at no charge to you, provided you return the item to be replaced with proof of purchase to Apple or an authorized Apple dealer during the 90-day period after you purchased the software. In addition, Apple will replace damaged software media and manuals for as long as the software product is included in Apple's Media Exchange Program. While not an upgrade or update method, this program offers additional protection for up to two years or more from the date of your original purchase. See your authorized Apple dealer for program coverage and details. In some countries the replacement period may be different; check with your authorized Apple dealer.

**ALL IMPLIED WARRANTIES ON THE MEDIA AND MANUALS, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.**

Even though Apple has tested the software and reviewed the documentation, **APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS, OR IMPLIED, WITH RESPECT TO SOFTWARE, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS SOFTWARE IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE.**

**IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT IN THE SOFTWARE OR ITS DOCUMENTATION,** even if advised of the possibility of such damages. In particular, Apple shall have no liability for any programs or data stored in or used with Apple products, including the costs of recovering such programs or data.

**THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS, OR IMPLIED.** No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Apple Computer, Inc.



Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014-6299
(408) 996-1010



Simultaneously published in the United States and Canada.

# A/UX Command Reference

**Contents**

# About This Manual

This manual is one of three primary manuals in the set of A/UX reference manuals. *A/UX Command Reference*, *A/UX Programmer's Reference*, and *A/UX System Administrator's Reference* contain information about most of the provisions of A/UX, such as its commands, its library routines, its system calls, and its file formats.

These reference manuals constitute a compact encyclopedia of A/UX information. As in an encyclopedia, the information is subdivided into subdocuments, or "manual pages." The information in each manual-page subdocument adheres to a distinctive presentation format. For example, information about command syntax is consistently presented under the heading "Synopsis." (This format is described in detail later in this preface.)

Because most of us need occasional reminders regarding the order and kind of arguments that can accompany a command, the information in the "Synopsis" and "Arguments" sections is presented for use by users at all levels. However, the information in the "Description" section is often written for more advanced users; novices most likely will not be able to learn about the provisions of A/UX from these reference manuals alone.

Because these reference manuals are not intended to be tutorials or learning guides, they should not be the first A/UX books you read. If you are new to A/UX or are unfamiliar with a specific functional area (such as the Macintosh Finder), you should first read *A/UX Essentials* and the other A/UX user guides. After you have worked with A/UX, the reference manuals can help you understand new features or refresh your memory about features you already know.

## Manual pages: a standard for presenting information

The headings conventionally used in the manual pages have virtually become an industry standard for reference documents. Furthermore, the way that this large collection of subdocuments is conventionally organized into sections and books is also something of a standard.

Despite the standardization, locating specific information within this large body of documentation can often be difficult. First you must locate the correct manual page. Once you have the correct manual page, you can usually go

directly to the correct subsection.

To help you locate information, you should read the next section, which explains several means of finding the information you need.

To help you learn to use these books more effectively, other sections in this preface describe the presentation standards that are being used. Some of these are organizational standards that apply at the book and section level. Other conventions and content standards apply within the scope of each manual page, such as the use of standard subheadings and the conventional use of certain fonts and text styles.

Note that the most durable standards have been the standards that apply to the organization and primary headings of each manual page. Of course there are areas in which the A/UX reference books are exceptional, particularly in their more regular use of headings. These books also deviate from industry standards in a few typographic and style areas, which are described later in this preface. For example, the Courier font is used consistently to represent text that is displayed in a terminal window or entered as part of a command line. Other UNIX® books often use boldface type to represent such text.

There has been more instability with respect to how the manual pages are collected into sections and books. For more detailed information, see ''Previous Organization of Sections into Books'' later in this preface.

## Locating information in the reference manuals

You can locate information in the reference manuals by using one of the following tools:

- Table of contents. Each reference manual contains one general table of contents for the entire manual. Located at the beginning of each new section of manual pages is a detailed table of contents. (If a section must span from one binder to another, a tailored table of contents is provided for each of the subdivisions.) The general table of contents lists the sections covered in the complete manual. The detailed table of contents lists the manual pages contained within one section (or section subdivision) along with a brief description of the A/UX provision that is covered in each manual page.
- Query commands. The `man`, `whatis`, and `apropos` commands display on-screen all the information contained in a manual page or just the information in the ''Name'' section of one or more manual pages that

satisfy a search criterion. The next sections tells you how to use the on-line versions of the manual pages.

- *A/UX Reference Summary and Index.* This separate manual is considered part of the A/UX set of reference manuals, but it is not a "standard" resource like the other reference materials. Its primary purpose is to help you locate the correct manual page to refer to in other books. From its summaries, you might also occasionally find all the information you required. It contains the following subsections:
  - "Commands by function." This subsection classifies the A/UX user and system administrator commands by the general or most important function each performs. The summary gives you a broader view of the commands that are available and the context in which each is often used. Each command is mentioned just once in this listing.
  - "Command synopses." This subsection is a compact collection of syntax descriptions for all of the commands in *A/UX Command Reference* and in *A/UX System Administrator's Reference*. It may help you find the syntax of commands more quickly when the syntax is all you need.
  - "Index." The index lists key terms associated with A/UX subroutines and commands. These key terms can help you locate the manual page you need when you don't know if such a keyword-related command or subroutine exists.

The index provided in *A/UX Reference Summary and Index* is designed to be more compact and easier to use than the more industry-standard permuted index, which indiscriminately indexes manual pages under each of the words found in their "Name" sections.

The manual pages listed in the index portion *A/UX Reference Summary and Index* are indexed under more than one entry; for example, `lorder`(1) is included under "archive files," "sorting," and "cross-references." By using this type of index, you are more likely to find the reference you are looking for on the first try.

## Using the on-line documentation

In addition to the paper documentation in the reference manuals, A/UX provides several ways to search and read the contents of each manual page from your A/UX system. An advantage to the on-line version of the documentation is that the computer performs the work of filtering out (or skipping) all the manual

pages other than the one you specifically queried. The only prerequisite is that you already know its name (or a proper search string). However, you don't have to know how manual pages are organized by section numbers and by book titles.

To display a manual page on your screen, enter the `man` command followed by the name of the manual page you want to see. For example, to display the manual page for the `cat` command, including its description, syntax, options, and other pertinent information, you would enter

```
man cat
```

After the first screen of the text of a manual page appears, you can display subsequent screens of the text with each press of the SPACE BAR, until you reach the end of the man page. To display subsequent text one line at a time, press RETURN instead of the SPACE BAR. By pressing Q, you can quit the `man` command before viewing all of the manual page.

To display the descriptive information in the ''Name'' section of any manual page, enter the `whatis` command followed by the name of the provision you want described. In the following example, the command prompt is the percent sign, and the provision that is being queried is the `ls` command:

```
% whatis ls
ls(1)              - lists the contents of a directory
% 
```

To display a list of all manual pages whose ''Name'' sections contain a given keyword or string, enter the `apropos` command followed by a search word or search string enclosed in double quote characters. The names of A/UX provisions are listed on separate lines along with the descriptive information in the ''Name'' section of the manual page that describes those provisions. Sometimes several A/UX provisions are listed on the same line. In those cases, several A/UX provisions are described on a single manual page. You can tell which of these names is the formal name for the manual page because it will be followed by parentheses and an enclosed section number. In the following example, the command prompt is the percent sign, and the A/UX provisions that are queried are those which are described in manual pages whose ''Name'' section contains the word ''tape'':

```
% apropos tape
mt(1)           - magnetic tape manipulating program
frec(1M)        - recover files from a backup tape
mtio(7)         - interface conventions for magnetic tape devices
tc(7)           - Apple Tape Backup 40SC device driver
% ■
```

These documentation query commands are described more fully in the manual pages `man`(1), `whatis`(1), and `apropos`(1) in *A/UX Command Reference*.

## Book- and section-level presentation standards

Customarily, three books are used to house three collections of manual pages that are of concern to three different audiences:

- *A/UX Command Reference* is intended for users with normal file and device access privileges.
- *A/UX System Administrator's Reference* is intended for system administrators or equivalent users with unlimited device and file access privileges.
- *A/UX Programmer's Reference* is intended for programmers.

These books are further divided into sections, each of which contains a set of manual pages in alphabetical order. The standard sections and the audiences they serve are as follows:

- For users with normal access privileges, Section 1 and Section 6 describe utility and game commands.
- For users with unlimited access privileges, Section 1M and Section 8 describe system maintenance commands.
- For programmers, Section 2 describes system calls, Section 3 describes library routines, Section 4 describes file formats, Section 5 describes miscellaneous A/UX provisions, and Section 7 describes drivers and interfaces for devices.

While most of the manual pages describe an A/UX provision of some sort, there is one important exception per section: The first manual page in Sections 1, 1M, 2, 3, 4, 5, 6, 7 and 8 has the same name, `intro`. The `intro` manual pages do not describe a command or other provision of A/UX. Instead, they serve as an introduction to the rest of the manual pages in the section, providing section-

specific information and conventions. (These section-introduction manual pages are also exceptions in terms of the normal alphabetical arrangement of manual pages inside sections.)

For example, the manual page `intro`(2) introduces you to return values and provides an exhaustive list of error code values and their associated error strings. In the rest of the Section 2 manual pages, the error codes are mentioned briefly or merely listed, without detailed explanations.

More advanced readers will probably have occasion to use more than one of the reference manuals. For example, manual pages in the *A/UX Programmer's Reference* frequently make references to manual pages in sections contained in the other two primary reference manuals.

More information about the organization of the reference books is given later in this preface in "Current Organization of Sections into Books."

## How manual-page information is presented

The name of the manual page normally appears in both upper corners of each physical page. Some manual pages describe several routines or commands. For example, `chown` and `chgrp` are both described in a manual page with the primary name `chown`(1) at the upper corners. If you turn to the page `chgrp`(1), you find a reference to `chown`(1). (These cross-reference pages are included only in *A/UX Command Reference* and *A/UX System Administrator's Reference*.) However, if you enter the command `man chgrp`, the extended-coverage `chown`(1) manual page is displayed automatically.

All of the manual pages have a common format that uses the following subheadings. For the most part, the same kind of information appears under each of these subheadings. However, for manual pages that describe different kinds of A/UX provisions, the information under the same heading may differ. So, for example, the heading "Synopsis" contains syntax illustrations for Sections 1, 1M, and 8, but contains C declaration statements for Sections 2 and 3.

**NAME**
This section lists the names of the commands, programming routines, or other A/UX provisions that are described in the manual page. A succinct statement of their purpose is also provided.

**SYNOPSIS**
This section provides the syntax of a command or the data-type declarations associated with a programming routine.

**ARGUMENTS**
This section lists and describes the command options and arguments that can follow the command name on the command line.

**DESCRIPTION**
This section describes in detail the usage of a particular command or programming provision.

**EXAMPLES**
This section offers representative command lines that illustrate various uses of a command.

**STATUS MESSAGES AND VALUES**
This section describes possible error outcomes and, when applicable, possible success outcomes. For commands, exit values are not usually described if the command produces the customary zero exit value for success and a nonzero exit value for failure. For programming routines, the return value from a function is often an indication of completion status. In such cases, the return value is normally discussed in the ''Description'' section as well as in this section.

**WARNINGS**
This section describes possible usage scenarios that can damage the file system or file integrity or that produce results you would not normally anticipate.

**LIMITATIONS**
This section describes how the performance of a command or routine could become unreliable, or areas of functionality that an A/UX provision does not address.

**NOTES**
This section provides miscellaneous information regarding a command or routine, such as author or copyright information.

**FILES**
This section lists any files needed by the command, along with a brief description that identifies it as a file, directory, or link.

**SEE ALSO**
This section provides a list of references to related information.

## Visual conventions for the A/UX reference manuals

A/UX books follow specific styling conventions. For example, words that require special emphasis appear in specific fonts or styles. This section describes

the conventions used in all the A/UX reference books.

## Keys and key combinations

Certain keys on the keyboard have special names. These modifier and character keys, often used in combination with other keys, perform various functions. In this book, the names of these keys appear in the format of an initial capital letter followed by small capital letters.

Here is a list of the most common key names:

| | | |
|---|---|---|
| CAPS LOCK | ENTER | SHIFT |
| COMMAND | ESCAPE | SPACE BAR |
| CONTROL | OPTION | TAB |
| DELETE | RETURN | |

Sometimes two or more key names are joined by hyphens. The hyphens indicate that you press these keys simultaneously to perform a specific function. For example,

Press CONTROL-K

means "While holding down the CONTROL key, press the K key."

## Terminology

In A/UX manuals, a certain term can represent a specific set of actions. For example, the word "enter" indicates that you type a series of characters, then press the RETURN key. The instruction

Enter `whoami`.

means "Type `whoami`, then press the RETURN key." (If you entered this text at a command prompt, the system would respond by displaying your current account name.)

Here is a list of common terms and their corresponding actions.

| Term | Action |
|---|---|
| Click | Press and then immediately release the mouse button. |
| Choose | Activate a command that appears in a menu. To choose a command from a pull-down menu, position the pointer on the menu title and, while holding down the mouse button, slide the mouse toward you until the command is highlighted. Then release the mouse button. |
| Drag | Position the pointer on an icon, press and hold down the mouse button while moving the mouse so that the icon moves to the desired position, and then release the mouse button. |
| Enter | Type the series of characters indicated, then press the RETURN key. |
| Press | Press one key only. (Do not press the RETURN key afterward.) |
| Select | To select an icon, position the mouse pointer on the item, then click (see "Click," above). To select text, use a drag-style operation (see "Drag," above). When selecting a range of text, the drag operation highlights the text from the starting point over and across lines to the final position of the pointer when the mouse button was released. |
| Type | Type the series of characters indicated, without pressing the RETURN key afterward. |

## The Courier font

Throughout the A/UX reference manuals, words that appear on the screen or that you must type exactly as shown are in the Courier font.

Here's an example:

Type `date` on the command line and press RETURN.

This instruction means that you should type the word "date" exactly as shown, then press the RETURN key.

After you press RETURN, text such as this will appear on the screen:

```
Fri Nov  1 11:15:43 PST 1991
```

In this case, the Courier font is used to represent exactly what appears on the screen.

All A/UX manual page names are shown in the Courier font. For example, `ls`(1) indicates that `ls` is the name of a manual page that occurs in Section 1. More information about the use of the Courier font in manual pages is given in "Styling of A/UX Command Elements" and in "Styling of Cross-References to Manual Pages" later in this preface.

## Font styles

Italics are used to indicate that a word or set of words is a placeholder for part of a command line. Here is a sample command syntax illustration:

`cat` *file*

The italicized term *file* is a placeholder for the name of a file. If you wanted to display the contents of a file named `Elvis`, you would type the filename `Elvis` in place of *file*. In other words, you would enter

```
cat Elvis
```

## Styling of A/UX command elements

A/UX commands are entered in accordance with their command syntax. A typical A/UX command line includes the command name first, followed by options and arguments. For example, here is an illustration of the syntax for the `wc` command:

`wc [-l] [-w]` *file...*

In this syntax illustration, `wc` is the command, `-l` and `-w` are options, and *file* is an argument.

A "command option" modifies the action of a command, often by changing its mode of operation (such as read mode or write mode).

An ''argument'' is any element that follows the command name. Command arguments other than command options typically specify the objects upon which the command should act. You often supply the names of files that you want a command to process, so *file* is frequently the last element in syntax illustrations.

Brackets and ellipsis characters in a syntax illustration should be considered part of a syntax notation. This is represented by the use of body font instead of Courier for these characters. Their font treatment tells you that you are not supposed to type these characters as part of the command line. Their meaning as a syntax notation is described next.

The brackets enclose an optional item or a group of optional items. If an optional item has constituent parts that are also optional, these parts are themselves enclosed in brackets, as in this syntax illustration:

```
lpr [-i [numcols]]
```

This syntax illustration shows that the indent (`-i`) command option can be followed by the number of columns to indent the printed page. It also shows that you can omit the number of columns; if you do, the `lpr` command uses the default indent value.

An ellipsis (...) follows an argument that can be repeated any number of times on a command line. If the ellipsis follows a bracketed group of items, the group of items can be repeated any number of times on the command line.

When command options are mutually exclusive, they cannot both be specified at the same time. In such cases, more than one syntax illustration is usually provided:

```
pax -r[other-option-for-archive-reading]...
pax -w[other-option-for-archive-writing]...
```

Outside of syntax illustrations, command options are shown with a leading hyphen also in the Courier font. When you supply multiple command options in an actual command line, only one leading hyphen is normally required. For example the following command line contains two options, `-r` and `-f`:

```
pax -rf /dev/rfloppy0
```

In the example, the `-f` option (pronounced ''minus f'') is entered without its own hyphen, even though when mentioned in running text it appears with a leading hyphen.

### Styling of cross-references to manual pages

The manual pages are organized primarily in terms of sections, and secondarily in terms of books for different audiences. The standard A/UX cross-reference notation leaves out the book title, but refers to the section designation:

*item*(*section*)

where *item* is the name of the command, subroutine, or other A/UX provision, and *section* is the section where the manual page resides.

For example,

`cat`(1)

refers to the command `cat`, which is described in Section 1, which is in *A/UX Command Reference*.

As a guide to the location of sections, you can refer to the general table of contents of each of the primary reference manuals, or to ''Current Organization of Sections into Books'' later in this preface. (The binder spines are also labeled with the section numbers, and occasionally section subdivisions, that are in each binder.)

Note also that there are a number of subcategory designations that can follow the digit reference in (1), (2), (3), (4), and (5), such as (1N). Detailed explanations of these subcategory designations are provided later in this preface.

## Previous organization of sections into books

You may be curious about the logic behind the numbering of sections. The derivation of this numbering is much clearer when you realize that originally there was only one reference manual, the *UNIX User Manual*. In fact the manual pages were once considered the primary UNIX documentation, and the other books were originally considered supplements.

In the early days, all the manual pages easily fit into one book, in sections numbered 1 through 8. Section 8 originally contained the manual pages that are now located in Section 1M.

With the expansion of the original sections as UNIX grew, it became necessary to split the original book into several books, and this was done according to the audience they served. However, the original section numbering was preserved after the split because by then each number had come to have a particular meaning to UNIX users.

Because the original section numbers were preserved and then sections were recollated in accordance with the audience they served, the resulting books do not, for the most part, contain sequentially numbered sections.

The next section explains in detail how the sections are currently mapped into books.

There was another factor that led to the need to preserve the original section numbers. Some routines, system calls, and commands have the same names. To allow you to distinguish one from another, the section number is often included along with the name. While new section numbers could have helped distinguish these entities, the old numbers were much more familiar to UNIX users.

Besides distinguishing amongst identically named A/UX provisions, the section number helps identify each manual page as one that describes a command, a system call, a library routine, and so forth. Regular UNIX users sooner or later memorize what category is identified by each section number. After doing so, you can deduce how the sections must be split up into books—since each book serves a particular audience and each section category also goes along with a particular audience, the match-ups become fairly easy for you to make. The memorization part of this task is more or less considered an initiation rite for those who wish to learn to use UNIX effectively.

Until the 3.0 release of A/UX, the organization of sections into books was static. With the 3.0 release however, Section 7 has been moved out of *A/UX System Administrator's Reference* and into *A/UX Programmer's Reference*. This means that command provisions are now the exclusive focus of both *A/UX Command Reference* and *A/UX System Administrator's Reference*.

## Current organization of sections into books

All manual pages are grouped by section. The sections are grouped by general function and are numbered according to standard conventions as follows:

1 User Commands

1M System Maintenance Commands

2 System Calls

3 Subroutines

4 File Formats

5 Miscellaneous Facilities

6 Games

7 Drivers and Interfaces for Devices

8 A/UX Startup Shell Commands

Each group or section of manual pages is located in one of the reference books. Each reference book may comprise more than one binder. This section explains where these sections are currently located with respect to the three primary reference books. It also describes any subcategories that may be present in a given section.

*A/UX Command Reference* contains Sections 1 and 6.

- Section 1—User Commands
  This section describes commands that require no special access privileges. The commands in Section 1 may also belong to a special category, such as networking commands. Where applicable, these categories are indicated by a letter designation that follows the section number. For example, the "N" in `ypcat`(1N) indicates that this manual page describes a networking command. Here is an explanation of each subcategory:

  1C Communications commands, such as `cu` and `tip`.

  1G Graphics commands, such as `graph` and `tplot`.

  1N Networking commands, such as those that help support various networking subsystems, including the Network File System (NFS), Remote Process Control (RPC) subsystem, and Internet subsystem.

- Section 6—Games
  This section contains all of the games provided with A/UX, such as `cribbage` and `worms`.

*A/UX Programmer's Reference* contains Sections 2 through 5 and Section 7.

- Section 2—System Calls
  This section describes the services provided by the A/UX system kernel, including the C language interface. It includes two special categories. Where applicable, these categories are indicated by the letter designation that follows the section number. For example, the "N" in `connect`(2N) indicates that this manual page describes a networking command. Here is an explanation of each subcategory:

  2N Networking system calls

  2P POSIX system calls

- Section 3—Subroutines
  This section describes the available subroutines. The binary versions of these subroutines are in the system libraries in the `/lib` and `/usr/lib` directories. The section includes seven special categories. Where applicable, these categories are indicated by the letter designation that follows the section number. For example, the "N" in `mount`(3N) indicates that this manual page describes a networking command. Here is an explanation of each subcategory:

  3C C and assembly-language library routines

  3F Fortran library routines

  3M Mathematical library routines

  3N Networking routines

  2P POSIX routines

  3S Standard I/O library routines

  3X Miscellaneous routines

- Section 4—File Formats
  This section describes the structure of some files, but does not include files that are used by only one command (such as the assembler's intermediate files). The C language `struct` declarations corresponding to these formats are in the `/usr/include` and `/usr/include/sys` directories. There is one special category in this section, indicated by the letter designation "N" following the section number:

4N Networking formats

- Section 5—Miscellaneous Facilities
  This section describes various character sets, macro packages, and other miscellaneous facilities. There are two special categories in this section. Where applicable, these categories are indicated by the letter designation that follows the section number. For example, the "P" in `tcp`(1P) indicates a protocol. Here is an explanation of each subcategory:

  5F Protocol families

  5P Protocol descriptions

- Section 7—Drivers and Interfaces for Devices
  This section describes the drivers and interfaces through which devices are normally accessed. Access to one or more disk devices is fairly transparent when you are working with them in terms of files. When you want to use A/UX commands to communicate with devices more directly, at a level beyond the moderation of file systems, device files serve your needs. Such a level of communication permits you to request more explicit operating modes that may be supported by a disk (such as accessing disk partition maps), or that may be supported by other types of devices, such as tape drives and modems. For example, you can access a tape device in automatic-rewind mode as described in `tc`(7).

*A/UX System Administrator's Reference* contains Sections 1M and 8.

- Section 1M—System Maintenance Commands
  This section describes system maintenance programs such as `fsck` and `mkfs`.

- Section 8—A/UX Startup Shell Commands
  This section describes the commands that are available from within the A/UX Startup shell. This section includes detailed descriptions of the commands that contribute to the boot process and those that help with the maintenance of inactive file systems.

## For more information

To find out where you need to go for more information about how to use A/UX, see *Road Map to A/UX*. This guide contains descriptions of each A/UX guide and ordering information for all the guides in the A/UX documentation suite.

# Table of Contents

## Section 1: User Commands (G-P)

**NAME**

get — gets a version of an SCCS file

**SYNOPSIS**

get [-a*seq-no*] [-b] [-c*cutoff*] [-e] [-g] [-i*list*] [-k] [-l[*p*]] [-m] [-n] [-p] [-r*SID*] [-s] [-t] [-w*string*] [-x*list*] *file...*

**ARGUMENTS**

-a*seq-no*

The delta sequence number of the SCCS file delta (version) to be retrieved (see sccsfile(4)). This keyletter is used by the comb(1) command; it is not a generally useful keyletter, and users should not use it. If both the -r and -a options are specified, the -a options is used. Care should be taken when using the -a option in conjunction with the -e option, as the SID of the delta to be created may not be what one expects. The -r option can be used with the -a and -e options to control the naming of the SID of the delta to be created.

-b Indicates that the new delta should have an SID in a new branch as shown in Table 1, when used with the -e option. This option is ignored if the b is not present in the file (see admin(1)) or if the retrieved delta is not a leaf delta. (A leaf delta is one that has no successors on the SCCS file tree.)

*Note:* A branch delta may always be created from a nonleaf delta.

-c*cutoff*

Specifies the *cutoff* date-time, in the form: *YY*[MM[*DD*[*HH*[MM[*SS*]]]]]. No changes (deltas) to the SCCS file which were created after the specified *cutoff* date-time are included in the generated ASCII text file. Units omitted from the date-time default to their maximum possible values; that is, -c7502 is equivalent to -c750228235959. Any number of non-numeric characters may separate the various 2-digit pieces of the *cutoff* date-time. This feature allows one to specify a *cutoff* date in the form: -c77/2/2 9:22:25. Note that this implies that one may use the %E% and %U% identification keywords (see later) for a nested get within, for example, the input to a send(2N) command:

```
~!get "-c%E%  %U%" s.file
```

-e Indicates that the get is for the purpose of editing or making a change (delta) to the SCCS file via a subsequent use of delta(1). When this option is used in a get command for a particular version (SID) of the SCCS file, it prevents a further get from editing on the same SID until delta is executed or the j (joint edit) flag is set in the SCCS

file (see admin(1)). Concurrent use of get -e for different SIDs is always allowed.

If the *g-file* generated by get with an -e option is accidentally ruined in the process of editing it, it may be regenerated by re-executing the get command with the -k option in place of the -e option.

SCCS file protection specified via the ceiling, floor, and authorized user list stored in the SCCS file (see admin(1)) are enforced when the -e keyletter is used.

*file* Specifies the file to be processed.

-g Suppresses the actual retrieval of text from the SCCS file. It is primarily used to generate an *l-file*, or to verify the existence of a particular SID.

-i*list*
Specifies a *list* of deltas to be included (forced to be applied) in the creation of the generated file. The *list* has the following syntax:

*<list>* : : = *<range>* | *<list>* , *<range>*
*<range>* : : = *SID* | *SID-SID*

SID, the SCCS Identification of a delta, may be in any form shown in the ''SID Specified'' column; partial SIDs are interpreted as shown in the ''SID Retrieved'' column of Table 1.

-k Suppresses the replacement of identification keywords (described below) in the retrieved text by their value. The -k option is implied by the -e option.

-l[*p*]
Causes a delta summary to be written into an *l-file*. If -lp is used, then an *l-file* is not created; the delta summary is written on the standard output instead. See NOTES for the format of the *l-file*.

-m Causes each text line retrieved from the SCCS file to be preceded by the SID of the delta that inserted the text line in the SCCS file. The format is: SID, followed by a horizontal tab, followed by the text line.

-n Causes each generated text line to be preceded with the %M% identification keyword value (described later) The format is: %M% value, followed by a horizontal tab, followed by the text line. When both the -m and -n options are used, the format is: %M% value, followed by a horizontal tab, followed by the -m option generated format.

-p Causes the text retrieved from the SCCS file to be written on the standard output. No *g-file* is created. All output which normally goes to the standard output goes to file descriptor 2 instead, unless the -s

option is used, in which case it disappears.

-r*SID*
Specifies the SCCS identification string (SID) of the version (delta) of an SCCS file to be retrieved. The table that follows these descriptions shows, for the most useful cases, what version of an SCCS file is retrieved (as well as the SID of the version to be eventually created by delta(1) if the -e keyletter is also used) as a function of the SID specified.

-s Suppresses all output normally written on the standard output. However, fatal error messages (which always go to file descriptor 2) remain unaffected.

-t Accesses the most recently created (top) delta in a given release (for example, -r1), or release and level (for example, -r1.2).

-w*string*
Substitutes *string* for all occurrences of %W% when running get on the file.

-x*list*
Specifies a *list* of deltas to be excluded (forced not to be applied) in the creation of the generated file. See the -i option for the *list* format.

**DESCRIPTION**

get generates an ASCII text file from each named SCCS file according to the specifications given by keyletter arguments that begin with -. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, get behaves as though each file in the directory is specified as a named file, except that non-SCCS files (last component of the pathname does not begin with s.) and unreadable files are silently ignored. If a name of - is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The generated text is normally written into a file called the *g-file*, the name of which is derived from the SCCS filename simply by removing the leading s. (see also NOTES, later in this section).

For each file processed, get responds (on the standard output) with the SID being accessed and with the number of lines retrieved from the SCCS file.

If the -e options is used, the SID of the delta to be made appears after the SID accessed and before the number of lines generated. If there is more than one named file or if a directory or standard input is named, each

filename is printed (preceded by a newline) before it is processed. If the `-i` option is used included deltas are listed following the notation `Included`; if the `-x` option is used, excluded deltas are listed following the notation `Excluded`.

Determination of SCCS Identification String

| SID* Specified | -b Keyletter Used† | Other Conditions | SID Retrieved | SID of Delta to be Created |
|---|---|---|---|---|
| none‡ | no | R defaults to mR | mR.mL | mR.(mL+1) |
| none‡ | yes | R defaults to mR | mR.mL | mR.mL.(mB+1).1 |
| R | no | R > mR | mR.mL | R.1*** |
| R | no | R = mR | mR.mL | mR.(mL+1) |
| R | yes | R > mR | mR.mL | mR.mL.(mB+1).1 |
| R | yes | R = mR | mR.mL | mR.mL.(mB+1).1 |
| R | - | R < mR and R does *not* exist | hR.mL** | hR.mL.(mB+1).1 |
| R | - | Trunk succ.# in release > R and R exists | R.mL | R.mL.(mB+1).1 |
| R.L | no | No trunk succ. | R.L | R.(L+1) |
| R.L | yes | No trunk succ. | R.L | R.L.(mB+1).1 |
| R.L | - | Trunk succ. in release ≥ R | R.L | R.L.(mB+1).1 |
| R.L.B | no | No branch succ. | R.L.B.mS | R.L.B.(mS+1) |
| R.L.B | yes | No branch succ. | R.L.B.mS | R.L.(mB+1).1 |
| R.L.B.S | no | No branch succ. | R.L.B.S | R.L.B.(S+1) |
| R.L.B.S | yes | No branch succ. | R.L.B.S | R.L.(mB+1).1 |
| R.L.B.S | - | Branch succ. | R.L.B.S | R.L.(mB+1).1 |

* R, L, B, and S are the *release*, *level*, *branch*, and *sequence* components of the SID , respectively; 'm' means *maximum*. Thus, for example, `R.mL` means the maximum level number within release **R**; "R.L.(mB+1).1" means the first sequence number on the *new* branch (i.e., maximum branch number plus one) of level L within release R. Note that if the SID specified is of the form R.L, R.L.B, or R.L.B.S, each of the specified components *must* exist.

** "hR" is the highest *existing* release that is lower than the specified, *nonexistent*, release R.

*** This is used to force creation of the *first* delta in a *new* release.

\# Successor.

† The `-b` option is effective only if the `b` flag (see `admin`(1)) is present in the file. An entry of - means "irrelevant."

‡ This case applies if the d (default SID) flag is *not* present in the file. If the d flag *is* present in the file, then the SID obtained from the d flag is interpreted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

**Identification keywords**

Identifying information is inserted into the text retrieved from the SCCS file by replacing *identification keywords* with their value wherever they occur. The following keywords may be used in the text stored in an SCCS file:

*Keyword Value*

%M%
: Module name: either the value of the m flag in the file (see admin(1)), or if absent, the name of the SCCS file with the leading s. removed.

%I%
: SCCS identification (SID) (%R%.%L%.%B%.%S%) of the retrieved text.

%R%
: Release.

%L%
: Level.

%B%
: Branch.

%S%
: Sequence.

%D%
: Current date (*YY/MM/DD*).

%H%
: Current date (*MM/DD/YY*).

%T%
: Current time (*HH:MM:SS*).

%E%
: Date newest applied delta was created (*YY/MM/DD*).

%G%
: Date newest applied delta was created (*MM/DD/YY*).

%U%
: Time newest applied delta was created (*HH:MM:SS*).

`%Y%`
Module type: value of the `t` flag in the SCCS file (see `admin`(1)).

`%F%`
SCCS filename.

`%P%`
Fully qualified SCCS filename.

`%Q%`
The value of the `q` flag in the file (see `admin`(1)).

`%C%`
Current line number. This keyword is intended for identifying messages output by the program such as `this should not have happened` type errors. It is not intended to be used on every line to provide sequence numbers.

`%Z%`
The 4-character string `@(#)` recognizable by `what`(1).

`%W%`
A shorthand notation for constructing `what`(1) strings for A/UX system program files.
`%W% = %Z%%M%<horizontal-tab>%I%`

`%A%`
Another shorthand notation for constructing `what`(1) strings for non-A/UX system program files.
`%A% = %Z%%Y% %M% %I%%Z%`

**EXAMPLES**

The command:

```
get -e s.file1
```

generates from the SCCS format file, `s.file1`, the text file, `file1`, for editing.

**NOTES**

Several auxiliary files may be created by `get`. These files are known generically as the *g-file*, *l-file*, *p-file*, and *z-file*. The letter before the hyphen is called the *tag*. An auxiliary filename is formed from the SCCS filename; the last component of all SCCS filenames must be of the form `s.`*module-name*, and the auxiliary files are named by replacing the leading `s` with the tag. The *g-file* is an exception to this scheme: the *g-file* is named by removing the `s.` prefix. For example, `s.xyz.c`, the auxiliary filenames would be `xyz.c`, `l.xyz.c`, `p.xyz.c`, and `z.xyz.c`, respectively.

The *g-file*, which contains the generated text, is created in the current directory (unless the `-p` option is used). A *g-file* is created in all cases, whether or not any lines of text were generated by the `get`. It is owned by the real user. If the `-k` option is used or implied its mode is 644; otherwise, its mode is 444. Only the real user need have write permission in the current directory.

The *l-file* contains a table showing which deltas were applied in generating the retrieved text. The *l-file* is created in the current directory if the `-l` keyletter is used; its mode is 444 and it is owned by the real user. Only the real user need have write permission in the current directory.

Lines in the *l-file* have the following format:

a. A blank character if the delta was applied;
   * otherwise.

b. A blank character if the delta was applied or was not applied and ignored;
   * if the delta was not applied and was not ignored.

c. A code indicating a ''special reason'' why the delta was or was not applied:

   `I`: Included.

   `X`: Excluded.

   `C`: Cut off (by a `-c` option).

d. Blank.

e. SCCS identification (SID).

f. TAB character.

g. Date and time (in the form *YY/MM/DD* HH:MM:SS*)* of creation.

h. Blank.

i. Login name of person who created `delta`.

   The comments and MR data follow on subsequent lines, indented one horizontal tab character. A blank line terminates each entry.

The *p-file* is used to pass information resulting from a `get` with an `-e` option along to `delta`. Its contents are also used to prevent a subsequent execution of `get` with an `-e` option for the same SID until `delta` is executed or the joint edit flag, `j`, (see `admin`(1)) is set in the SCCS file. The *p-file* is created in the directory containing the SCCS file and the effective user must have write permission in that directory. Its mode is 644 and it is owned by the effective user. The format of the *p-file* is: the acquired SID, followed by a blank, followed by the SID that the new delta will have when it is made, followed by a blank, followed by the login name

of the real user, followed by a blank, followed by the date-time the `get` was executed, followed by a blank and the `-i` option argument if it was present, followed by a blank and the `-x` option argument if it was present, followed by a newline. There can be an arbitrary number of lines in the *p-file* at any time; no two lines can have the same new delta SID.

The *z-file* serves as a *lock-out* mechanism against simultaneous updates. Its contents are the binary (2 bytes) process ID of the command (that is, `get`) that created it. The *z-file* is created in the directory containing the SCCS file for the duration of `get`. The same protection restrictions as those for the *p-file* apply for the *z-file*. The *z-file* is created mode 444.

**LIMITATIONS**

If the effective user has write permission (either explicitly or implicitly) in the directory containing the SCCS files, but the real user does not, then only one file may be named when the `-e` option is used.

**DIAGNOSTICS**

Use `help` for explanations.

**FILES**

`/usr/bin/get`
Executable file

**SEE ALSO**

`admin`(1), `cdc`(1), `comb`(1), `delta`(1), `help`(1), `prs`(1), `rmdel`(1), `sact`(1), `sccs`(1), `sccsdiff`(1), `unget`(1), `val`(1), `what`(1)

`sccsfile`(4) in *A/UX Programmer's Reference*

"SCCS Reference" in *A/UX Programming Languages and Tools, Volume 2*

**NAME**

getopt — parses command options

**SYNOPSIS**

getopt [*flag-letter*[:]]... [*input-string*]

**ARGUMENTS**

*flag-letter*[:]

Helps control how *input-string* is manipulated to detect flags and flag arguments. If a *flag-letter* is followed by a : (colon), getopt expects to find a flag-specific argument following that flag in the *input-string*. For example,

```
getopt a: $*
```

requires that -a always be followed by its own argument (either with or without a space separator), as in the following:

```
yourcommand -a param ...
yourcommand -aparam ...
```

*input-string*

Specifies the input string to be parsed. The special option -- can be used within *input-string* to request that only a portion of *input-string* actually be processed for the presence of flags. Any text following -- is not processed. If it is not supplied explicitly, getopt still generates the symbol in its output to help separate any options and arguments found from any nonflag arguments that might remain in *input-string*. For example,

```
getopt abo: $*
```

returns

```
-a -o param -- xxxx yyyy zzzz
```

when you place the getopt command line (shown above) in a command script invoked with

```
yourcommand -aoparam xxxx yyyy zzzz
```

Even though a hyphen was not specified in front of each option in this example, the output of getopt includes hyphens in front of both a and o.

**DESCRIPTION**

getopt returns *input-string* with additional separators to help distinguish any options, any arguments associated with the options, and any arguments not associated with the options. By replacing *input-string* with the command arguments $* for a script, getopt helps shell scripts to parse their command-line arguments by making a regularized copy of them as well as checking them for legal options. The regularization that getopt

can perform for each option is twofold or threefold:

1. Each option on the command line is returned separated with white space.
2. Each option on the command line is returned with a leading hyphen.
3. Optionally, the argument associated with a given is returned with white space.

To reset the shell's positional parameters (`$1 $2...`) so that they are regularized by `getopt` and so that each discrete flag and flag argument is stored as a unique positional parameter, specify the output of `getopt` as the argument for `set` by using command substitution:

```
set -- 'getopt abo: $*'
```

### Quoted Arguments

`getopt` correctly parses quoted arguments within *input-string*. However, if the input string you wish to parse with `getopt` is specified as `$*` in order to request the parsing of command-line arguments, any quotes that may be present in the command line are automatically stripped by the shell. In such cases you need to use a reference to the unstripped version of the command-line arguments, `$@`, which is available in the `sh` and `ksh` shells. For example

```
getopt a:b: "$@"
```

correctly returns

```
-a 'hello world' -b oneword --
```

when the `getopt` command line (shown above) is in a script invoked with

```
yourcommand -a'hello world' -b oneword
```

The challenge then becomes resetting the shell's positional parameters so that `'hello world'` is interpreted as one positional argument rather than two positional arguments (`'hello` as one argument and `world'` as another). To do so, use `eval` to invoke the `set` function, as in the following:

```
eval set -- 'getopt abo: "$@"'
```

To preserve the opportunity to process the exit status of `getopt`, the `eval` command line cannot be used as shown preceding. (The exit status from `getopt` is lost when `eval` is used to evaluate a command string.)

The only recourse is to defer the resetting of positional arguments until after the exit status stored in the `$?` variable can be tested:

```
x='getopt abo: "$@"'
if [ $? != 0 ]
```

```
then
   echo $USAGE
   exit 2
fi
eval set -- $x
```

A nonzero exit value conventionally indicates that processing was terminated abnormally. So in the example preceding, the value of the exit status variable is used to detect whether or not the string processing performed by `getopt` succeeded: which in turn depends on whether or not `getopt` recognized and regularized the input string in terms of the control arguments supplied.

**EXAMPLES**

The following code fragment shows how one might process the arguments for a command that can take the options `a` or `b`, as well as the option `o`, which requires an argument:

```
x='getopt abo: "$@"'
if [ $? != 0 ]
then
        echo $USAGE
        exit 2
if
eval set -- $x
for i in "$@"
do
        case $i in
        -a | -b)    FLAG=$i; shift;;
        -o)         OARG=$2; shift 2;;
        --)         shift;   break;;
        esac
done
```

If this code is placed in a script called `cmd`, then any of the following invocations are accepted as equivalent:

```
cmd -aoarg file1 file2
cmd -a -oarg file1 file2
cmd -o arg -a file1 file2
cmd -a -oarg -- file1 file2
```

The script also interprets any imbedded blanks in arguments correctly, as long as the arguments are quoted as in the following:

```
cmd -aoarg "file one" "file two"
cmd -a -o "an arg" "file two" "file two"
cmd -o "an arg" -a "file one" "file two"
```

```
cmd -a -o"an arg" -- "file two" "file two"
```

**DIAGNOSTICS**

The `getopt` command prints an error message on the standard error when it encounters an option letter not included as a *flag-letter*.

**FILES**

`/bin/getopt`
Executable file

**SEE ALSO**

`csh`(1), `ksh`(1), `sh`(1)

`getopt`(3C) in *A/UX Programmer's Reference*

## NAME

grap — invokes a pic preprocessor for drawing graphs

## SYNOPSIS

grap [-T*tty-type*] [-l] [-] [*file*]...

## ARGUMENTS

- Specifies the standard input.

*file* Specifies the file to be preprocessed by the grap command.

-l Stops grap from looking for a library file of macro defines, /usr/lib/dwb/grap.defines.

-T*tty-type*
Specifies *tty-type* as grap's output device. Currently supported devices are psc (POSTSCRIPT device such as the Apple LaserWriter) and aps (Autologic APS-5). The default is -Tpsc.

## DESCRIPTION

grap is a language for typesetting graphs. It is also the name of a preprocessor that feeds input to pic. Thus, a typical command line would appear as follows:

grap *files* | pic | troff | *output-device*

Graphs are surrounded by the troff commands .G1 and .G2. Data that is enclosed is scaled and plotted, with tick marks supplied automatically. Commands exist to modify the frame, add labels, override the default ticks, change the plotting style, define coordinate ranges and transformations, and include data from files. In addition, grap provides the same loops, conditionals, and macro processing that pic does.

## FILES

/usr/bin/grap
Executable file

/usr/lib/dwb/grap.defines
File containing definitions of standard plotting characters

## SEE ALSO

pic(1)

"grap Reference," in *A/UX Text Processing Tools*

**NAME**

graph — draws a graph

**SYNOPSIS**

graph [-a [*sp*] [*st*]] [-b] [-c*label*] [-g [*style*]] [-h *hspace*] [-l *title*] [-m[*mode*]] [-r *rspace*] [-s] [-t] [-u *uspace*] [-w *wspace*] [-x [l] [a] [b] [c]] [-y [l] [a] [b] [c]]

**ARGUMENTS**

-a [*sp*] [*st*]
Supplies abscissas automatically (they are missing from the input); spacing is given by *sp* (default 1). The option, *st*, is the starting point for automatic abscissas (default 0 or the lower limit given by the -x option.

-b Breaks (disconnects) the graph after each label in the input.

-c*label*
Specifies a character string given by *label* which is the default label for each point.

-g [*style*]
Specifies a grid style. where Replace *style* with one of the following: 0=no grid, 1=frame with ticks, and 2=full grid (default).

-h *hspace*
Specifies the fraction of the space for height.

-l *title*
Specifies the label for the graph.

-m[*mode*]
Specifies the mode (style) of connecting lines: 0=disconnected, 1=connected (default). Some devices give distinguishable line styles for other small integers (e.g., the Tektronix 4014: 2=dotted, 3=dash-dot, 4=short-dash, 5=long-dash).

-r *rspace*
Specifies the fraction of the space to move right before plotting.

-s Saves the screen, don't erase before plotting.

-t Transposes horizontal and vertical axes.

-u *uspace*
Specifies the fraction of the space to move up before plotting.

-w *wspace*
Specifies the fraction of the space for width. (-x now applies to the vertical axis.)

`-x` [*l*] [*a*] [*b*] [*c*]
Specifies certain quantities for the *x* axis. If `l` is present, *x* axis is logarithmic. *a* (and *b*) are lower (and upper) *x* limits. *c*, if present, is the grid spacing on the *x* axis. Normally, these quantities are determined automatically.

`-y` [*l*] [*a*] [*b*] [*c*]
Specifies certain quantities for the *x* axis. If `l` is present, *x* axis is logarithmic. *a* (and *b*) are lower (and upper) *x* limits. *c*, if present, is the grid spacing on the *x* axis. Normally these quantities are determined automatically.

**DESCRIPTION**

`graph` with no options takes pairs of numbers from the standard input as abscissas and ordinates of a graph. Successive points are connected by straight lines. The graph is encoded on the standard output for display by the `tplot` filters.

If the coordinates of a point are followed by a non-numeric string, that string is printed as a label beginning on the point. Labels may be surrounded with quotes (`"`), in which case they may be empty or contain blanks and numbers; labels never contain newlines.

A legend indicating grid range is produced with a grid unless the `-s` option is present. If a specified lower limit exceeds the upper limit, the axis is reversed.

**LIMITATIONS**

The `graph` command stores all points internally and drops those for which there isn't room.

Segments that run out of bounds are dropped, not windowed.

Logarithmic axes may not be reversed.

Options and their arguments must be delimited by at least one space.

**FILES**

`/usr/bin/graph`
Executable file

**SEE ALSO**

`spline`(1G), `tplot`(1G)

**NAME**

greek — filters text for vintage display devices

**SYNOPSIS**

greek [-T*terminal*]

**ARGUMENTS**

-T*terminal*

Specifies an alternate terminal type to be used with the greek command. The following *terminal*s are currently recognized:

300
DASI 300

300-12
DASI 300 in 12-pitch

300s
DASI 300s

300s-12
DASI 300s in 12-pitch

450
DASI 450

450-12
DASI 450 in 12-pitch

1620
Diablo 1620 (alias DASI 450)

1620-12
Diablo 1620 (alias DASI 450) in 12-pitch

4014
Tektronix 4014

tek
Tektronix 4014

**DESCRIPTION**

greek is a filter that reinterprets the extended character set, as well as the reverse and half-line motions, of a 128-character Teletype Model 37 terminal and certain other terminals. Special characters are simulated by overstriking, if necessary and possible. If the argument is omitted, greek attempts to use the environment variable $TERM (see environ(5)).

**EXAMPLES**

The command:

```
nroff file | greek -T4014
```

reinterprets the extended character set on a Tektronix 4014 terminal.

**FILES**

`/usr/bin/greek`
Executable file

`/usr/bin/300`
File containing terminal information

`/usr/bin/300s`
File containing terminal information

`/usr/bin/4014`
File containing terminal information

`/usr/bin/450`
File containing terminal information

**SEE ALSO**

`300`(1), `4014`(1), `450`(1), `eqn`(1), `mm`(1), `nroff`(1), `tplot`(1G)
`term`(4), `environ`(5), `greek`(5) in *A/UX Programmer's Reference*

**NAME**

grep, egrep, fgrep — search a file for a specific pattern

**SYNOPSIS**

grep [-b] [-c] [-i] [-l] [-n] [-s] [-v] *expression* [*file*]...

egrep [-b] [-c] [-e *expression*] [-f *file*] [-i] [-l] [-n] [-v] [*expression*] [*file*]...

fgrep [-b] [-c] [-e *expression*] [-f *file*] [-i] [-l] [-n] [-v] [-x] [*strings*] [*file*]...

**ARGUMENTS**

-b Precedes each line by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.

-c Prints only a count of matching lines.

-e *expression*
Acts the same as a simple *expression* argument, but useful when the *expression* begins with a -. This option does not work with the grep command.

*expression*
Specifies the regular expression that is used in the egrep command.

-f *file*
Takes the regular *expression* (egrep) or *strings* list (fgrep) from the *file*.

*file* Specifies the file that will be searched.

-i Ignores upper/lowercase distinction during comparisons.

-l Lists (once) only the names of files with matching lines, separated by newlines.

-n Precedes each line by its relative line number in the file.

-s Suppresses the error messages produced for nonexistent or unreadable files. This option is used for grep only.

*string*
Specifies the string of character to look for in the specified file.

-v Prints all lines but those matching.

-x Means exact. Only lines matched in their entirety are printed. This option is only used for fgrep.

**DESCRIPTION**

grep searches the input *files* (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output. The grep command patterns are limited regular expressions in the style of ed;

they use a compact nondeterministic algorithm.

`egrep` patterns are full regular expressions; they use a fast deterministic algorithm that sometimes needs exponential space.

`fgrep` patterns are fixed *strings*; it is fast and compact.

In all cases, the filename is output if there is more than one input file. Care should be taken when using the characters `$`, `*`, `[`, `^`, `|`, `(`, `)`, and `\` in *expression*, because they are also meaningful to the shell. It is safest to enclose the entire *expression* argument in single quotation marks (`'...'`).

The `fgrep` command searches for lines that contain one of the *strings* separated by newlines.

The `egrep` command accepts regular expressions as in `ed`(1), except for `\(` and `\)`, with the addition of:

1. A regular expression followed by `+` matches one or more occurrences of the regular expression.
2. A regular expression followed by `?` matches 0 or 1 occurrences of the regular expression.
3. Two regular expressions separated by `|` or by a newline match strings that are matched by either.
4. A regular expression may be enclosed in parentheses `( )` for grouping.

The order of precedence of operators is `[ ]`, then `* ? +`, then concatenation, then `|` and newline.

**EXAMPLES**

The command:

```
grep -v -c 'regular' grep.1
```

reports a count of the number of lines that do not contain the word *regular* in the file `grep.1`.

**LIMITATIONS**

Ideally there should be only one `grep`, but we do not know a single algorithm that spans a wide enough range of space-time tradeoffs.

Lines are limited to `BUFSIZ` characters; longer lines are truncated. (`BUFSIZ` is defined in `/usr/include/stdio.h`.)

The `egrep` command does not recognize ranges, such as `[a-z]`, in character classes.

If there is a line with embedded nulls, `grep` will only match up to the first null; if it matches, it will print the entire line.

**STATUS MESSAGES AND VALUES**

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files (even if matches were found).

**FILES**

`/bin/grep`
Executable file

`/bin/egrep`
Executable file

`/bin/fgrep`
Executable file

**SEE ALSO**

awk(1), csh(1), ed(1), ex(1), ksh(1), lex(1), sed(1), sh(1), vi(1)

**NAME**

groups — displays group memberships

**SYNOPSIS**

groups [*user*]

**ARGUMENTS**

*user*

Specifies the name of the user whose groups you want displayed.

**DESCRIPTION**

groups shows the groups to which you or the optionally-specified *user* belong. Each user belongs to a group specified in the password file /etc/passwd and possibly to other groups as specified in the file /etc/group. If you do not own a file, but belong to the group which owns it, you are granted group access to the file.

When a new file is created, it is given the group of the containing directory.

**LIMITATIONS**

More groups should be allowed. Eight groups is currently the limit.

**FILES**

/usr/bin/groups

Executable file

/etc/passwd

File containing user passwords.

/etc/group

File containing group passwords.

**SEE ALSO**

setgroups(2) in *A/UX Programmer's Reference*

*See* spell(1)

*See* spell(1)

**NAME**

head — displays the first few lines of a file

**SYNOPSIS**

head [-*count*] [*file*]...

**ARGUMENTS**

*count*
Specifies the number of lines to be displayed. If this option is not given, the default is 10.

*file* Specifies the file to be displayed. If you specify *file* as a dash, (-), the standard input is read.

**DESCRIPTION**

This filter displays the first lines (specified by *count*), for each of the specified files or for the standard input.

**EXAMPLES**

The command:

```
head -6 filea fileb filec
```

will print out the first six lines of the three specified files. The filename will appear before each new set of headlines listed, if more than one file has been specified.

**FILES**

/bin/head
Executable file

**SEE ALSO**

awk(1), cat(1), more(1), pg(1), tail(1)

## NAME

`help` — provides help information about SCCS commands and messages

## SYNOPSIS

`help` [*args*]...

## ARGUMENTS

*args*

Specifies the message number (which normally appear in parentheses following the message) or command name for which you want information. If this option is not specified, `help` will prompt you for one.

## DESCRIPTION

`help` finds information to explain a message from an SCCS command or explain the use of an SCCS command. Zero or more arguments may be supplied.

The arguments must be of one of the following types:

type 1
: Begins with non-numerics, ends in numerics. The non-numeric prefix is usually an abbreviation for the program or set of routines which produced the message (for example, `ge4`, for message 6 from the `get` command).

type 2
: Does not contain numerics as a command (such as `get`)

type 3
: Is all numeric (for example, `26`)

The response of the program will be the explanatory information related to the argument, if there is any.

When all else fails, enter:

```
help stuck
```

## EXAMPLES

The command:

```
help he2
```

prints the message for error number `he2`.

## STATUS MESSAGES AND VALUES

Use `help` for explanations.

**FILES**

`/usr/bin/help`
Executable file

`/usr/lib/help/*`
Message files

`/usr/lib/help/helploc`
File containing pathnames leading to custom message files

`/usr/lib/help/lib/help2`
Executable file called by `help`

**SEE ALSO**

`admin`(1), `cdc`(1), `comb`(1), `delta`(1), `get`(1), `unget`(1), `help`(1), `prs`(1), `rmdel`(1), `sact`(1), `sccsdiff`(1), `val`(1), `what`(1)

**NAME**

hex — converts an object file to Motorola S-record format

**SYNOPSIS**

hex [-f] [-l] [-n#] [-ns8] [-r] [-s0] [-s2] [+*saddr*] *ifile*

**ARGUMENTS**

+*saddr*
Specifies the starting load address (in hex).

-f Causes the file specified to be shipped as is without treating it as an object file.

*ifile* Specifies the file to be downloaded. The file's starting address is used if *saddr* is not present.

-l Outputs the "Loading at" message.

-n#
Specifies the number of characters to output per record. Replace # with a decimal number.

-ns8
Does not output a trailing s8 (s9) record.

-r Outputs a carriage return at the end of each S-record (instead of a newline).

-s0
Outputs a leading s0 record.

-s2
Records only (no s1 records are produced).

**DESCRIPTION**

hex translates executable object files into ASCII formats suitable for Motorola S-record downloading.

**EXAMPLES**

In the command:

```
hex objfile
```

objfile is the object file to be downloaded.

**FILES**

/usr/bin/hex
Executable file

**SEE ALSO**

as(1), od(1), rcvhex(1)

a.out(4) in *A/UX Programmer's Reference*

## NAME

hostid — sets or displays the identifier of the current host system

## SYNOPSIS

hostid [*identifier*]

## ARGUMENTS

*identifier*

Specifies the identifier to be displayed.

## DESCRIPTION

hostid displays the identifier of the current host in hexadecimal. This numeric value is expected to be unique across all hosts and is normally set to the host's Internet address (for Ethernet or TCP/IP). The superuser may set the hostid by giving a hexadecimal argument; this is usually done in the startup script /etc/sysinitrc.

## FILES

/bin/hostid

Executable file

## SEE ALSO

gethostid(2N)

**NAME**

hostname — sets or displays the name of the current host system

**SYNOPSIS**

hostname [*nameofhost*]

**ARGUMENTS**

*nameofhost*

Specifies the name of the host system you wish to be displayed or set.

**DESCRIPTION**

hostname displays the name of the current host. The superuser can set the hostname by giving an argument; this is usually done in the startup script /etc/sysinitrc.

**FILES**

/bin/hostname

Executable file

**SEE ALSO**

gethostname(2N)

**NAME**

hyphen — finds hyphenated words

**SYNOPSIS**

hyphen [*file*]...

**ARGUMENTS**

*file* Specifies the file that is searched for hyphenated words. If this option is not given, hyphen uses the standard input.

**DESCRIPTION**

hyphen finds all the hyphenated words ending lines in *file*s and prints them on the standard output. If hyphen encounters a -, it uses the standard input. Thus, hyphen may be used as a filter.

**EXAMPLES**

You would use the following command line to proofread nroff's hyphenation in *file*s:

```
mm file | hyphen
```

**LIMITATIONS**

The hyphen command cannot cope with hyphenated italics (or underlined words); it frequently will either miss them altogether or mishandle them.

The hyphen command occasionally gets confused but with no ill effects other than spurious extra output.

**FILES**

/usr/bin/hyphen
Executable file

**SEE ALSO**

grep(1), mm(1), troff(1)

**NAME**

id — displays user and group IDs and names

**SYNOPSIS**

```
id
```

**DESCRIPTION**

id writes a message on the standard output giving the user and group IDs and the corresponding names of the invoking process. If the effective and real IDs do not match, both are displayed.

**EXAMPLES**

If logged in as the user guest, the command:

```
id
```

will return

```
uid=100 (guest) gid=100 (users)
```

where 100 and guest are the user's ID number and name; 100 and users are the user's group ID number and group name. These values are set up in the administrative file /etc/passwd.

**FILES**

/usr/bin/id
: Executable file

/etc/passwd
: File containing user IDs

**SEE ALSO**

logname(1), whoami(1)

getuid(2) in *A/UX Programmer's Reference*

## NAME

ident — displays RCS keywords and their values

## SYNOPSIS

ident *file...*

## ARGUMENTS

*file* Specifies the file that is to be searched.

## DESCRIPTION

ident searches the named files for all occurrences of the pattern $*keyword*:... $, where *keyword* is one of the following:

```
Author
Date
Header
Locker
Log
Revision
Source
State
```

These patterns are normally inserted automatically by the RCS command co(1), but can also be inserted manually.

The ident program works on text files as well as object files.

## EXAMPLES

If the C program in file f.c contains the line

```
char rcsid[] = "$Header: utility $";
```

and f.c is compiled into f.o, then the command

```
ident f.c f.o
```

will print

```
f.c:
      $Header: utility $
f.o:
      $Header: utility $
```

## NOTES

Author: Walter F. Tichy, Purdue University, West Lafayette, IN 47907


## SEE ALSO

ci(1), co(1), rcs(1), rcsdiff(1), rcsintro(1), rcsmerge(1), rlog(1)

`rcsfile`(4) in *A/UX Programmer's Reference*

Walter F. Tichy, ''Design, Implementation, and Evaluation of a Revision Control System,'' in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, September 1982

## NAME

indent — indents and formats C program source

## SYNOPSIS

indent *input* [*output*] [-bc, -nbc] [-br, -bl] [-c*n*] [-cd*n*] [[-dj], -ndj] [-d*n*] [-i*n*] [-l*n*] [-v, -nv]

## ARGUMENTS

[-bc, -nbc]

Causes a newline to be forced after each comma in a declaration. If -bc is specified, a newline will be forced after each comma in a declaration. If -nbc is specified, this option is turned off. The default is -bc.

[-br, -bl]

Specifies the format of complex statements. If -bl is specified, complex statements will be lined up like this:

```
if (...)
{
    code
}
```

If -br (the default) is specified, they will look like this:

```
if (...) {
    code
}
```

-c*n*

Specifies the column in which comments will start. Replace *n* with the column number. The default is 33.

-cd*n*

Specifies the column in which comments on declarations will start. Replace *n* with the column number. The default is for these comments to start in the same column as other comments.

[-dj], -ndj

Specifies the justification of the declarations. -dj will cause declarations to be left justified. -ndj will cause them to be indented the same as code. The default is -ndj.

-d*n*

Controls the placement of comments which are not to the right of code. Specifying -d2 means that such comments will be placed two indentation levels to the left of code. The default, -d1, places comments one indentation level to the left of code. -d0 lines up these comments with the code. See the section on comment indentation following.

`-i`*n*
Specifies the number of spaces for one indentation level. Replace *n* with the number of spaces. The default is 4.

*input*
Specifies the file to be formatted.

`-l`*n*
Specifies the maximum length of an output line. Replace *n* with the length of the output line. The default is 75.

*output*
Specifies the results of the formatted *input* file. When specified, `indent` checks to make sure it is different from *input*. This option is not given, the formatted file will be written back into *input* and a ''backup'' copy of *input* will be written in the current directory.

`[-v],-nv`
Turns verbose mode on or off. `-v` turns on verbose mode, and `-nv` turns it off. When in verbose mode, `indent` will report when it splits one line of input into two or more lines of output, and it will give some size statistics at completion. The default is `-nv`.

**DESCRIPTION**

`indent` is intended primarily as a C program formatter. Specifically, `indent` can indent code lines, align comments, insert spaces around operators where necessary, and break up declaration lists as in `inta,b,c;`.

The `indent` command will not break up long statements to make them fit within the maximum line length, but it will flag lines that are too long. Lines will be broken so that each statement starts a new line. Comments will be lined up one indentation level to the left of the code, and an attempt is made to line up identifiers in declarations.

The options may appear before or after the file names. If *input* is named `/blah/blah/file`, the backup file will be named `.Bfile`.

You may set up your own ''profile'' of defaults to `indent` by creating the file `.indent.pro` in your home directory and including whatever switches you like. If `indent` is run and a profile file exists, then it is read to set up the program's defaults. Switches on the command line, though, will always override profile switches. The profile file must be a single line of not more than 127 characters. The switches should be separated on the line by spaces or tabs.

**Multiline expressions**

The `indent` command will not break up complicated expressions that extend over multiple lines, but it will usually correctly indent such expressions which have already been broken up. Such an expression might end up looking like this:

```
x =
        (
            (Arbitrary parenthesized expression)
            +
            (
                (Parenthesized expression)
                *
                (Parenthesized expression)
            )
        );
```

**Comments**

The `indent` command recognizes four kinds of comments. They are: straight text, `box` comments, UNIX-style comments, and unchanged comments. The action taken with these various types are as follows:

Straight Text

All other comments are treated as straight text. The `indent` command will fit as many words (separated by blanks, tabs, or newlines) on a line as possible. Straight text comments will be indented.

Box Comments

The `indent` command assumes that any comment with a dash immediately after the start of comment (i.e. `/*-`) is a comment surrounded by a box of stars. Each line of such a comment will be left unchanged, except that the first nonblank character of each successive line will be lined up with the beginning slash of the first line. Box comments will be indented (see below).

UNIX-style Comments

This is the type of section header which is used extensively in the UNIX system source. If the start of comment (`/*`) appears on a line by itself, indent assumes that it is a UNIX-style comment. It will be treated similarly to box comments, except the first nonblank character on each line will be lined up with the "`*`" of the `/*`.

Unchanged Comments

Any comment which starts in column 1 will be left completely unchanged. This is intended primarily for documentation header pages. The check for unchanged comments is made before the check

for UNIX-style comments.

**Comment Indentation**

Box, UNIX-style, and straight text comments may be indented. If a comment is on a line with code, it will be started in the `comment column` which is set by the `-c`*n* option. Otherwise, the comment will be started at *n* indentation levels less than where code is currently being placed, where *n* is specified by the `-d`*n* option. (Indented comments will never be placed in column 1.) If the code on a line extends past the comment column, the comment will be moved to the next line.

**LIMITATIONS**

The `indent` command does not know how to format ''long'' declarations.

**STATUS MESSAGES AND VALUES**

Status messages generally tell that a text line has been broken or is too long for the output line.

**FILES**

`/usr/ucb/indent`
Executable file

`~/.indent.pro`
Profile file

**SEE ALSO**

`cb`(1)

## NAME

indxbib — builds an inverted index for a bibliography

## SYNOPSIS

indxbib [*database*]... [*file*]...

## ARGUMENTS

*database*
Specifies the database from which to make the index.

*file* Specifies the file from which to make the index.

## DESCRIPTION

indxbib makes an inverted index to the named *databases* or *files* for use by lookbib and refer. These files contain bibliographic references (or other kinds of information) separated by blank lines.

A bibliographic reference is a set of lines, constituting fields of bibliographic information. Each field starts on a line beginning with a %, followed by a key-letter, then a blank, and finally the contents of the field, which may continue until the next line starting with %.

The indxbib command is a shell script that calls /usr/lib/refer/mkey and /usr/lib/refer/inv. The first program, mkey, truncates words to 6 characters, and maps uppercase to lowercase. It also discards words shorter than 3 characters, words among the 100 most common English words, and numbers (dates) < 1900 or > 2000. These parameters can be changed; see refer(1). The second program, inv, creates an entry file (.ia), a posting file (.ib), and a tag file (.ic), all in the working directory.

## LIMITATIONS

Probably all dates should be indexed, since many disciplines refer to literature written in the 1800s or earlier.

## FILES

/usr/ucb/indxbib
Executable file

*file*.ia
Output file where *file* is the name of the file or database

*file*.ib
Output file where *file* is the name of the file or database

*file*.ic
Output file where *file* is the name of the file or database

*file*.ig
Output file where *file* is the name of the file or database

**SEE ALSO**

addbib(1), lookbib(1), refer(1), roffbib(1), sortbib(1)

## NAME

ipcrm — removes interprocess communications facilities

## SYNOPSIS

ipcrm [-m *shmid*] [-M *shmkey*] [-q *msqid*] [-Q *msgkey*] [-s *semid*] [-S *semkey*]

## ARGUMENTS

-m *shmid*
: Removes the shared memory identifier *shmid* from the system. The shared memory segment and data structure associated with it are destroyed after the last detach.

-M *shmkey*
: Removes the shared memory identifier, created with key *shmkey*, from the system. The shared memory segment and data structure associated with it are destroyed after the last detach.

-q *msqid*
: Removes the message queue identifier *msqid* from the system and destroys the message queue and data structure associated with it.

-Q *msgkey*
: Removes the message queue identifier, created with key *msgkey*, from the system and destroys the message queue and data structure associated with it.

-s *semid*
: Removes the semaphore identifier *semid* from the system and destroys the set of semaphores and data structure associated with it.

-S *semkey*
: Removes the semaphore identifier, created with key *semkey*, from the system and destroys the set of semaphores and data structure associated with it.

## DESCRIPTION

ipcrm will remove one or more specified message, semaphore, or shared memory identifiers. The identifiers are specified by the options.

The details of the removes are described in msgctl(2), shmctl(2), and semctl(2). The identifiers and keys may be found by using ipcs(1).

## FILES

/bin/ipcrm
: Executable file

**SEE ALSO**

ipcs(1)

msgctl(2), msgget(2), msgop(2), semctl(2), semget(2), semop(2), shmctl(2), shmget(2), shmop(2) in *A/UX Programmer's Reference*

**NAME**

ipcs — reports interprocess communication facilities status

**SYNOPSIS**

ipcs [-a] [-b] [-c] [-C *corefile*] [-m] [-N *namelist*] [-o] [-p] [-q] [-s] [-t]

**ARGUMENTS**

-a Uses all print options. (This is a shorthand notation for the -b, -c, -o, -p, and -t options.)

-b Prints information on largest allowable size (maximum number of bytes in messages on queue for message queues, size of segments for shared memory, and number of semaphores in each set for semaphores). See below for meaning of columns in a listing.

-c Prints the creator's login name and group name.

-C *corefile*
Uses the file *corefile* in place of /dev/kmem.

-m Prints information about active shared memory segments.

-N *namelist*
Takes the argument as the name of an alternate *namelist* (/unix is the default).

-o Prints information on outstanding usage (number of messages on queue and total number of bytes in messages on queue for message queues and number of processes attached to shared memory segments).

-p Prints the process number information (process ID of last process to send a message, process ID of last process to receive a message on message queues, and process ID of creating process and process ID of last process to attach or detach on shared memory segments).

-q Prints information about active message queues.

-s Prints information about active semaphores.

-t Prints time information (time of the last control operation that changed the access permissions for all facilities; time of last *msgsnd* and last *msgrcv* on message queues, last *shmat* and last *shmdt* on shared memory, last semop(2) on semaphores).

**DESCRIPTION**

`ipcs` prints certain information about active inter-process communication facilities. Without options, information is printed in short format for message queues, shared memory, and semaphores that are currently active in the system.

If any of the `-q`, `-m`, or `-s` options are specified, information about only those indicated will be printed. If none of these are specified, information about all three will be printed.

The column headings and the meaning of the columns in an `ipcs` listing follow. The letters in parentheses indicate the options that cause the corresponding heading to appear, while the word in parentheses, *all*, means that the heading always appears. Note that these options determine only what information is provided for each facility; they do not determine which facilities will be listed.

`T` (*all*)
Type of the facility:

- `q` message queue
- `m` shared memory segment
- `s` semaphore

`ID` (*all*)
The identifier for the facility entry.

`KEY` (all)
The key used as an argument to *msgget*, *semget*, or *shmget* to create the facility entry.

*Note:* The key of a shared memory segment is changed to `IPC_PRIVATE` when the segment has been removed until all processes attached to the segment detach it.

`MODE` (*all*)
The facility access modes and flags. The mode consists of 11 characters that are interpreted as follows:

The characters are:

- `R` if a process is waiting on a *msgrcv*;
- `S` if a process is waiting on a *msgsnd*;
- `D` if the associated shared memory segment has been removed, it will disappear when the last process attached to the segment detaches it;

C if the associated shared memory segment is to be cleared when the first attach is executed;

\- if the corresponding special flag is not set.

The next characters are interpreted as three sets of three bits each. The first set refers to the owner's permissions; the next to permissions of others in the user-group of the facility entry; and the last to all others. Within each set, the first character indicates permission to read, the second character indicates permission to write or alter the facility entry, and the last character is currently unused.

The permissions are indicated as follows:

r if read permission is granted;

w if write permission is granted;

a if alter permission is granted;

\- if the indicated permission is not granted.

OWNER (*all*)
The login name of the owner of the facility entry.

GROUP (*all*)
The group name of the owner's group of the facility entry.

CREATOR (a,c)
The login name of the creator of the facility entry.

CGROUP (a,c)
The group name of the creator's group of the facility entry.

CBYTES (a,o)
The number of bytes in messages currently outstanding on the associated message queue.

QNUM (a,o)
The number of messages currently outstanding on the associated message queue.

QBYTES (a,b)
The maximum number of bytes allowed in messages outstanding on the associated message queue.

LSPID (a,p)
The process ID of the last process to send a message to the associated queue.

LRPID (a,p)
The process ID of the last process to receive a message from the associated queue.

STIME (a,t)
The time the last message was sent to the associated queue.

RTIME (a,t)
The time the last message was received from the associated queue.

CTIME (a,t)
The time when the associated entry was created or changed.

NATTCH (a,o)
The number of processes attached to the associated shared memory segment.

SEGSZ (a,b)
The size of the associated shared memory segment.

CPID (a,p)
The process ID of the creator of the shared memory entry.

LPID (a,p)
The process ID of the last process to attach, or detach, the shared memory segment.

ATIME (a,t)
The time the last attach was completed to the associated shared memory segment.

DTIME (a,t)
The time the last detach was completed on the associated shared memory segment.

NSEMS (a,b)
The number of semaphores in the set associated with the semaphore entry.

OTIME (a,t)
The time the last semaphore operation was completed on the set associated with the semaphore entry.

**LIMITATIONS**

Things can change while ipcs is running; the picture it gives is only a close approximation to reality.

**FILES**

/bin/ipcs
Executable file

/unix
System namelist directory

/dev/kmem
Memory file

`/etc/passwd`
File containing user names
`/etc/group`
File containing group names

**SEE ALSO**
`ipcrm`(1)
`msgop`(2), `semop`(2), `shmop`(2) in *A/UX Programmer's Reference*

*See* mactoiso(1)

## NAME

iw2 — prepares data to be printed on the Apple ImageWriter II printer

## SYNOPSIS

iw2 [-a *dotspace*] [-b] [-c *color*] [-d] [-D *udcfile*] [-f] [-h] [-k *mode*] [-l *language*] [-m *margin*] [-n *length*] [-o *file*] [-p *pitch*] [-q *quality*] [-s *spacing*] [-t *tabs*] [-u] [-U *udcfile*] [-w *value*] [-x] [-z] [*file*]...

## ARGUMENTS

-a *dotspace*
: Adds dot spaces to proportional pitch text. When the Apple Imagewriter II is printing in a proportional pitch, the space allotted to each character depends on the shape of the character. Each character has one dot space added after it to keep it from running into the next character. This option allows from 1 to 6 additional dot spaces to be *added* after each proportional character.

-b
: Prints boldface text. Each dot of the character is printed twice with a small shift of position.

-c *color*
: Prints text in color. The Apple Imagewriter II can print in color by using the color ribbon. The color ribbon contains four bands of color: yellow, cyan, magenta, and black. In addition, the Apple Imagewriter II automatically prints orange, green, and purple by overprinting one color with another, as follows:

  black
  : Selects the black color ribbon band.

  yellow
  : Selects the yellow color ribbon band.

  red
  : Selects the magenta color ribbon band. You can specify this color by magenta as well.

  blue
  : Selects the cyan color ribbon band. You can specify this color by cyan as well.

  orange
  : Prints orange by overprinting yellow and magenta.

  green
  : Prints green by overprinting yellow and cyan.

  purple
  : Prints purple by overprinting magenta and cyan.

-d
: Prints double-width characters. Each character is printed with double the number of dots with which it is normally printed.

`-D` *udcfile*
Works the same as the `-U` option, except that the *udcfile* filename is prefixed with the directory pathname `/usr/lib/iw2/` (see the `-U` option later in this section).

`-f` Outputs an initial formfeed before any files are printed. Generally used with the Apple Imagewriter II sheetfeeder.

*file* Specifies the file that is prepared for printing. If no file is specified, the standard input is assumed.

`-h` Prints half-height characters. Half-height characters are printed by cutting in half the vertical distance between the rows of dots that make up the characters.

`-k` *mode*
Selects print direction mode. The Apple Imagewriter II can print from left-to-right or bidirectional. Left-to-right, while slower, improves the precision at which characters line up.

`lr` Print left-to-right only.

`bi` Print bidirectional.

`-l` *language*
Selects language font. As an aid, there are 8 different language fonts used for printing text in other languages. Each of these fonts substitutes characters for these ten American font symbols:

`# @ [ \ ] ' { | } ~`

`american`
Selects the American language font.

`italian`
Selects the Italian language font.

`danish`
Selects the Danish language font.

`british`
Selects the British language font.

`german`
Selects the German language font.

`swedish`
Selects the Swedish language font.

`french`
Selects the French language font.

`spanish`
Selects the Spanish language font.

`-m` *margin*
Specifies the left page margin. This sets the leftmost column to start printing in. Normally zero, the column number may be set from zero (leftmost) to a value that depends on the current character pitch, as shown in the following list.

| Pitch | Chars/line | Range |
|---|---|---|
| 9 | 72 | 0 to 71 |
| 10 | 80 | 0 to 79 |
| 12 | 96 | 0 to 95 |
| 13.4 | 107 | 0 to 106 |
| 15 | 120 | 0 to 119 |
| 17 | 136 | 0 to 135 |
| pica | *depends* | 0 to 71 |
| elite | *depends* | 0 to 79 |

For setting the margin when using proportional fonts, elite uses 10 characters per inch and pica uses 12 characters per inch.

`-n` *length*
Specifies page length. This must be an integer value in inches. If the number is preceded by a `/`, it will be considered as *length*/144 in. That is, both `-n 11` and `-n /1584` will set a page length of 11 inches.

`-o` *file*
Specifies an output file. By default, `iw2` writes to the standard output, so this option will redirect the output to *file*.

`-p` *pitch*
Specifies pitch, or characters per inch. The Apple Imagewriter II prints in eight different widths (character pitches), from 9 characters per inch (cpi) to 17 cpi. Two of the character pitches print proportionally; that is, the space allotted to each character depends on the shape of the character.

`9` Prints at 9 cpi, for 72 characters per line.

`10` Prints at 10 cpi, for 80 characters per line.

`12` Prints at 12 cpi, for 96 characters per line.

`13` Prints at 13.4 cpi, for 107 characters per line. `13.4` may also be specified.

`15` Prints at 15 cpi, for 120 characters per line.

`17` Prints at at 17 cpi, for 136 characters per line.

`pica`
Prints pica proportional font. Averages 10 cpi.

`elite`
Prints elite proportional font. Averages 12 cpi.

`-q` *quality*
Specifies quality of printing. The Apple Imagewriter II can print ASCII text in one of three qualities: draft (250 characters per second), correspondence (180 cps), and near letter quality (45 cps).

`draft`
Prints in draft quality mode.

`better`
Prints in better, or correspondence quality mode.

`nlq`
Prints in best, near letter quality. You may also specify `best` for this mode.

`-s` *spacing*
Specifies spacing, or distance between lines. This value can be specified in two ways.

`2` Sets line spacing to 2 lines per inch.

`3` Sets line spacing to 3 lines per inch.

`4` Sets line spacing to 4 lines per inch.

`6` Sets line spacing to 6 lines per inch.

`8` Sets line spacing to 8 lines per inch.

`9` Sets line spacing to 9 lines per inch.

The value can also have a slash (/) affixed to it. Then, this value indicates line spacing at 1/144 in. For example, three lines per inch would be a spacing of 48/144 in., and could be specified by either `-s 3` or `-s /48`.

`-t` *tabs*
Specifies tab settings. Default tabs are set every 8 columns (9, 17, 25, ...). This option clears all default tab stops and is used to set custom tab stops. Tabs are specified by numbers followed by commas. For example, to set tabs every four columns (up to column 25):

```
-t 5,9,13,17,21,25
```

The limit on the number of settable tabs is 8. The highest legal column for the tab stop must lie in the left margin range. See the `-m`

option for the margin range table.

`-u` Causes all characters and spaces to be underlined.

`-U` *udcfile*
Loads user defined characters from the file *udcfile*, the contents of which are defined later in this section.

`-w` *value*
Sets dot spacing for proportional pitch text. When the Apple Imagewriter II is printing in a proportional pitch, the space allotted to each character depends on the shape of the character. Each character has a single dot space added after it to keep it from running into the next character. This option allows setting dot spaces for the proportional character set. Dot spacing may be set from 0 to 9 dot spaces. Each proportional character will always include one dot space, thus the settings of 0 through 9 allow you to set the dot spacing from 1 to 10.

`-x` Resets the Apple Imagewriter II initialization sequences (that set the default settings). In this program, first the default sequences are processed (see ''Defaults'' later in this section), then the environment variable, and then the options. This option, when encountered, resets the buffer holding the initialization sequences that were built by processing the default and environmental variable.

`-z` Specifies that all zeros are to be printed with a slash through them.

## DESCRIPTION

The Apple Imagewriter II is a dot matrix printer that works as a normal ASCII character set printer. It has many options, including color ribbons, various print qualities, national language character sets, downloadable fonts, and more. `iw2` is a program that accepts options indicating that a file or files (or standard input) is to be printed with various Apple Imagewriter II option sets.

The `iw2` command prepares the named files for eventual printing on the Apple Imagewriter II by sending appropriate Apple Imagewriter II control codes and then the named files to the standard output.

## UDC files

A UDC (user defined character) file consists of ASCII text that defines the bit patterns that make up a character. More than one character can be defined in a UDC file, and any character may be redefined. Characters that are not defined in a UDC file print out in the normal ASCII character bit pattern. For example, to define the ASCII space character (SP) to resemble an upside down and backwards capital L:

```
=040
```

```
1####.
2...#.
3...#.
4...#.
5...#.
6...#.
7...#.
8...#.
```

In a UDC file, each character is defined by 9 text lines. The first line starts with an equal sign (=), and is followed by an octal, decimal, or hexadecimal number that indicates the character to be defined. Octal, decimal, or hexadecimal is selected by using the standard C language conventions.

The next 8 lines define the 8 rows of the character. Notice that the lines are numbered. These numbers correspond to the nine-wire print head. You are limited to 8 rows. You can specify rows 1 through 8, or rows 2 through 9. Each line contains a period (.) to indicate no dot, and a pound sign (#) to indicate dot. The width of the character is computed by the longest line encountered in the 8 lines. You should place extra periods at the right columns of the character definition to allow for space between it and the adjacent character.

For example, we have redefined the letter ''A'' to be a vertical bar, with a small amount of space between it and the character on its left, and a lot of space between it and the character on the right.

```
=0x41
1.##...
2.##...
3.##...
4.##...
5.##...
6.##...
7.##...
8.##...
```

The maximum width of any character is 16 columns of dots.

**Defaults**

| | |
|---|---|
| Draft font | Standard ASCII |
| American language | Pitch is 12 cpi (Elite) |
| Black color | Set default tabs every 8 columns (12 cpi) |
| Stop double width print | Stop underlining |

| | |
|---|---|
| Stop boldface | Stop half-height text |
| Stop sub/super scripting | Zeros unslashed |
| Set left margin at 0 | Set page length to 11 inches |
| Bidirectional printing | 6 lines per inch spacing |
| Forward line feeding | Paper-out sensor on |
| Insert CR before LF/FF | No LF when line is full |
| CR, LF, FF cause printing | Ignore 8th data bit |
| Perforation skip disabled | Dot spacing is zero |

**Environment variables**

The environment variable

```
APPLE_IMAGEWRITER_II_PRINT_OPTIONS
```

can be used to supply default print options. All options may be specified in the environment variable. In the C shell, a typical setting of the environment variable would be

```
setenv APPLE_IMAGEWRITER_II_PRINT_OPTIONS\
                "-c red -q better"
```

**EXAMPLES**

The command:

```
iw2 -c red -q nlq -l british
```

will print text using the red color ribbon, in near letter quality (`nlq`) mode, using the British language font.

**NOTES**

When using the `-x` option, you specify character strings, as needed, to set various Apple Imagewriter II capabilities, without knowing the machine dependent codes. For example, if you wished to print a file, using `pr`(1), but wanted the header to be in red and the rest of the file in black, you could do the following:

```
set red='iw2 -x -c red < /dev/null'
black='iw2 -x -c black < /dev/null'
pr -h "$red this is the heading $black" $1 | lp
```

If you wanted to change the word ''red'' in the file `foobar` to print in the color red, you could do the following:

```
set red='iw2 -x -c red < /dev/null'
set black='iw2 -x -c black < /dev/null'
sed s/red/"$red"red"$black"/g foobar | lp
```

Always remember that you must set and unset the capability, or else the characters following what you have set will remain that way. Also note that in the `set red` and `set black` lines is the ‘ character (the ASCII character with the value of hexadecimal 60).

The -o option is ignored when iw2 reads from the standard input. If an input file is specified as an argument, then the -o option works as documented.

**FILES**

/usr/bin/iw2
Executable file

**SEE ALSO**

daiw(1), lp(1)

**NAME**

join — combines (joins) two relational files

**SYNOPSIS**

join [-a*n*] [-e *string*] [-j*n* *m*] [-o *list*] [-t*c*] *file1* *file2*

**ARGUMENTS**

-a*n*
: Produces a line for each unpairable line in file *n*, in addition to the normal output. Replace *n* with a 1 or a 2 which refers to either *file1* or *file2*, respectively.

-e *string*
: Replaces empty output fields with the string *s*.

*file1*
: Specifies the first file to be joined with *file2*.

*file2*
: Specifies the second file to be joined with *file1*.

-j*n* *m*
: Joins on the *m*th field of file *n*. If *n* is missing, use the *m*th field in each file. Fields are numbered starting with 1. Replace *n* with a 1 or a 2 which refers to either *file1* or *file2*, respectively.

-o *list*
: Causes each output line to comprise the fields specified in *list*, each element of which has the form *n* . *m*, where *n* is a file number and *m* is a field number. The common field is not printed unless specifically requested.

-t*c*
: Uses the character *c* as a separator (tab character). Every appearance of *c* in a line is significant. The character *c* is used as the field separator for both input and output. Note that this option must be used to preserve tabs and multiple spaces in a file.

**DESCRIPTION**

join forms, on the standard output, a join of the two relations specified by the lines of *file1* and *file2*. If *file1* is -, the standard input is used.

*file1* and *file2* must be sorted in increasing ASCII collating sequence on the fields on which they are to be joined, normally the first in each line.

There is one line in the output for each pair of lines in *file1* and *file2* that have identical join fields. The output line normally consists of the common field, then the rest of the line from *file1*, then the rest of the line from *file2*.

The default input field separators are blank, tab, or newline. In this case, multiple separators count as one field separator, and leading separators are ignored. Thus, to preserve tabs and multiple occurrences of spaces in a

file, you must select tabs as the alternate delimiter using the `-t` option where *c* is the tab character (see `-t` option above). The default output field separator is a blank.

**EXAMPLES**

If *file1* contains:

```
Austen -
Bailey -
Clark -
Dawson -
Smith -
```

and *file2* contains:

```
Austen Jack Anchor Brewery
Clark Maryann Shoeshop
Daniels Steve Computer Software
Dawson Sylvia Toot Sweets
Smith Sally Talcum Powdery
```

then the command:

```
join -j1 1 -j2 1 -o 2.2 2.1 1.2 2.3 2.4 file1 file2
```

will generate

```
Jack Austen - Anchor Brewery
Maryann Clark - Shoeshop
Sylvia Dawson - Toot Sweets
Sally Smith - Talcum Powdery
```

The command:

```
join -j1 4 -j2 3 -o 1.1 2.1 1.6 -t: /etc/passwd /etc/group
```

joins the password file and the group file, matching on the numeric group ID, and the login name, the group name, and the login directory. It is assumed that the files have been sorted in ASCII collating sequence on the group ID fields.

**LIMITATIONS**

With default field separation, the collating sequence is that of `sort -b`; with `-t`, the sequence is that of a plain sort.

The conventions of `join`(1), `sort`(1), `comm`(1), `uniq`(1) and `awk`(1) are wildly incongruous.

Filenames that are numeric may cause conflict when the `-o` option is used right before listing filenames.

**FILES**

/usr/bin/join
Executable file

**SEE ALSO**

awk(1), comm(1), sort(1), uniq(1)

**NAME**

kermit — invokes the Kermit file-transfer program

**SYNOPSIS**

kermit [-a *fn1*] [-b *n*] [-c] [-d] [-f] [-g *rfn*] [-h] [-i] [-k] [-l *dev* [-n]] [-p *x*] [-q] [-r] [-s *fn*] [-t] [-w] [-x] [*file*]...

**ARGUMENTS**

-a *fn1*

Specifies an alternative name for a single file if you have specified a file transfer option. For example,

```
kermit -s foo -a bar
```

sends the file `foo`, telling the receiver that its name is `bar`. If more than one file arrives or is sent, only the first file is affected by this option: For example:

```
kermit -ra baz
```

stores the first incoming file under the name `baz`.

-b *n*

Specifies the baud rate for the line given in the `-l` option, as in

```
kermit -l /dev/ttyi5 -b 9600
```

This option should always be used when the `-l` option is used, since the speed of an external communication line is not necessarily what you expect.

-c Establishes a terminal connection over the specified or default communication line, before any protocol transaction takes place. Get back to the local system by typing the escape character (normally CONTROL-Backslash) followed by the letter `c`.

-d Records debugging information in the file `debug.log` in the current directory. Use this option if you believe the program is misbehaving, and show the resulting log to your local `kermit` maintainer.

-f Sends a ''finish'' command to a remote server.

*file* Specifies the file to be moved.

-g *rfn*

Requests (actively) a remote server to send the named file or files; *rfn* is a file specification in the remote host's own syntax. If *fn* happens to contain any special shell characters, like `*`, these must be quoted, as in

```
kermit -g x\*.\?
```

-h Displays a brief synopsis of the command-line options.

`-i` Specifies that files should be sent or received exactly ''as is'' with no conversions. This option is necessary for transmitting binary files. It can also be used to slightly boost efficiency in UNIX-to-UNIX® transfers of text files by eliminating carriage-return line-feed/newline conversion.

`-k` Receives (passively) a file or files, sending them to standard output. This option can be used in several ways. Here are some examples.

```
kermit -k
```

displays the incoming files on your screen; this command is to be used only in ''local mode.'' (Local mode is described in ''Interactive Operation,'' later in this manual page.)

```
kermit -k > fn1
```

sends the incoming file or files to the named file, *fn1*. If more than one file arrives, all are concatenated together into the single file *fn1*.

```
kermit -k | command
```

pipes the incoming data (single or multiple files) to the indicated command, as in

```
kermit -k | sort > sorted.stuff
```

`-l` *dev*
Specifies a terminal line to use for file-transfer operations and terminal connection, as in:

```
kermit -l /dev/ttyi5
```

When an external line is being used, you might also need some additional options for successful communication with the remote system.

`-n` Acts like the `-c` option, but after a protocol transaction takes place. The `-c` and `-n` options can both be used in the same command.

`-p` *x*
Specifies the parity: `e`, `o`, `m`, `s`, or `n` (even, odd, mark, space, or none). If parity is other than none, then the 8th-bit prefixing mechanism will be used for transferring 8-bit binary data, provided the opposite `kermit` command uses the same mechanism. The default parity is none.

`-q` Specifies background mode (quiet); suppresses screen update during file transfer, for instance to allow a file transfer to proceed in the background.

`-r` Receives a file or files. Causes `kermit` to wait passively for files to arrive.

`-s` *fn*
Sends the specified file or files. If *fn* contains metacharacters, the A/UX shell expands *fn* into a list. If *fn* is `-`, then `kermit` sends from standard input, which must come either from a file or from a parallel process, as shown in these lines:

```
kermit -s - < foo.bar

ls -l | kermit -s -
```

You cannot use this mechanism to send the terminal type. If you want to send a file whose name is `-`, you can precede it with a pathname, as in

```
kermit -s ./-
```

`-t` Specifies half-duplex, line turnaround with XON as the handshake character.

`-w` Specifies write-protect; avoid filename collisions for incoming files.

`-x` Begins server operation. This option can be used in either local or remote mode.

**DESCRIPTION**

`kermit` is a file-transfer program that allows you to move files between computers with many different operating systems and architectures. This manual page describes version 4C of the program.

Arguments are optional. If `kermit` is executed without arguments, it enters command mode. Otherwise, `kermit` reads the arguments off the command line and interprets them.

The following notation is used in command descriptions:

[ ] Any field in brackets is optional.

{*x*,*y* ,*z*}
Alternatives are listed in braces.

*c* A decimal number between 0 and 127 representing the value of an ASCII character.

*cc* A decimal number between 0 and 31, or else exactly 127, representing the value of an ASCII control character.

*fn* Specifies an A/UX file specification, possibly containing the asterisk (*) metacharacter which matches all character strings, or the question mark metacharacter (?), ''?'', which matches any single character.

*fn1* An A/UX file specification that may not contain * or ?.

*n* A decimal number between 0 and 94.

*rfn* A remote file specification in the remote system's own syntax, which can denote a single file or a group of files.

*rfn1* A remote file specification that should denote only a single file.

The command-line options can specify either actions or settings. If kermit is invoked with a command line that specifies no actions, it issues a prompt and begins interactive dialog. Action options specify either protocol transactions or terminal connection.

The command line must not contain more than one protocol action option.

**Interactive Operation**

The interactive prompt for the kermit command is:

```
C-Kermit>
```

In response to this prompt, you can type any valid command. The kermit command executes the command and then prompts you for another command. The process continues until you tell the program to terminate.

Commands begin with a keyword, normally an English verb, such as send. You can omit trailing characters from any keyword, so long as you specify sufficient characters to distinguish it from any other keyword valid in that field. Certain commonly used keywords (such as send, receive, and connect) have special nonunique abbreviations (s for send, (r for receive, (c for connect).

Certain characters have special functions in interactive commands:

? A question mark, typed at any point in a command, causes kermit to display a message explaining what is possible or expected at that point. Depending on the context, the message may be a brief phrase, a menu of keywords, or a list of files.

\ Backslash; causes any of the other characters in this list to be entered into the command, literally. To enter a backslash, type two backslashes in a row (\\). A single backslash immediately preceding a carriage return allows you to continue the command on the next line.

CR Carriage return; enters the command for execution. A line-feed (LF) or form-feed (FF) can also be used for this purpose.

DEL
The DELETE or RUBOUT key; deletes the preceding character from the command. You can also use BS (CONTROL-H) for this function.

ESC
The ESCAPE or ALTMODE key; requests completion of the current

keyword or filename, or insertion of a default value. The result will be a beep if the requested operation fails.

^R CONTROL-R; redisplays the current command.

SP Space; delimits fields (keywords, filenames, numbers) within a command. HT (Horizontal Tab) can also be used for this purpose.

^U CONTROL-U; erases the entire command.

^W CONTROL-W; erases the rightmost word from the command line.

You can type the editing characters (DEL, ^W, and so on.) repeatedly, to delete all the way back to the prompt. No action will be performed until you enter the command by pressing RETURN, the line-feed key, or the form-feed key. Command is entered by typing carriage return, linefeed, or formfeed. If you make any mistakes, you will receive an informative error message and a new prompt; make liberal use of ? and ESC to feel your way through the commands. One important command is help; you should use it the first time you run kermit.

In interactive mode, kermit accepts commands from files as well as from the keyboard. When you enter interactive mode, kermit looks for the file .kermrc in your home or current directory (first looking in the home directory, then looking in the current one) and executes any commands it finds there. These commands must be in interactive format, not A/UX command-line format. A take command is also provided for use at any time during an interactive session. Command files can be nested to any reasonable depth.

Here is a brief list of kermit interactive commands:

! Executes an A/UX shell command.

bye
Terminates the connection to and log outs of a remote kermit server.

close
Closes a log file.

connect
Establishes a terminal connection to a remote system.

cwd
Changes the working directory.

dial
Dials a telephone number.

directory
Displays a directory listing.

`echo`
Displays arguments literally.

`exit`
Exits from the program, closing any open logs.

`finish`
Instructs a remote `kermit` server to exit, but not log out.

`get`
Gets files from a remote `kermit` server.

`help`
Displays a help message for a given command.

`log`
Opens a log file — debugging, packet, session, or transaction.

`quit`
Acts the same as `exit`.

`receive`
Passively waits for files to arrive.

`remote`
Issues file-management commands to a remote `kermit` server.

`script`
Executes a login script with a remote system.

`send`
Sends files.

`server`
Begins server operation.

`set`
Sets various parameters.

`show`
Displays values of `set` parameters.

`space`
Displays current disk-space usage.

`statistics`
Displays statistics about the most recent transaction.

`take`
Executes commands from a file.

The `set` parameters are as follows:

`block-check`
Specifies the level of packet error detection.

`delay`
Specifies how long to wait before sending the first packet.

`duplex`
Specifies which side echoes during `connect`.

`escape-character`
Specifies the character with which to prefix escape commands during `connect`.

`file`
Sets various file parameters.

`flow-control`
Specifies the communications line full-duplex flow control.

`handshake`
Specifies the communications line half-duplex turnaround character.

`line`
Specifies the communications line device name.

`modem-dialer`
Specifies the type of modem-dialer on the communications line.

`parity`
Specifies the communications line character parity.

`prompt`
Changes the `kermit` program's prompt.

`receive`
Sets various parameters for inbound packets.

`send`
Sets various parameters for outbound packets.

`speed`
Specifies the communications line speed.

The `remote` commands are as follows:

`cwd`
Changes the remote working directory.

`delete`
Deletes remote files.

`directory`
Displays a listing of names of remote files.

`help`
Requests help from a remote server.

`host`
Issues a command to the remote host in its own command language.

`space`
Displays current disk-space usage on the remote system.

`type`
Displays a remote file on your screen.

`who`
Displays who's logged in, or information about a user.

**Remote and Local Operation**

The `kermit` program is ''local'' if it is running on a personal computer or workstation that you are using directly, or if it is running on a multi-user system and transferring files over an external communications line, not from your job's controlling terminal or console. The `kermit` program is `remote` if it is running on a multi-user system and transferring files over its own controlling terminal's communications line, connected to your personal computer or workstation.

If you are running `kermit` on a personal computer, it is in local mode by default, with the ''back port'' designated for file transfer and terminal connection. If you are running `kermit` on a multi-user (time-sharing) system, it is in remote mode unless you explicitly point it at an external line for file transfer or terminal connection.

The `-g` *rfn*, `-f`, `-c`, and `-n` commands can be used only with a `kermit` program that is local, either by default or because the `-l` option has been specified.

On a time-sharing system, the `-l` and `-b` options must also be included with the `-r`, `-k`, or `-s` option if the other `kermit` program is on a remote system.

If `kermit` is in local mode, the screen (standard output) is continuously updated to show the progress of the file transfer. A dot is printed for every four data packets; other packets are shown by type (for example, `S` for Send-Init); `T` is printed when there's a timeout; and `%` is printed for each retransmission. In addition, you can type (to standard input) the following ''interrupt'' commands during file transfer:

CONTROL-F
Interrupts the current file and goes on to the next (if any).

CONTROL-B
Interrupts the entire batch of files and terminates the transaction.

CONTROL-R
Resends the current packet.

CONTROL-A
Displays a status report for the current transaction.

These interrupt characters differ from the ones used in other kermit implementations to avoid conflict with A/UX shell interrupt characters. With System III and System V implementations of the UNIX system, interrupt commands must be preceded by an escape character (such as CONTROL-\).

**LIMITATIONS**

See recent issues of the Info-Kermit digest (on ARPANET or Usenet) for a list of bugs.

**STATUS MESSAGES AND VALUES**

The diagnostics produced by kermit itself are intended to be self-explanatory.

**FILES**

$HOME/.kermrc
File containing kermit initialization commands

./.kermrc
File containing kermit initialization commands

/usr/bin/kermit
Executable file

/usr/spool/locks
Directory in which kermit makes a lock file, which prevents other programs from using the serial port that kermit is using

**SEE ALSO**

cu(1C), uucp(1C)

*Kermit User's Guide*, Frank da Cruz and Bill Catchings, Columbia University, 6th Edition

## NAME

kill — terminates a process

## SYNOPSIS

kill [*-sig*] *pid...*

## ARGUMENTS

*pid* Specifies the process identification number (*pid*) of the process to be killed.

*-sig*

Sends the corresponding signal instead of terminate (see signal(3)),ifthe *-sig* option is given. In particular kill -9... is the surest kill; especially with NFS, the 9 signal does not always destroy the process.

## DESCRIPTION

kill sends signal 15 (terminate) to the specified processes. This will normally kill processes that do not catch or ignore the signal. The process number (*pid*) of each asynchronous process started with & is reported by the shell (unless more than one process is started in a pipeline, in which case the number of the last process in the pipeline is reported). Process numbers may also be found by using ps.

Details of the kill are described in kill(2). For example, if process number 0 is specified, all processes in the process group are signaled.

The to-be-killed process must belong to the current user unless he is the superuser.

Similar versions of kill are built into ksh(1) and csh(1).

## EXAMPLES

The command:

```
kill 24068
```

sends signal 15 to the process with the ID number 24068.

## FILES

/bin/kill

Executable file

## SEE ALSO

ps(1), sh(1), csh(1), ksh(1)

kill(2), signal(3) in *A/UX Programmer's Reference*

## NAME

ksh — runs the Korn shell, an enhanced command interpreter that is backward-compatible with the Bourne shell (sh)

## SYNOPSIS

ksh [-a] [-c *string*] [-e] [-f] [-h] [-i] [-k] [-m] [-n] [-o *option*]... [-p] [--*positional-arg*]... [±*positional-arg*]... [-r] [-s] [-t] [-u] [-v] [-x] [*file*]...

## ARGUMENTS

+ [*positional-arg*]...

- [*positional-arg*]...
  Turn off -x and -v options and suppress examination of remaining shell arguments for interpretation as command options. Instead, the positional variables of the shell are reassigned in terms of the *positional-arg* values supplied, which makes it possible to set $1 to a value beginning with a hyphen.

-- [*positional-arg*]...
  Reassigns the positional variables. The remaining arguments are positional parameters and are assigned, in order, to $1, $2, and so on. If no arguments follow this option, the positional parameters are unset.

-a Exports the new values for any subsequently modified variables without your having to use an explicit export command.

-c *string*
  Specifies a command line (*string*) that you want the ksh subshell to run before exiting. Using this option is one way to run the Korn shell noninteractively.

-e Executes the command associated with the ERR condition by the trap command, if any was set. Exits if a command returns false. This mode is disabled while the system is reading the startup files (.kshrc and .profile).

-f Disables filename generation. (See ''Filename Generation'' in the ''Description'' section later in this manual page.)

*file* Specifies the filename for a shell script you want the ksh subshell to run before exiting.

-h Affects the way the built-in alias command builds aliases. The command-name portion of the aliased command is expanded into a full pathname so that the alias can continue to locate the same command program even when identically named commands are later created and located in directories that are also part of the search path. An alias built in this fashion is a tracked alias. See ''Aliasing'' in the ''Description'' section for more information about tracked aliases.

`-i` Establishes an interactive mode of operation.

`-k` Establishes a mode of operation where any command-line elements that are variable assignments are placed in the environment of a command, regardless of their position on the command line. Otherwise, only variable assignments that precede the command name are placed in the environment of a command.

`-m` Causes background jobs to run in a separate process group that is not associated with the terminal. The exit status of background jobs is reported in a completion message. This mode is turned on automatically for interactive shells.

`-n` Reads commands and checks them for syntax errors, but does not execute them. This option is ignored for interactive shells.

`-o` [*option*]
Puts into effect the option specified.

With no *option* arguments, `-o` prints the current preference settings. The option letters corresponding to the active settings are also merged into a single string that is stored in the variable named ''hyphen'' (`$-`). To set or unset these options once an interactive session is underway, use `set` as described in ''Built-in Commands'' in the ''Description'' section later in this manual page. You can replace *option* with any of the following values:

`allexport`
Exports variables automatically whenever they are reset; establishes the same operating mode as does the `-a` option.

`bgnice`
Runs all background jobs at a lower priority.

`emacs`
Establishes an `emacs`-style, command-input editor for command entry.

`errexit`
Executes the `ERR` trap; establishes the same operating mode as does the `-e` option.

`gmacs`
Establishes a `gmacs`-style, command-input editor for command entry.

`ignoreeof`
Prevents `ksh` from exiting when an end-of-file character is received. The `exit` command must be explicitly executed.

`keyword`
Affects the way parameters are placed in the environment; establishes the same operating mode as does the `-k` option.

`markdirs`
Appends a trailing slash (`/`) to all directory names resulting from filename generation.

`monitor`
Alters the usual process group assigned to background jobs; establishes the same operating mode as does the `-m` option.

`noclobber`
Prevents the overwriting of existing files when output is redirected to files.

`noexec`
Checks command lines without executing them; establishes the same operating mode as does the `-n` option. Using `noexec` helps you determine whether `ksh` can interpret an input data stream as valid commands.

`noglob`
Disables filename generation; establishes the same operating mode as does the `-f` option.

`nolog`
Prevents `ksh` from saving function definitions in the history file.

`nounset`
Causes an error to be reported when an uninitialized variable is referenced within a command line; establishes the same operating mode as does the `-u` option.

`privileged`
Resets `PATH` to the default search path; establishes the same operating mode as does the `-p` option.

`trackall`
Tracks aliases by way of storing the full pathname to the aliased command; establishes the same operating mode as does the `-h` option.

`verbose`
Prints each command line (exactly as it appears in the input) before it is executed; establishes the same operating mode as does the `-v` option.

`vi` Establishes a command-line-input editor for command entry. This is also called the ''cooked'' (processed) mode. This editor

has fewer features than the viraw editor, and thus has faster response time than it.

viraw
Establishes a command-character-input editor for command entry similar to the vi argument, except that input is processed on a character-by-character basis. This editor mode is also known as the ''raw'' mode. The viraw editor has more features and is more reliable, but causes longer response times for all users on multi-user systems. This editor has horizontal scrolling and Tabs are always expanded.

xtrace
Prints each command and its arguments after those arguments have been processed for metacharacters but just prior to the execution of the command; establishes the same operating mode as does the -x option.

-p Resets the PATH variable to the system default value, disables processing of the $HOME/.profile file, and uses the file /etc/suid_profile instead of the ENV file. This mode is automatically enabled whenever the effective user ID (or group ID) is not equal to the real user ID (or group ID).

-r Restricts certain shell functions. The following actions are not allowed: (1) changing the directory; (2) setting the value of SHELL, ENV, or PATH; (3) specifying path or command names containing /; and (4) redirecting output by using > or >>. These restrictions are enforced after the .profile and ENV files are interpreted.

When you enter a command that ksh determines to be a shell script, the restricted shell invokes another instance of ksh to execute it. These secondary Korn shells are not restricted as is the parent shell. Thus, the restricted form of ksh allows shell scripts to run with more complete privileges despite the limitations of the parent shell.

By administering the account so that certain setup actions are placed in .profile (where these restrictions are not yet enforced), an operating environment is established that precisely limits the actions that can be taken by its users. One of these setup actions should be to set the working directory to a designated directory other than the login directory. To maintain these security limitations, the so-designated working directory should deny the restricted account write permission. If write permission is granted for the designated working directory, then umask should be set to deny execute permission for any new files.

Also, the system administrator usually sets up a new directory of commands. For example, /usr/rbin can be created and a limited number of links placed there so that /usr/bin can be made inaccessible for restricted accounts through their PATH variable setting in .profile. Applications intended for restricted account use should also be placed there.

-s Ends the interactive mode of entering commands from standard input. Also, sorts the positional parameters when used with set -s, but not on the command line.

-t Exits after reading and executing one command.

-u Treats the presence of unset parameters as an error when substitution is necessary.

-v Prints command lines exactly as they are read from the input, before shell metacharacters are interpreted.

-x Prints commands and their arguments after shell metacharacters are interpreted, but prior to the execution of the command.

**DESCRIPTION**

ksh is a shell (or command interpreter) that accepts and dispatches command lines. It is largely responsible, along with CommandShell, for supporting the command-line interface of A/UX. Like the Macintosh Finder, the shell allows you to select the A/UX program or utility you want to run next, or to run it in conjunction with other programs that are already running.

The ksh program is also one of the A/UX commands that you can run once an initial shell (command interpreter) is running. Using ksh this way is one method for switching between the different shell programs available. (See csh(1) and sh(1) for information about these other shells.) When you run the ksh command, the previously interactive shell is suspended until you exit the ksh subshell.

To enter commands, you normally open a CommandShell window. When you do, CommandShell runs a shell to support the command-entry function in the window. The choice of shell is controlled by the preference variable SHELL, which is normally initialized along with other startup values in .login or .profile in your home directory. If SHELL is not set in one of those startup files, the shell spawned will be the same as your login shell (as described in passwd(4) and chsh(1)).

This manual page treats ksh just like any other command despite the fact that you need a shell program to support the invocation of commands in the first place. You should become familiar with one or more of the shells to allow you to take advantage of the command capabilities of A/UX. As a

prerequisite, you should learn about the CommandShell application (described in `CommandShell`(1) and *A/UX Essentials*). CommandShell supports the more visible and Macintosh-like elements of your system: the command windows, the mouse, and the menu functions at your disposal when you enter command lines.

### Using `ksh`

For the shell program, the work of interpreting the commands you enter can be broken down into several steps: (1) prompting for and accepting lines of input; (2) deciphering the text of an input line, one unit of which is expected to be the name of a command; and (3) locating and running the (object) file containing the low-level instructions that give the command its functionality.

The following three subsections that follow briefly discuss each of the three steps that make up a single computer-human interaction as mediated by a shell program (running interactively).

**Step one: obtaining input.** To indicate its readiness to process a command, the shell displays a prompt message or symbol at the beginning of the line. You contribute to the text displayed on the command line by typing a command name after the prompt. During command entry, the shell displays each character you type, placing it at the end of the command string and advancing the location of the cursor. During this time, the shell honors special characters that are not intended as part of the command string, such as the delete character (generated by the DELETE key). This ''silent'' conversation between you and the shell is limited to certain line-editing operations as well as the processing of other special characters such as the interrupt and end-of-line characters. The shell interprets these characters and takes appropriate action. Often this action changes the composition of the command string being displayed, as in the case of a delete character.

The end-of-line (newline) character is generated when you press the RETURN key. When the shell received this character, it considers the command line to be complete. The command-line processing that the shell performs next is described in the next two subsections. While processing a command line, the shell does not display a new prompt, although the cursor may already be at the beginning of a new line. At any point during command-line processing and command execution until you see a new command prompt signaling completion, `ksh` honors an interrupt signal. Sometimes the generation of an interrupt signal results in the partial execution of the command or, if you are very quick, in no execution of the command. The interrupt character is typically the CONTROL-C key combination. See ''Controlling Foreground Jobs'' later in the ''Description'' section.

**Step two: deciphering input.** The shell can recognize and correctly interpret a variety of command-line elements. Only very experienced programmers will know how to make the best use of all of the constructs that the shell is able to interpret. Most users do not need to learn all of these features in order to build useful command lines.

The simplest acceptable command entry consists of the name of a command with no other elements, as shown in this example. (The `$` symbol is the command prompt in this example.)

```
$ date
Tue Jun 18 12:01:25 PDT 1991
$ ■
```

In addition to the command name, command body may be required, depending on the command you are entering. The command body is subject to some processing by the shell, followed by final processing performed by the command program itself. The command body is typically broken down into two major elements, command options and arguments. The command options are typically individual letters. If you want to enter more than one command option, you normally merge the options into one character string. Whether you specify one or more command options, you must usually precede each option with a hyphen (-). Following the command options are the command arguments. Each argument is a string of characters, separated from one another or from the command options by a space or tab character, as shown here:

*command -options arguments*

When merged together, two or more command options can also be considered a single command argument, which can be formed into one string. With most commands, however, the options can be supplied as multiple space-separated strings, each consisting of a hyphen followed by a particular option letter:

*command* `-a -f -v`

The following command line includes one command option (`-l`) and one command argument (`memofile`) that is not a command option:

```
ls -l memofile
```

Most of the syntax descriptions that follow use the term *command* to refer to both the command name and its options and arguments.

Often you can save yourself typing by relying on the shell to preprocess the command arguments in terms of substituting one element for lengthier text to which it refers. Sometimes one element will represent a lengthier replacement that is actually several distinct arguments. Processing of this

type is sometimes called a ''substitution.''

In the ksh shell, an interesting form of substitution is the use of an alias, which is a brief way to refer to a longer command line. You create these alias names by using the alias command. See ''Aliasing'' later in the ''Description'' section, for more information regarding aliases.

Substitutions other than aliasing require a metacharacter to help trigger substitution of the appropriate text. To indicate that you are making a reference to a variable, you precede the variable name with a dollar sign ($) metacharacter. For aliases, a substitution is also performed, but with a notable difference in the request format: No metacharacter is required because the name being used as an alias must be placed at the beginning of a command line, and ksh always checks the command name at the beginning of a line to see if it is a previously defined alias. Unlike aliases, variable references can be placed at a point other than the beginning of a command line and still trigger a substitution.

When ksh finds a properly positioned alias, it replaces it with the command name and command body that the alias was set to represent. If you had placed another command body after the alias at the beginning of the line, the new command body is added to the end of the command body (if any) that was stored for the alias. Note that the required positioning of command options ahead of other command arguments can become disturbed during the substitution process. You should plan the use of an alias to which you expect to append arguments so that you can define it in such a way that the enclosing command line that you supply extends the aliased command in a legal manner. For aliases that are associated with multiple commands, the last-referenced command is the only one that is subject to extension when the alias name is substituted.

Another form of substitution applies to specially delimited subcommands:

$((*command*))

Unlike substitutions of variables and aliases, which have static values that you assigned to them at some earlier time, command substitutions create replacement text by executing a command that generates output text. This text can reflect the system state precisely at the current moment. See ''Command Substitution'' later in the ''Description'' section, for more information about this type of substitution.

Shell metacharacters are processed by the shell rather than by a command program. By processing shell metacharacters, the shell shifts the determination of the user interface away from individual command programs, and a more consistent command-line interface is easier to achieve. The benefit for you is that one shell-supported meta syntax can be applied to a number of command lines. After learning this metacharacter-

based syntax, you can apply it very broadly to most of the commands you use.

**Step three: dispatching other programs.** Completion of this step is closely related to the shell's ability to complete the previous step (deciphering of input). For example, if it cannot find a correctly entered command name (or Korn shell alias), the shell cuts this step short. In this case, the shell displays an error message instead of dispatching the program corresponding to a command. After the error message, the shell displays a new command prompt to initiate the next computer-human interaction.

Suppose you included the ampersand metacharacter (`&`) on a command line. In step two, the shell detects its presence and removes it from the argument string that is passed to the command. In step three, the shell alters the way it dispatches the command program because of the metacharacter's presence. It uses a special invocation mode called ''background mode.''

When a command is invoked in background mode, the shell does not wait for it to complete before initiating a new computer-human interaction. Rather, the shell prompts immediately for a new command, and any work initiated by the last command is performed concurrently. Thus, there is an immediate transition from step three of the current computer-human interaction to step one of a new computer-human interaction. You must use a dedicated command to delete background-mode processes on those occasions when the background process does not self-terminate, or when you want to stop its execution prior to its completion. (See the description of `kill` in ''Built-in Commands,'' near the end of the ''Description'' section.) You can also bring a background job back to the foreground as described in ''Controlling Jobs Not in the Foreground,'' later in the ''Description'' section.

### Format of Command Lines

The `ksh` program has certain restrictions on the ordering of elements of commands. These restrictions make their interpretation easier and their format more regular.

The metacharacters are often oddball characters that would not normally be a part of the command you are entering. These characters are less likely to be confused with command options or arguments.

Many of the specially interpreted metacharacters help you enter and run commands more efficiently, for example by avoiding lengthy typing. Once you learn to use the shell well, you will be able to enter shorter commands that nevertheless take advantage of very specialized processing or processing modes. For example, a terse notation is used to indicate that you want to direct the output of a command into a file, or to concurrently run one program with other programs already running. Another advantage is

that these metacharacter-triggered changes in processing are initiated in the same way for almost all commands.

A disadvantage is that the oddball metacharacters create strange-looking command lines. Another disadvantage is that they may be difficult to memorize. When you want them to be treated literally, these metacharacters must be specially delimited, which also adds to the strange appearance of some command lines.

This section describes the `ksh`-imposed rules for the structure of command lines. One restriction is that the command name must precede the command options and arguments. There are other requirements as well, such as the use of a command delimiter when you want to enter multiple commands on one line.

Some of the shell metacharacters that you must use with care inside command lines are:

`;` `&` `|` `$` `(` `)` `<` `>` `~` `?` `*` `[` `]` *newline space tab*

Not all of the functions of these metacharacters are described in this section. It makes sense to introduce some of them later, where they can be discussed along with some of the more advanced topics with which they are associated. (Additional metacharacter tokens are also described in ''Additional Korn Shell Metacharacters,'' near the end of the ''Description'' section.)

Spaces and tabs are both referred to as ''white space,'' and one is as acceptable as the other when white space is required. White space is required between the command name and its (sometimes optional) command arguments:

*command-name white-space command-arg*

(Whether or not arguments are optional depends on the individual syntax requirements for particular commands.) White space is also used to delimit command arguments when you want to specify several of them:

*command-name* [*white-space command-arg*]...

For more information regarding the treatment of white space characters, see ''Argument Parsing,'' later in the ''Description'' section.

Command separators are one type of shell metacharacter. They permit the specification of more than one command in the same line. The semicolon is interpreted as this type of metacharacter.

The ampersand is also a command separator, with added functionality. It establishes background mode for the preceding command. When you establish background mode with an ampersand, do not include a semicolon

as well. Because of this exclusivity, two syntax descriptions are needed to show the legal command syntaxes involving these command separators:

*command* [ ; *command*]...

*command* [& *command*]...

With each command, you can specify an input and output redirection. Thus you can expand each occurrence of *command* as follows:

*command* [*redirect-in*] [*redirect-out*]

The value of the *redirect-out* element is the metacharacter > (greater-than sign) followed by a filename:

> *output-file*

The value of the *redirect-in* element is the metacharacter < (less-than sign) followed by a filename:

< *input-file*

Another form of redirection involves multiple commands that share an information flow, bypassing the need for intermediate files. Consider these two commands:

```
who >/tmp/data
grep tty1  </tmp/data
```

The preceding sequence is equivalent to a pipe joining the two commands as a single processing request, as follows:

```
who | grep tty1
```

The output of the first command (`who`) is channeled directly to the input of the second command (`grep`). The pipe metacharacter (a vertical bar) indicates this channeling of data between commands. You can extend the pipeline to channel the output of a second command to the input of a third command, and so forth:

*command* |*command* [|*command*]...

While syntactically legal, file redirections inside a series of pipelined commands can conflict with the redirection established by the pipe, as in this example:

*command1* >*file* |*command2* |*command3*

In this case, the input channeled to *command2* is empty because the output of *command1* is redirected to *file* first. See `tee`(1) if you need to channel a data stream to a file as well as into another command.

Here is the the general format of a sensibly constructed pipeline, with no data redirection conflicts:

*command* [<*file1*] [|*command*]... |*command* [>*file2*] [&]

Note that a processing pipe such as this is equivalent to one processing job, particularly in terms of job control (as described next in ''Controlling Foreground Jobs'' and ''Controlling Jobs Not in the Foreground''). You cannot use command separators except at the end of a pipeline. By putting an ampersand at the end of a pipeline, you place in background mode all of the processing of the pipelined commands that precede the ampersand. If you use a semicolon instead of an ampersand, the pipelined commands are executed completely; after that, the command (or another pipeline) after the semicolon is executed.

The processing request shown here illustrates how the pipe character causes a subsequent newline to be ignored (rather than treated as the end-of-command character).

```
$ who |
> grep console
mikee      console      Jun 17 10:17
$ ■
```

For brevity of notation, the term *command* is used to represent any single- or multiple-command line, such as a pipeline, along with any file redirections.

**Controlling Foreground Jobs**

For jobs running in the foreground, a measure of control is available through certain control characters. To discontinue execution of a foreground command that is being processed, you can use the interrupt character. To discontinue processing of an interactive shell other than the login shell, you can use an end-of-file character (or the `exit` command).

The interrupt character that stops a foreground command does not affect commands that are running in the background. For background processes, also known as ''jobs,'' other control provisions are required. These provisions are also known as ''job-control facilities.'' Rather than responding to control characters (such as interrupt characters), jobs respond to signals that are explicitly sent to them by discrete, signal-sending commands. Because there can be more than one background job running at the same time, you must also specify a command argument that can identify the job. To assist you, commands are available to display the numbers associated with jobs, including one called `jobs`. (See the next section that follows this one for related information.)

Two ways of controlling jobs are necessary because there are two kinds of running jobs: foreground jobs and background jobs. Because foreground jobs execute one at a time, any command-based means of job control is awkward for controlling a foreground job. Instead you must use a key that generates an interrupt character to terminate a foreground job prematurely. From within the same interactive shell, you cannot cause a newly entered command to be examined while the prior foreground command is still running. (Normally you wait, until a new command prompt appears before you even begin to type another command.)

Besides quitting a foreground job prematurely, you can also suspend its execution, normally by pressing CONTROL-Z. Later, you can resume its execution by using the job-control commands of `ksh`.

To see what key combinations produce various control characters, enter the `stty` command with the `-a` option. In the output of `stty`, the caret (^) is used to represent the CONTROL key. By specifying other arguments, you change the mappings of keys to control characters (see `stty`(1)).

When you suspend a job, `ksh` prints a short status line about the job before it issues the next prompt. Among other things, the status line reports the `ksh`-assigned job number (enclosed in brackets). By supplying that number as an argument to other built-in commands, such as `bg` and `fg`, you can further manipulate the state of the suspended job. For instance, you can resume its execution in the background by entering this command format:

`bg %`*job-number*

When you no longer want to enter new commands, you may want to place a stopped job or a background job back in the foreground. To do so, use this command format:

`fg %`*job-number*

For more information about the `fg` command, see ''Built-in Commands,'' near the end of the ''Description'' section. For more information about monitoring both stopped jobs and jobs running in background mode, see the next section.

**Controlling Jobs Not in the Foreground**

To help you control and monitor running and stopped background jobs, `ksh` keeps track of the state of each job.

This job-tracking service of `ksh` helps keep you informed about the progress of jobs and helps prevent you from losing track of running jobs. For example, if you try to exit from the shell while jobs are suspended, you receive this warning:

```
You have stopped (running) jobs
```

You can use the `jobs` command to see which jobs are suspended. If you try the exit command again, the shell does not warn you a second time, and the suspended jobs are terminated.

Whenever a job becomes blocked and no further progress is possible, `ksh` informs you of its status. This information is made to appear just before a new shell prompt so that you can better distinguish it from the output of other commands. This particular job-tracking feature can be extended so that any background job that needs to display output similarly stops running, as described in the paragraphs later in this section that discuss the `tostop` argument for `stty`.

Other customizations are also possible. For example, when you use the `monitor` option, each background job can be set to trigger another command upon its completion. Triggering commands requires setting a trap for the `CHLD` signal. (See the description of `trap` in "Built-in Commands" near the end of the "Description" section.)

Using the `ksh`-assigned job number as an argument to certain built-in commands, you can place existing jobs in the foreground or background (restarting them in the process). You can determine the `ksh`-assigned job number in two ways: (1) You can use the built-in `jobs` command to obtain a numbered list of jobs. (2) You can notice the job number `ksh` assigns to a command line that is run in the background when `ksh` displays its status immediately after entry. The format of this status message is as follows:

[*jobno*] *process-id*

This line indicates that the command just entered is running in the background (asynchronously); has the process ID shown; and can be referenced as *jobno* for use along with the built-in, job control commands of `ksh`.

To control jobs with commands that are not shell built-ins, you must use the process ID to refer to the process. The process ID is also reported by the process status command, `ps`, which is described in `ps`(1).

When you use the built-in `ksh` commands for controlling jobs, you reference a job by using a job number, prefixed by the percent character (`%`). For instance, to place job number 1 in the foreground, enter

```
fg %1
```

The argument `%%` or `%+` can be used to refer to the most recent background job. The argument `%-` refers to the next-most-recent job.

You can also reference a job by using a string that matches the command name you originally entered to begin the job. Thus, the following command restarts a suspended `ed` job, provided a suspended job whose name begins with the string `ed` is present:

```
fg %ed
```

Similarly, the following command format resumes any job whose original command line contained the string *old-command-substring*.

```
fg %?old-command-substring
```

The shell keeps track of the most recent job. In `ksh` messages about jobs and in the report displayed by the `jobs` command, the most recent job is prefixed with a plus sign (+) and the next-most-recent job is prefixed with a minus sign (-). The following processing request illustrates this point:

```
$ sleep 525 &
$ sleep 330 &
$ sleep 250 &
$ jobs
[3] +  Running                 sleep 250 &
[2] -  Running                 sleep 330 &
[1]    Running                 sleep 525 &
$
```

The process running in the foreground has fairly exclusive access to input entered at the terminal. If a foreground command is underway, then the shell shares access to input typed at the terminal with the foreground command. By sharing access to input, the shell still has a chance to interpret certain control characters, such as the interrupt character, or to accept characters of the next command to be run (a type-ahead feature). However, a job running in the background cannot continue to run if it requires user input, because it does not have access to any of the data typed at the keyboard.

Thus, when any of the background jobs requires user input, the shell stops the job. You can resume its execution by making it the foreground job and then supplying it with the data that it requires. To make it the foreground job, you must enter a command, so you will have to wait until any foreground job still underway completes, or you will have to interrupt it.

Background jobs are normally allowed to send output to the terminal (or associated CommandShell window) without interference from `ksh`. However, this manner of operation can be disabled, so that `ksh` stops the execution of any job that needs to display a message. To establish the shell as a moderator for background jobs that are about to display output, enter

```
stty tostop
```

If you have used the `monitor` option (as described in the next section, "Establishing Preference Settings,"), your interactive shell associates a job with each pipeline.

A/UX provides another set of job-control commands that perform many of the same functions made available by the built-in commands of `ksh`. An advantage of using the discrete job-control commands such as `kill`, `nice`, and `ps` is that they are always available, even when you change to a shell other than `ksh`. These commands are described as separate entries elsewhere in the *A/UX Command Reference*. These discrete A/UX commands use a process ID number to identify jobs. However, the `ksh`-assigned job number is usually much shorter than the process ID number, and the built-in commands offer you greater simplicity.

### Establishing Preference Settings

You can establish preference settings in several ways. For example, you can reduce the likelihood of overwriting an existing file with a new file of the same name by establishing the `noclobber` option, as follows:

```
set -o noclobber
```

To unset this preference, enter

```
set +o noclobber
```

If you are able to start the shell yourself, you can request the same preference on the command line, as follows:

```
ksh -o noclobber
```

Preferences such as `noclobber` are either on or off (set or unset, established or unestablished, and so on). You can determine the state of these preferences by using `set` with the `-o` option, as shown here:

```
$ set -o
Current option settings
allexport        off
bgnice           off
emacs            off
errexit          off
gmacs            off
ignoreeof        off
interactive      on
keyword          off
markdirs         off
monitor          on
noexec           off
```

```
noglob          off
nounset         off
protected       off
restricted      off
trackall        off
verbose         off
vi              on
viraw           on
xtrace          off
$ ■
```

Other preferences, particularly those that can assume more states than on or off, can be stored in variables. To switch to a different command-input editor, you can make an assignment such as this one:

```
EDITOR=vi
```

This particular variable assignment establishes `vi` as the command-input editor you wish to use. You could establish other editing styles, such as `emacs` and `gmacs`, in a similar way.

Other preferences with many possible values are handled through commands built into `ksh` rather than through variables. For instance, you use the built-in commands `uulimit` and `umask` to establish operating limits (such as maximum file size) and default permissions for new files. For more details, see ''Built-in Commands'' near the end of the ''Description'' section.

The ways of selecting preferences described so far do not make those settings permanent. They are in effect only as long as you use the shell into which you entered them. To retain these settings between uses of various login shells (after logging out and logging back in), you need to place them in a ''shell startup'' file. For `ksh`, the startup file is `.profile`. This is the file from which the shell obtains your initial preference settings whenever you log in to the system.

Even if you have established a preference setting in a startup file or at the command prompt for the shell that is running, you can lose those settings if you invoke a subshell. To help establish preferences that persist not only across login shells, but also from shell to subshell, enter the preference in another startup file specified by the variable `ENV`, which is initially set to `.kshrc` in your home directory.

A/UX offers another way of retaining preference settings from shell to subshell without entering them in a startup file, but you can use it only for preference settings that are held in variables. After you store a value in the variable, you reset or set its export attribute. The value of an exported variable in a subshell is the value it had as of the time it was last exported.

However, this way of establishing a preference variable value does not persist across login sessions, and is subject to override by similar assignments placed in an ENV file.

To set the export attribute of a variable, enter either one of the following commands:

```
typeset -e variable
export variable
```

In a similar fashion, you can export aliases and functions to any ksh subshells, as the following commands illustrate:

```
typeset -ef function-name
alias -e alias-name
```

Exported variables can affect commands you execute as well as subshells you invoke. Commands are also able to respond to settings contained in exported variables (sometimes called ''environmental variables''). Typically, however, commands ignore all but a few of the values that you export into the environment, unless the command is ksh itself. Typically, you export those values that at the very least affect a ksh subshell.

TERM is an example of a variable that both commands and subshells regularly honor. The value of this variable also helps establish what type of terminal device you are using. The value of this variable helps A/UX programs look up the correct control sequences to use with particular terminals for various display functions, such as advancing the cursor location (see termcap(4) and terminfo(4)). Initially, TERM is set to a value identifying the console terminal as a Macintosh computer. However, during a CommandShell session in which you communicate with a host computer over a network, chances are slight that the host will understand this local setting for TERM, so another value should be used for the remote shell, such as vt100.

You can temporarily export a preference variable value for the duration of one command. For instance, to ignore the currently exported value of TERM and to use vt100 instead, but only for the vi command, enter

```
TERM=vt100 vi
```

You can use the set command with the -k option to affect the way shell variable assignments are treated when interspersed with other command elements. (Also see the discussion of -k in ''Arguments,'' earlier in this manual page). For example, consider the variable assignments inside the following command block:

```
echo var=b c ; set -k  ; echo var=b c
```

Because the first `echo` command is interpreted before the `-k` option takes effect, it generates this output:

```
var=b c
```

Because the second `echo` command is interpreted after the `-k` option takes effect, it generates this output:

```
c
```

Use of this feature is strongly discouraged. This option may not be supported in future releases.

You can use the discrete command `printenv`, or the built-in command `typeset` with the `-x`, option to find out the names and values for all exported variables.

When invoked, `ksh` gets its environment variable settings either from the parent shell or from one of the log-in programs. It then passes the settings to any commands or subshells you invoke, unless you manually removed them from the environment first (by removing the export attribute).

Here are some other variables with which you should be familiar:

`CDPATH`
: Contains a list of search directories that are honored by the `cd` command. See the description of the `cd` command in ''Built-in Commands,'' near the end of the ''Description'' section.

`CMDSHELLPREFS`
: Contains the name of a file in your home directory where CommandShell stores and reads your preferences. See `CommandShell`(1) for more information.

`EDITOR`
`HISTFILE`
`HISTSIZE`
: Contain values that select the style of editor for command lines, select the file where previously entered commands are stored, and set the number of command lines subject to storage and recall, as described in the next section, ''Command Reentry.''

`FINDER_EDITOR`
: Contains the pathname for the editing application to be launched when you open a text file by way of the A/UX Finder.

`PATH`
: Contains a list of command search directories. (See ''Command Execution'' later in the ''Description'' section.)

PS1
Contains the string used as your primary command prompt.

PS2
Contains the string used as your secondary command prompt for those occasions when you must enter a block of commands that spans more than one line. (See ''Using Repetition and Branching Constructs,'' later in the ''Description'' section, for related information.)

MAIL
MAILCHECK
MAILPATH
Contain values that help enable and customize electronic mail. See ''Other Built-in Variables,'' near the end of the ''Description'' section, for more information.

TBMEMORY
TBPATCHES
TBRAM
TBSYSTEM
TBTRAP
TBWARN
Contain values that help configure system parameters that apply to the virtual Macintosh environment. See startmac(1) for information regarding these variables.

**Command Reentry**

The text corresponding to the most recent block of commands is saved in a history file. The value of the variable HISTSIZE determines how many lines of commands are saved; it is initially set to 128. The value of the variable HISTFILE selects the file where the saved commands are stored; it is initially set to $HOME/.sh_history.

Subshells also have access to commands that were previously given from the parent shell, provided that you do not change the value of HISTFILE for the subshell.

You can use the built-in command fc to select a previous line to edit and reuse. To display a list of recently performed commands, enter:

```
fc -l
```

This command produces a list of recently used commands, such as this one:

```
416     date
417     whoami
418     cd /tmp
419     ls -tC
420     cat lastfsck
```

```
421     cd
422     ls -tC
423     more today
424     rm today
425     fc -l
```

To edit and reuse one of the commands from the history list, specify the line number as an argument to the `fc` command:

`fc` *line-number*

If you supply a string rather than a line number, the most recent command whose starting letters match the letters in the string is recalled for editing and reuse. You can also specify a range of old commands to be recalled for editing. Refer to ''Built-in Commands,'' later in the ''Description'' section, for a more complete description of the command options and arguments for `fc`.

The edited command is printed and reexecuted when you leave the editor.

The editor used is that specified by the value of `FCEDIT`, which is initially set to `/bin/ed`. You can also set the value of this variable to `vi` for full-screen editing, or you can set it to `TextEditor` for a mouse-and-menus style of editing.

The `ksh` shell has its own built-in command-editing support as well. See ''Command-Line Editing Options,'' later in the ''Description'' section, and the sections that follow it.

**Frequently Used Built-in Commands**

This section provides a brief list showing some of the commands that are more fully described in ''Built-in Commands'' near the end of the ''Description'' section. You may want to familiarize yourself with the commands listed here sooner than any others.

`alias`
: Creates pseudo commands that you can use as a shortcut for typing in much longer command lines. Also see ''Using `ksh`,'' earlier, and ''Aliasing,'' later in the ''Description'' section.

`bg`
`fg`
`jobs`
`kill`
: Control any running and suspended background jobs you have started. Also see ''Controlling Jobs Not in the Foreground,'' earlier in the ''Description'' section.

`cd` Sets the current working directory to the directory specified as an argument.

`exit`
Exits `ksh`.

`pwd`
Displays the current working directory.

`ulimit`
Establishes the upper size limit for a file as one of the many limits that can be set.

`umask`
Establishes how file permissions are initially set for new files that you create.

**Command Execution**

The earlier sections in this manual page introduce the general functions of the shell and its commonly used features. The next series of topics provide a glimpse into the inner workings of the shell, including: processing that supports command execution, processing that can change the execution environment for a command based on metacharacters you placed in a command line, and processing that can change the value of various metacharacter-delimited command elements.

If you enter a command name that matches one of the built-in commands, the command is executed as part of the current shell process. As such, it is not subject to the job-control commands that would affect an independent process.

Next, the shell checks whether the command name matches one of the user-defined functions and evaluates the function's replacement, if necessary. A function is also executed as part of the current shell process, so it is also not subject to job control. For more detailed information about functions, refer to ''Functions'' near the end of the ''Description'' section. That section is generally useful for users who are also programmers.

Then the shell determines if any alias substitutions must be made. Whether it is the result of an alias substitution or not, the command name can refer to an A/UX command, a Macintosh application, or a command script. Before this determination can be made, `ksh` must locate the executable file associated with the named command. So `ksh` performs a search for the command file. It searches for a file of the same name as the command entered. It looks in each of the search directories specified in the `PATH` variable.

The `PATH` variable contains a list of directories where commands are customarily located. Directory names are separated by colons (`:`). The default search setting for recent versions of A/UX is as follows:

```
PATH=/bin:/usr/bin:/usr/ucb:/mac/bin:.
```

When you use this specification, the final directory searched is the current directory. The current directory is represented by a period. Because the period appears in the last colon-separated field, the current directory is the last directory to be searched. For the PATH variable value, the current directory can also be represented by two or more adjacent colons, or by a colon at the beginning or end of the path list.

The search process is not performed if the command name is specified with a leading slash (/) character. In such a case, ksh expects you to supply the absolute path that locates a file that can be executed, or a relative path that locates the executable file based upon the current working directory.

Once the executable file is found, the format of the file helps distinguish whether it should be run as a shell script, a command, or a Macintosh application. The latter option allows you to launch Macintosh applications by name, provided that they reside on an A/UX file system in a directory that is listed in the definition of the PATH variable. Note, however, that if the name contains a space, you need to enclose the command name in quotation marks. (See ''Escape Characters,'' later in the ''Description'' section for related information.) To make the application easier to launch from a command line, you may want to rename the file so that it doesn't contain spaces:

```
mv "MacDraw II"  macdraw
```

Launching Macintosh applications by name is about the same as using the launch command (described in launch(1)). This method supports the same -p (print) option described for launch:

*mac-application* [-p] [*app-document*]...

For a command to run successfully, ksh must be able to find an associated executable file. Successful execution also depends upon the execute permission for that file (which is interpreted according to the login account you used to log in). The often-misleading File not found error message is displayed if the file permissions do not permit you to run the command.

If the result of the command search yields a file that is a command-containing shell script, a subshell is spawned to interpret the script as described in ''Command Scripts,'' later in the ''Description'' section.

The processing steps described in this section represent the minimum processing that ksh performs to run a simple command such as:

```
$ date
Fri Jun 14 13:31:22 PDT 1991
$ ■
```

However, much more shell processing of command lines can optionally

take place. Before you can master the command-line interface, you need to better understand how that optional processing is performed by the shell. To give you a more detailed understanding of the optional `ksh` processing, you should read the series of the sections that follow this one. Each subsection elaborates on one type of optional shell processing:

- Subcommand Execution
- Aliasing
- Tilde Substitution
- Command Substitution
- Variable Substitution
- Filename Generation
- Argument Parsing
- Input/Output Redirection
- Escape Characters
- Extra Initial Processing for a Login Shell
- Extra Initial Processing for Subshells

**Subcommand Execution**

Parenthesized commands are executed by a subshell. Unlike a subshell that supports the running of command scripts, this subshell has access to nonexported variable values from the parent shell as well as exported ones.

You can combine commands to be executed by a subshell with ordinary commands into a single processing request, in formats such as these:

[*command command-separator*]... (*subcommand*)

(*subcommand*) [ *command-separator command*]...

Because the subshell can be set to a different working directory, you can use it to help initiate commands that operate in two different directories, but are still part of a single command-line request. One possibility is shown in the following example, which copies a directory/file hierarchy from one location to another:

```
cd fromdir; tar cf - . | (cd todir; tar xf -)
```

**Aliasing**

As described earlier, `ksh` performs a substitution when it encounters an alias name in the command-name portion of a command line.

When declaring aliases with the `alias` command, you can use any nonspecial printable character as the first character of the alias name. The remaining characters must be the same as those for a valid identifier. (See ''Lexical Rules for Identifiers'' later in the ''Description'' section.) For syntactic information regarding the use of the `alias` command, see ''Built-in Commands'' later in the ''Description'' section.

The replacement string for an alias is a command line. Such a string can contain one or more references to commands or executable shell scripts. If it includes multiple commands, command separators must delimit them as described in ''Format of Command Lines,'' earlier in the ''Description'' section.

Generally, the command-name portion of the replacement value for an alias is not tested for additional aliases. However, if the last-entered character of the replacement value is a space, `ksh` makes any appropriate alias substitution for the first command name in the replacement value.

You can use aliases to redefine the built-in commands, but you cannot use them to redefine the keywords described later in ''Using Repetition and Branching Constructs'' within the ''Description'' section.

You can create, list, and export aliases with the `alias` command. You can remove aliases with the `unalias` command. Exported aliases remain in effect for subshells but do not persist across login sessions unless you enter them in the file `.profile`. (See ''Establishing Preference Settings'' within the ''Description'' section.)

Aliases are frequently used as a shorthand for longer command pathnames. The tracking feature for aliases enhances this use. It helps avoid the processing overhead that is otherwise required to locate the associated command each time an alias is used. The search that is normally moderated by the list of directories stored in the `PATH` variable is not always necessary because the shell can remember the full pathname to the aliased command after its first use. When such a manner of operation is enabled for an alias, the alias becomes a tracked alias. All tracked aliases become subject to reresolution each time the `PATH` variable is reset. When you do so, a search will take place to determine the correct location of an aliased command when the (tracked) alias is next used. Once the location has been reestablished according to the new `PATH` setting, the tracked alias is once again able to inhibit further command searches.

To treat all the aliases you subsequently define as tracked aliases, use `ksh` with the `-h` option as described in the ''Arguments'' section, earlier in this manual page. For an interactive shell that is already running, you can use the `set` command with the `trackall` argument to establish the same `ksh` preference. (See ''Establishing Preference Settings'' earlier in the

"Description" section.)

You can set the export attribute for an alias to indicate that it should also be passed to any ksh subshells you invoke. To set this attribute, use the alias command with the -x option. For more information about exported values and the export attribute, see "Establishing Preference Settings," earlier. The following "exported aliases" are set, but you can unset or redefine them:

```
autoload='typeset -fu'
false='let 0'
functions='typeset -f'
hash='alias -t'
history='fc -l'
integer='typeset -i'
nohup='nohup '
r='fc -e -'
true=':'
type='whence -v'
```

### Tilde Substitution

After performing alias substitution, ksh processes the command line for a type of substitution known as "tilde substitution," so-named because the metacharacter that triggers it is the tilde (~). When used as a discrete command argument, or when placed at the beginning of a pathname, the tilde is replaced with the full pathname to your home directory (the directory that becomes your default working directory when you log in). If the tilde precedes an argument that is a login name from /etc/passwd, then ksh substitutes the home directory of the login name specified. If no match is found, ksh leaves the tilde and any attached text unsubstituted, and treats them literally as command arguments.

You use the tilde primarily to avoid some typing when you want to specify files in your home directory but your current working directory is set elsewhere. Using the tilde can also reduce typing when you are specifying the path to an executable command file located in your home directory.

The syntax for a command request that makes use of tilde substitution for the command name is as follows:

~[*login-name*][/*dir*].../*command arg*...

The syntax for a command request that makes use of tilde substitution for a command argument is as follows:

*command* ~[*login-name*][/*dir*]...[/*file*]

A tilde followed by a + or - is replaced by the value of the current working directory (PWD) or the old working directory (OLDPWD), respectively.

In addition, tilde substitutions are performed if the tilde is placed at the beginning of the assignment value for a variable:

*variable*=~*value*

The tilde can also be placed elsewhere in the assignment value for a variable, as long as it is after a colon (:). For example, tilde substitutions may permit you to specify search directories for the PATH variable more succinctly, as in this example:

```
$ PATH=/bin:/usr/bin:~:~/Tools:
$ echo $PATH
/bin:/usr/bin:/disk0/mikee:/disk0/mikee/Tools
$ ■
```

**Command Substitution**

After performing any tilde substitutions, ksh processes the command line for metacharacters that request another type of substitution. ''Command substitution'' permits you to enclose command lines inside other command lines. The enclosed commands are replaced with the output text they produce when they are run. Of course, any enclosed commands are run first.

You delimit an enclosed command by placing it inside parentheses, and by placing a dollar sign in front of the open parenthesis:

$(*enclosed-command-line*)

If it occupies the position normally occupied by a command, the enclosed command must output a legal command. If it occupies the position normally occupied by a command argument, the enclosed command must output a legal argument for the enclosing command.

You can use command substitution to avoid typing a long list of filenames, as in this example:

```
tbl $(grep '^\.TS' * |cut -f1 -d: |sort -u) |nroff
```

In this example, the enclosed command is as follows:

```
grep '^\.TS' * |cut -f1 -d: |sort -u
```

This enclosed command generates the arguments for the tbl (table-preprocessing) command. In this case it is a list of files in the current directory that contain at least one occurrence of the table-start instruction, .TS, located at the beginning of a line. (The function of the grep command is to find occurrences of strings in files and report them.) The cut command strips all but the first colon-delimited field from its input

data. Because `grep` reports the names of files, a colon, and the line that contains the search string, `cut` outputs only filenames in this example. Those filenames are sorted into a list without any duplicates by the next pipelined command, `sort -u`. So when the enclosed commands are done executing, they produce as the arguments for `tbl` a list of files containing `tbl` instructions. The output of `tbl` is intended to be processed by a document-formatting program, such as `nroff`, so the enclosing command pipes `tbl` output as input to the `nroff` command.

In addition to the usual method of requesting command substitution just described, you can enclose a command substitution in grave accents:

*`command`*

When you use this (archaic) delimiter, the command between the grave accents goes through two rounds of processing for quotation mark metacharacters before the command is executed. (See ''Escape Characters,'' later.)

Instead of specifying

```
$(cat file )
```

you can use the following request, which is faster:

$(<*file*)

Most built-in commands that are not requested along with input/output redirection are executed in the same process as `ksh` when they are used as a command inside another command.

**Variable Substitution**

After performing command substitution, `ksh` processes the command line for variable substitution (also known as ''parameter substitution'').

The dollar sign (`$`) metacharacter that introduces a variable name can be considered the trigger for variable substitution. The dollar sign and the variable name that follows it are replaced with the value of the variable. Here is the format:

$*variable*

Usually *variable* is composed in accordance with the rules for identifiers. This is the case in particular for all user-defined variables. (See ''Other Built-in Variables'' later in this section, for variables that are preestablished.)

Another class of variables can be referenced in this format:

$*digit*
${*digit...*}

These variables are called ''positional parameters.'' Their primary use is

to allow access to the command-line parameters from inside a user-programmed shell script. For more information about scripts, see ''Command Scripts,'' later in the ''Description'' section.

For more information about variables and arrays, see ''Variables and Arrays,'' near the end of the ''Description'' section.

**Filename Generation**

After performing variable substitution, `ksh` processes the command line for metacharacters that request filename generation. Filename generation involves replacing a shorthand reference to a file with a more complete pathname or filename. Such a shorthand reference involves certain metacharacters, also referred to as ''wildcards.'' Often more than one argument is generated in place of a shorthand argument. This happens when several filenames or pathnames satisfy the wildcard criteria.

If you use one or more wildcard metacharacters, `ksh` regards the word in which they appear as a shorthand notation to be expanded, as long as the word is not enclosed in single (') or double (") quotation marks.

One of the metacharacters, or wildcards, that trigger this processing is the asterisk (*). Because variable substitution is performed before filename generation, the wildcard can be part of the text stored in a variable, as the following processing request shows:

```
$ files=/.*rc
$ echo $files
/.cshrc /.kshrc
$ ■
```

The same results are evident in the following commands, in which variable substitution plays no role. Also note that filename generation is suppressed by the double quotation marks.

```
$ echo /.*rc
/.cshrc /.kshrc
$ echo "/.*rc"
/.*rc
$ ■
```

The `ksh` program sorts generated arguments alphabetically when multiple filenames are generated because of a request for filename generation. The output produced by `echo` in the following processing request comprises any files in the current directory that end with the suffix `.out`. Note that the output is sorted alphabetically.

```
$ echo *.out
temp.out work.out
$ ■
```

If ksh cannot generate any filenames based upon the wildcards in the argument you specify, then the argument is treated as a literal argument. Thus, if the current directory does not contain any files with a .out suffix, filename generation for the previous example would fail, producing this output:

```
$ cd
$ echo *.out
*.out
$
```

The term ''pattern'' is used to refer to the wildcard along with the word in which it appears. So *.out can be thought of as a single search pattern.

When used as the first character of a search pattern, such as *.out, the asterisk produces file matches for any filenames containing any number of characters of any value, that end with the characters .out. When used as the last character of a search pattern, such as memo*, the asterisk produces file matches for any filenames containing any number of characters of any value, that start with the letters memo.

Wildcards other than the asterisk are more restrictive in terms of the matches they can produce. Particularly, they match only one character within a filename. You must use multiple occurrences of these other wildcards in the pattern in order to match a filename of more than one character. The question mark matches any one character of any value. Suppose you enter:

```
ls ???
```

The output produced comprises all the filenames that contain exactly three characters. When question marks are used along with strings, as in memo?.out, the matched filenames must be the same length as the pattern.

To form an even more restrictive wildcard, a wildcard that matches only certain characters in certain positions, a single character is not sufficient. For this kind of wildcard specification, a bracketed character list is used. Suppose you enter

```
ls memo[123]
```

The output produced comprises all the filenames that begin with memo and that end with either 1, 2, or 3.

The following list describes the criteria established by each of the wildcard notations. Note that in this list, the brackets shown are supposed to be typed. (When the brackets appear in the normal text font, as is usually the case, they surround optional items.)

`*` Matches zero or more characters of any value.

`?` Matches one occurrence of any character value at a particular location.

`[`*char...*`]`
`[!`*char...*`]`
Match one occurrence of any one of the enclosed characters in a particular character position (first form). If the first character in the list is an exclamation mark (second form), a match is produced whenever any one letter other than *char* occurs at a particular character position.

The placeholder *char* can also be specified as a three-character sequence representing a range (based upon ASCII ordering) of characters to be considered matches. This three-character sequence has the following format:

*startchar-endchar*

Thus `[A-Z]` matches one occurrence of any uppercase letter and `[A-Za-z]` matches any alphabetic character in either uppercase or lowercase format. To include a hyphen (`-`) as one of the list of possible character matches, make it the first or last character enclosed within the brackets.

When the filenames you want `ksh` to generate must be discriminated from very similar filenames, you may need to use more than one pattern to generate the desired files. Multiple patterns are separated by a vertical bar (`|`), as shown in the following summary of compound pattern syntaxes. For the patterns in this list, the brackets should not be typed because they surround optional items.

`?(`*pattern*`[|`*pattern*`]...)`
Matches zero or one occurrence of the patterns specified.

`*(`*pattern*`[|`*pattern*`]...)`
Matches zero or more occurrences of the patterns specified.

`+(`*pattern*`[|`*pattern*`]...)`
Matches one or more occurrences of any pattern specified.

`@(`*pattern*`[|`*pattern*`]...)`
Matches exactly one of the patterns specified.

`!(`*pattern*`[|`*pattern*`]...)`
Matches all strings, except those matched by the patterns specified.

**Argument Parsing**

After performing filename generation, `ksh` processes the argument portion of the command line to determine the number of arguments, the extent of each, and the level of escapement, if any, that you requested to obtain the literal interpretation of metacharacters.

The first two functions involve parsing of the command argument string into discrete arguments according to the presence of argument-separator metacharacters. The space or tab metacharacters in a command line are in turn affected by the third function, analyzing the level of escapement. Escapement is specified by paired quotation marks or the backslash character. These metacharacters can override the normal interpretation of spaces or tabs as argument separators, making them legitimate argument string characters (See ''Escape Characters'' later in the ''Description'' section.)

**Input/Output Redirection**

The `ksh` shell performs input and output redirections after it has substituted aliases, evaluated command substitutions, evaluated functions, generated filenames, and replaced variables in accordance with metacharacters and other preference settings that you supplied.

You specify a redirection of input, output, or both by using the appropriate metacharacter notation. Unlike processes that involve substitution, the process of redirection does not require that you make changes to the elements of the command line that get passed to the command. Rather, you accomplish redirection by altering the processing environment with which the command interacts.

Redirection requests are not propagated to the command they affect. The shell parses them and processes them by itself. For this reason, redirections can be intermixed with other command elements in any way:

*command arg...* [*redirect-in*] [*redirect-out*]

[*redirect-in*] [*redirect-out*] *command arg...*

You specify the value of *redirect-out* as the metacharacter > (greater-than sign) followed by a filename:

> *output-file*

You specify the value of *redirect-in* as the metacharacter < (less-than sign) followed by a filename:

< *input-file*

Another form of redirection involves multiple commands that share an information flow, bypassing the need for intermediate files. Consider these two commands:

```
who >/tmp/data
grep tty1  </tmp/data
```

The preceding sequence is equivalent to a pipe joining the two commands as a single processing request, as follows:

```
who | grep tty1
```

The output of the first command (`who`) is channeled directly to the input of the second command (`grep`). The pipe metacharacter (a vertical bar) indicates this channeling of data between commands. You can extend the pipeline to channel the output of a second command to the input of a third command, and so forth:

*command* | *command* [ | *command*]...

More discussion of input and output file redirection and command pipes is given earlier in ''Format of Command Lines.''

You can request that filename generation be performed for the filename portion of a redirection request by using the wildcards described earlier in ''Filename Generation.''

Filename generation produces a full reference to a filename or pathname, from a shorthand reference (or pattern) that you provide. Note, however, that the pattern is treated literally if honoring it would generate multiple filenames.

More often, a redirection metacharacter introduces the name of a file that you want to use as the source of input or the destination of output for a command. You can also specify another parameter in a redirection request, the *channel* parameter. A more technical term frequently used in place of *channel* is ''file descriptor.'' Replace *channel* with a digit between `0` and `9`, where applicable. The following list shows all the redirection formats available.

*<file*
*channel<file*
: Establish *file* as the source of standard input (file descriptor 0) for the command line in which it appears. If *channel* is specified (as in the second form shown), it establishes *file* as the source of input for the channel specified.

*>file*
*channel>file*
: Direct the standard output (file descriptor 1) to *file* for the command in

which it appears. If *file* does not exist, it is created. If *file* exists and the `noclobber` option is on, an error is generated; if *file* exists and the `noclobber` option is off, the file is truncated to zero length.

If *channel* is specified (as in the second form shown), it establishes *file* as the destination for output written to the channel specified.

>|*file*
Directs the standard output (file descriptor 1) to *file* for the command in which it appears. This format differs from > because it overrides the `noclobber` option, which might produce an error message because a file by the same name already exists.

>>*file*
Directs the standard output (file descriptor 1) to *file* for the command in which it appears. If *file* exists, output is appended to it; otherwise, *file* is created.

<>*file*
Establishes *file* as the source of standard input and the destination of standard output.

<<[-]*word*
Establishes a range of lines as the source of standard input for the command line in which it appears. The range of lines starts with the next line and continues until a line is reached that consists of *word* alone, or that contains an end-of-file character. For this type of redirection, no parameter substitution, command substitution, or filename generation is performed on *word*. The resulting range of lines that is used as input is sometimes called a ''here-document.'' If any character of *word* appears in quotation marks, the input lines are treated literally. Otherwise, `ksh` performs parameter and command substitution on the input lines and the backslash (`\`) must precede certain metacharacters to ensure their literal interpretation as input data. Particularly, the following metacharacters are subject to interpretation unless escaped: backslash (`\`), dollar sign (`$`), and grave accent (`` ` ``). If - is appended to <<, `ksh` strips all leading tabs from *word* and from the range of lines used as input.

<&*channel*
Establishes as the source for input the file (or device) currently associated with the file descriptor *channel* for the command in which it appears.

>&*channel*
Directs the standard output of a command to the file (or device) currently associated with the file descriptor *channel* for the command in which it appears.

*app-channel*<&*channel*
: Establishes the file (or device) referenced by *channel* as the source of input read by the application from *app-channel*.

*app-channel*>&*channel*
: Directs data sent by the application to *app-channel* to the file or device referenced by *channel*.

<&-
: Closes the current source of standard input for a command.

>&-
: Closes the current destination of standard output for a command.

[*channel*]<&p
: Connects the output from the coprocess to the standard input or, if *channel* is supplied, to the file or device referenced by *channel*.

[*channel*]>&p
: Connects the input of the coprocess to standard output or, if *channel* is supplied, to the file or device referenced by *channel*.

The order in which redirections are specified is significant. The shell evaluates redirections from left to right, making new associations based on the already established state. For example, the following line first associates file descriptor 1 with file `fname`; then it associates file descriptor 2 with the file currently associated with file descriptor 1 (that is, `fname`):

*command* `1>fname 2>&1`

If the order of redirections were reversed, as in

*command* `2>&1 1>fname`

then the output directed to file descriptor 2 would be sent to the display rather than to the file `fname`. This is why: File descriptor 2 is first set to correspond to the file or device associated with the standard output, which by default is the display since no redirection preceded it; then the standard output (file descriptor 1) is associated with file `fname`. The result is that the standard output alone is redirected to the file `fname`, which could have been much more simply requested as follows:

*command* `>fname`

If a command is followed by an ampersand (`&`) and job control is not active, the default standard input for the command is the empty file `/dev/null`. Otherwise, the default execution environment for each new command is the same as the shell from which it is invoked in terms of its file descriptors. You can override those defaults by using any of the input/output redirections described in this section.

### Escape Characters

To disable the special interpretation of metacharacters, such as spaces and dollar signs, you can insert other characters to cause the metacharacters to be interpreted literally. For example, any metacharacter preceded by a backslash is interpreted literally.

By preceding a newline character with a backslash, you can allow a long command to take up more than one line, as follows:

```
$ command argument1 argument2 argument3 \
argument4
$ ■
```

The conversion of a metacharacter to a nonspecial character that is treated literally is sometimes called ''escaping,'' and the characters that help accomplish this are called ''escape characters.'' A casual name for this (escapement) process is ''quoting.''

Normally, the shell begins parsing a new argument whenever it encounters a space character. To launch a program named SpiffWrite II, you could enter

```
"SpiffWrite II"
```

The double quotation marks suppress the interpretation of this line as a command named `SpiffWrite` with one command argument, `II`.

When you do not want to supply a value for the first argument to a command, but you do want to supply values for the second and third arguments, you need to pass an empty string in the place of the first argument. Normally, an empty string would not result from the substitution of a variable that held nothing (or the evaluation of a subcommand that produced nothing). To ensure that they can be interpreted as empty arguments when empty, you can enclose such references within double quotation marks:

```
paste "$file" /etc/passwd
```

The following command will not wait to read from the standard input if `filenames` is empty:

```
cat "`cat filenames`"
```

By enclosing a sequence of characters that includes metacharacters inside single quotation marks, you prevent the usual interpretation of the metacharacters (effectively suppressing filename generation, variable substitution, and subcommand execution). However, a single quotation mark cannot be treated literally within text delimited in this way, unless it is preceded by a backslash.

Note that the single quotation marks offer a different level of escapement than do the double quotation marks. For example, to treat two variable references and an intervening space as one argument, enclose them in double quotation marks:

```
command "$variable1 $variable2"
```

It is a mistake to enclose them in single quotation marks, unless you want to treat the entire string as one literal argument, as in this example:

```
$ echo '$variable1 $variable2'
'$variable1 $variable2'
$
```

If you enclose a sequence of characters that includes metacharacters in double quotation marks, the metacharacters triggering variable and subcommand substitution are not escaped, enabling those functions for the delimited text. In particular, the following metacharacters are still treated specially when enclosed in double quotation marks:

```
\ ' $
```

To cause these characters to be treated literally when enclosed in double quotation marks, precede each with a backslash.

You can avoid the special interpretation of keywords and aliases by escaping any character of the keyword or alias name. The recognition of function names and built-in command names (listed later in "Built-in Commands") cannot be altered with escape characters.

**Extra Initial Processing for a Login Shell**

A login shell is invoked in a characteristic way by `exec` as part of the login process. The login shell helps trigger processing that should take place only once, immediately after you log in. This specially-timed processing permits preference settings stored in the startup file `.profile` to take effect each time you log in.

The programs that manage the login process invoke `ksh` and pass its execution environment with the `$0` positional parameter set to `-ksh`. Upon inspecting `$0` and finding the leading hyphen, `ksh` reads commands from `/etc/profile` and then from either `.profile` in the current directory or `$HOME/.profile`, if either file exists.

You can customize preference settings by using these startup files and by using exported and unexported attributes for variables. (For more information, see "Establishing Preference Settings" earlier, and in the next section.)

**Extra Initial Processing for Subshells**

For any invocation of ksh, including one for establishing a login shell, the command lines from the file (if any) stored in the variable ENV are read and executed. For this reason, preference settings established in the file referenced by ENV are also established for all invocations of ksh subshells, whereas preferences established from .profile are passed to subshells only if the settings are exported and only if they are not overridden by settings in the ENV file.

(If ENV contains the appropriate metacharacters, ksh processes the replacement value for further substitutions. This process permits ENV to contain a reference to the built-in variable $HOME.)

By default, *home*/.kshrc is the value stored in ENV, where *home* is the home directory for your user account. If ENV is not set or is empty, no initial commands are executed.

**Command-Line Editing Options**

This is the first of several sections about the editing of command lines. The following sections also deal with these topics:

- The emacs Editing Mode
- The vi Editing Mode
- Commands for vi Input Mode
- Cursor Movement Commands for vi Edit Mode
- History Commands for vi Edit Mode
- Text Editing Commands for vi Edit Mode
- Other Commands for vi Edit Mode

If you have already learned how to use a UNIX®-style editor such as vi or ed, you can invoke those actual programs rather than using the imitations of them inside ksh. If you prefer to edit command lines by using a mouse-and-menus approach, you can even use TextEditor. To enter one of these editors, store the name of the desired editor in FCEDIT and then use the fc command as described earlier in "Command Reentry." The editing systems that are described in this section are more tightly integrated with ksh, so they can offer even speedier access to a previously entered command.

Because of their reference-style treatment here, you should already be comfortable with one or more of the editing programs as described in *A/UX Text Editing Tools* in order to be comfortable reading these sections. Be forewarned, however, that not all operations operate identically, particularly for the vi editing mode.

When the command-line editing feature of ksh is active, you can edit the current command line and scroll up and down to inspect, edit, and reuse any of your previously entered commands within the range of the history file. (See "Command Reentry" earlier, for more information about the history file.)

You choose one style of command-line editing by assigning the variable VISUAL or the variable EDITOR one of these values:

```
vi
emacs
gmacs
```

These built-in editors are modeled after the the stand-alone editors that have the same names. For more introductory information, see the commercial books that describe the emacs and gmacs editors, or see *A/UX Text Editing Tools* for an introduction to the vi editor.

Each of these built-in editors displays a recalled command line after the most recent command prompt. You can consider the area of the display affected as a one-line text window, the contents of which you can scroll to view other command lines in the command history file. You can set the width of the text window by using the variable COLUMNS. The text window width is set to 80 columns automatically whenever COLUMNS is unset.

If the command line that you enter or that you recall from the history file exceeds the length of the text window, a special character is displayed at its boundary. This character provides a reference point (mark) for the characters horizontally scrolled into view as you move the cursor toward the end of a long command line.

As the cursor reaches the text window boundaries, the text is horizontally scrolled so that the window continues to enclose the cursor. The mark changes, depending on where text has scrolled off the window. If text has scrolled off the end of the window, a > is displayed. If text has scrolled off the beginning of the window, a < is displayed. If text has scrolled off both the beginning and the ending of the window, a * is displayed.

**Command-Line Editing With vi**

When you use the vi style of editing, you flip between two modes of operation, one for entering text-editing commands and one for entering the text of a command line. The data entry mode is called "input mode." The command entry mode, called "edit mode," can be broken down further into modes such as character-insert and character-overwrite modes.

Initially, you are placed in input mode so that you can enter the text of a command line. To begin editing text that has been entered, you switch to the edit mode by pressing ESCAPE, move the cursor over the character position that requires a correction (using motion commands in edit mode), and enter an edit command that invokes input mode once again, and then insert or overwrite text.

When you are ready to run the command line, you press RETURN. You can do so from either input or edit mode.

In the command tables in subsequent sections, the notation for control characters is caret ( ^ ) followed by the character. For example, ^f is the notation for CONTROL-F. You enter this key combination by pressing the F key while holding down the CONTROL key. You do not have to press the SHIFT key.

**Commands for `vi` Input Mode**

By default the editor is in input mode which lets you enter a command line that you wish to run. Within this mode the following character-commands are accepted.

Erase
: Deletes previous character. (You define the erase character by using the `stty` command. This character is usually ^H or #.)

^D
: Terminates the shell.

\
: Causes the next erase, kill, or end-of-line character to be interpreted literally.

**Cursor Movement Commands for `vi` Edit Mode**

The following character sequences move the cursor in edit mode. For most of these commands, you can optionally provide a count parameter that proportionally increases the distance that the cursor travels.

[*count*]l
: Moves the cursor forward (right) one character.

[*count*]w
: Moves the cursor to the beginning of the next word.

[*count*]W
: Moves the cursor to the beginning of the next word that follows a space.

[*count*]e
: Moves the cursor to the end of the current word.

[*count*]E
: Moves the cursor to the end of the current space-delimited word.

[*count*]h
: Moves the cursor backward (left) one character.

[*count*]b
Moves the cursor to the previous word.
[*count*]B
Moves the cursor to the preceding space-separated word.
[*count*]f*c*
Finds the next occurrence of character *c* in the current line.
[*count*]F*c*
Finds the previous occurrence of character *c* in the current line.
[*count*]t*c*
Finds the next occurrence of *c* in the current line, then moves backward one character.
[*count*]T*c*
Finds the previous occurrence of *c* in the current line, then moves forward one character.
[*count*];
Repeats the last single-character find command (f, F, t, or T).
[*count*],
Reverses the last single-character find command.
0 Moves the cursor to the beginning of the line.
^ Moves the cursor to the first nonblank character in the line.
$ Moves the cursor to the end of the line.
% Moves the cursor to the balancing (, ), {, }, [, or ]. If the cursor is not on one of these characters, the remainder of the line is searched for the first occurrence of one of these characters.

**History Commands for vi Edit Mode**

The following character sequences display commands you entered previously as long as they fall within the range of your command history file:

[*count*]k
Displays the previous command each time k is entered. If a *count* parameter is supplied, the command that is *count* commands prior to the current one is displayed.
[*count*]-
Displays the previous command. Equivalent to k.
[*count*]j
Displays the next command each time j is entered. If a *count* parameter is supplied, the command that is *count* commands more recent than the current one is displayed (if such a command exists).
[*count*]+
Displays the next command. Equivalent to j.
[*lineno*]G
Displays the command numbered *lineno*. You can obtain the line

numbers associated with commands by entering `fc -l`. The default is the least recent history command.

`/`[*pattern*]
: Searches backward through the history file for a previous command containing *pattern*. You indicate the end of the pattern by pressing RETURN (or by generating a newline). If you do not specify a pattern, the most recently specified pattern will be used.

`?`[*pattern*]
: Searches forward through the history file for a previous command containing *pattern*, as a counterpart to `/`.

`n`
: Searches forward for the next match of the last pattern specified by a `/` or `?` command.

`N`
: Searches backward for the next match of the last pattern specified by a `/` or `?` command.

**Text Edit Commands for `vi` Edit Mode**

You can use these commands to modify the currently displayed line. To use these commands, you either switch from edit mode to input mode and type new characters into a line; or you remain in edit mode and directly change a specific amount of text. In the latter case you can continue supplying other edit-mode commands. In the former case, you cannot access edit-mode commands again until you leave input mode.

`a`
: Enters input mode. New text is entered after the current character.

`A`
: Enters input mode. New text is entered after the last character of the current line. Equivalent to the command `$a`.

[*count*]`c`*motion*

`c`[*count*]*motion*
: Enter input mode after marking for deletion the string starting with the current character and ending with the character that is *count* units away in the direction and units given by *motion*. If *motion* is `c`, the entire line is deleted and you enter input mode.

`C`
: Enters input mode after marking the current line for deletion.

`S`
: Enters input mode after deleting the current line; equivalent to the command `cc`.

`D`
: Enters input mode after deleting the string starting with the current character and continuing to the end of the line. Equivalent to the command `d$`.

[*count*]`d`*motion*

`d`[*count*]*motion*
: Delete from the current character to the character that is *count* units away in the direction and units given by *motion*. If *motion* is `d`, the entire line is deleted.

`i` Enters input mode. New text is inserted before the current character.

`I` Enters input mode. New text is inserted at the beginning of the current line. Equivalent to the command `^i`.

[*count*]`P`
Inserts the contents of the deletion buffer before the cursor.

[*count*]`p`
Inserts the contents of the deletion buffer after the cursor.

`R` Enters input mode. New text replaces existing text as the cursor advances over the old text.

`r`*c* Replaces the current character with *c*, while remaining in command mode.

[*count*]`x`
Deletes the current character.

[*count*]`X`
Deletes the preceding character.

[*count*]`.`
Repeats the previous text-modification command, replacing the previously supplied count with *count*.

`~` Converts the case of the current character and advances the cursor.

[*word-number*]`_`
Enters input mode after pasting a word from the most recently executed command into the current command line. The particular word inserted depends on the value of *word-number*. If this parameter is omitted, the last word of the previous command line is inserted. Otherwise, *word-number* selects the word to be inserted in an ordinal manner.

`*` Attempts filename generation based on the current word. Filenames that begin with the same letters as the current word are generated. If no match is found, a beep is generated. Otherwise, the word is replaced by the matched filename and you enter input mode.

`\` Attempts pathname completion. Replaces the current word with the longest common prefix of all filenames that begin with the same letters as the current word. If the match is unique and the match is a directory, a slash (/) is appended. If the match is unique and the match is a file, a space is appended.

### Other Commands for `vi` Edit Mode

[*count*]y*motion*

y[*count*]*motion*

Copy (''yank'') text from the current character to the character selected by *motion* and put it into the deletion buffer. The text and cursor are unchanged.

`Y` Copies (''yanks'') text from the current position to the end of the line and puts it in the deletion buffer. Equivalent to `y$`.

`u` Undoes the last text-modification command.

`U` Undoes all the text-modification commands performed on the line.

[*count*]`v`

Displays this command into the input buffer.

```
fc -e ${VISUAL:-${EDITOR:-vi}} count
```

If *count* is omitted, the current line is used.

`^L` Inserts a line feed and redisplays the current line if you are in edit mode. Otherwise, this command is treated literally.

`^J`

Executes the current line, regardless of mode.

`^M`

Executes the current line, regardless of mode.

`#` Inserts a number sign (#) before the line and then sends the line, converting it into a shell comment. Use this command to insert the current line into the history file without executing it.

`=`

Generates a lists of filenames that begin with the same letters as the current word.

`@`*letter* Searches for an alias by the name *_letter*. (Note that an underscore is prepended to the letter.) If an alias of this name is defined, its value is displayed.

### The `emacs` and `gmacs` Editing Mode

You enter the `emacs` (or `gmacs`) editing mode by using the `emacs` (or `gmacs`) setting for either the variable `EDITOR` or the variable `VISUAL`. The only difference between `emacs` and `gmacs` modes is the way they handle CONTROL-T: The `emacs` mode transposes the current character with the next character, whereas `gmacs` mode transposes the two previous characters.

To edit lines, move the cursor to the point needing correction and then insert or delete characters or words as needed. All edit commands operate from any place on the line (not just at the beginning). All of the editing commands are control characters or escape sequences.

In the command table that follows, the notation for control characters is a caret (^) followed by the character. For example, ^F is the notation for CONTROL-F. The SHIFT key is *not* pressed. (The notation ^? indicates DELETE.)

The notation for escape sequences is M- followed by a character. For example, you enter M-f (pronounced "meta f") by pressing ESCAPE (ASCII 033) followed by the F key. The character case is significant; M-F is not the same as M-f, so in this case you do not press SHIFT.

^F Moves the cursor forward (right) one character.

M-f
: Moves the cursor forward one word. (The editor considers a word to be a string of characters consisting only of letters, digits, and underscores.)

^B Moves the cursor backward (left) one character.

M-b
: Moves the cursor backward one word.

^A Moves the cursor to the beginning of the line.

^E Moves the cursor to the end of the line.

^]*char*
: Moves the cursor to the first occurrence of the letter *char* on the current line.

^X^X
: Interchanges the cursor and mark.

*erase*
: Deletes the previous character. (You define the erase character by using the stty command. This character is usually CONTROL-H or #.)

^D Deletes the current character.

M-d
: Deletes current word.

M-^H
: Deletes the previous word. (Pronounced "meta backspace".)

M-h
: Deletes the previous word.

M-^?
: Deletes the previous word. (Pronounced "meta delete".) If your interrupt character is ^? (DELETE, the default), this command will not work.

^T Transposes the current character with next character in emacs mode. Transposes the two previous characters in gmacs mode.

^C Capitalizes the current character.

M-c
Capitalizes the current word.

M-l
Changes the current word to lowercase.

^K Deletes all characters from the cursor to the end of the line. If given a parameter of 0, it deletes all characters from the beginning of the line to the cursor.

^W Deletes all characters from the cursor to the mark.

M-p
Pushes the region from the cursor to the mark on the stack.

*kill* Deletes the entire current line. (You define the kill character by using the `stty` command. This character is usually CONTROL-G or `@`.) If you use two kill characters in succession, all subsequent kill characters will cause the generation of a line feed (useful when you are using paper terminals).

^Y Pastes the last-yanked text to the line.

^L Inserts a line feed and prints the current line.

^@ Sets the mark.

M-*space*
Sets the mark. (Pronounced ''meta space''.)

^J
Executes the current line.

^M
Executes the current line.

*eof* Produces the end-of-file character, normally CONTROL-D. This command terminates the shell if the current command line is empty except for the end-of-file character.

^P Displays the command previous to the current one. Each time CONTROL-P is entered, the command prior to the one just fetched is displayed.

M-<
Displays the least recent (oldest) history line.

M->
Displays the most recent (youngest) history line.

^N Displays the next-most-recent command. As long as more recent ones exist, each time CONTROL-N is entered, the next command in a previous sequence of commands is displayed.

^R*string*
Searches backward for a previous command line containing *string*. If a parameter of 0 is given, the search is forward. The string is terminated by a return or newline character. If *string* is preceded by a caret (^), the matched line must begin with *string*. If *string* is omitted, the next command line containing the most recently specified string is accessed. In this case, a parameter of 0 reverses the direction of the

search.

^O Executes the current line and fetches the next line relative to the current line from the history file.

M-*digits*
Defines the numeric parameter; the digits are taken as a parameter to the next command. The commands that accept a parameter are ., ^F, ^B, *erase*, ^D, ^K, ^R, ^P, ^N, M-., M-_, M-b, M-c, M-d, M-f, M-h, and M-^H.

M-*letter*
Searches the alias list for an alias by the name *_letter*.(Note that an underscore prefix is added.) If an alias of this name is defined, its value is inserted on the input queue. The letter must not conflict with one of the metafunctions described in this list (so do not use the letters f, b, d, p, l, c, h).

M-.
Inserts the last word of the previous command into the current line. If preceded by a numeric parameter, the value of this parameter determines which word to insert instead of the last word.

M-_
Inserts the last word of the previous command into the current line. Same as M-..

M-*
Attempts filename generation based on the current word. Filenames that begin with the same letters as the current word are generated.

M-ESCAPE
Attempts filename completion. Replaces the current word with the longest common prefix of all filenames that begin with the same letters as the current word. If the match is unique, a slash (/) is appended if the file is a directory and a space is appended if the file is not a directory.

M-=
Lists all files that begin with the same letters as the current word.

^U Multiplies the parameter of next command by 4.

\ Causes the next character to be interpreted literally. This character allows editing characters and your erase, kill, and interrupt characters to be entered in a command line or a search string.

^V Displays the version number for the shell.

**Command Scripts**

Command scripts are files containing a number of command lines that are run as one batch. The same command-line user interface that you use for normal command programs is also used to invoke scripts. You can run a file containing commands by supplying the name of the file as the command in a command line.

As described earlier in the section ''Using ksh,'' a shell performs a search for the file corresponding to each command. The order of this search is dictated by the list of directories stored in the PATH variable.

If the file is a shell script containing legal command lines, a subshell is spawned to interpret it. In the subshell, only ''exported'' aliases, functions, and variables retain the values they had in the parent shell.

To make your script programs as easy to run as A/UX commands, you need to locate them in one of the directories specified by PATH and extend permission to a range of users (or just yourself) to execute the script file. See chmod(1) for information about permission attributes for a file.

As an alternative to treating the shell script just like an A/UX command program, you can execute it by submitting the file as the input for ksh in one of these formats:

ksh *shell-script-file*
cat *shell-script-file* | ksh

To execute a script this way, you need read permission, but not execute permission. Note that any setuid and setgid file attributes are ignored. (These attributes are described in chmod(1).)

When you specify a shell script as an argument to ksh, ksh performs a PATH-moderated search to discover where the named shell script resides. This search is exactly like the search for scripts executed like commands.

When executed like a command, a script file for which you have set the setuid permissions, setgid permissions, or both is executed in a special way. The shell executes an agent that sets up an altered execution environment for use when running the script. This special subshell obtains its startup settings from /etc/suid_profile.

Since ksh reads at least one line at a time, new aliases do not affect subsequent commands on the same line, but affect only subsequent lines. For example, if you have two or more simple or compound commands on a single line, such as this:

```
alias bo=didley; bo
```

ksh reads all of the commands on the line before executing them. Therefore, bo is not equivalent to didley because bo does not follow the alias command line.

To add flexibility to shell scripts, traditional programming constructs are included. For example, you can establish a block of commands that is run only when a conditional test evaluates to true (such as whether a file exists), or when the test evaluates to false. Certain keywords help you indicate the scope of conditional blocks of commands. The keywords that

specify conditional blocks of commands are `if`, `elif`, `else`, and `fi`.

Other keywords define blocks of commands that can be repeated a number of times. Each time a repeatable block of commands is executed, a test is performed to see if another iteration of the loop should be performed. For example, you could use the program segment shown here to process each file in a group of files in the same way:

```
for file in chap1 chap2 chap3
do
  pr $file | lpr
  touch -a $file
done
```

### Use of Comments

An important part of programming is documenting the components of the program. Comments help anyone using the shell script understand the job of each program component.

The # metacharacter tells `ksh` to ignore the remaining text on a line, treating it as a comment rather than a command line.

Use comments throughout a lengthy shell script. Often they appear before a block of commands that has a common processing focus:

```
#  Was a valid filename was offered?
#  If not, report the error and exit.
if [[ ! -f $file ]]
then
   ...
fi
```

### Specialized Command-Line Processing for Scripts

When a script is invoked from the command line, the subshell that runs on its behalf is initialized to reflect the current state of the parent shell and the state of the command line. For example, any parent shell variables with the export attribute set are also initialized in the subshell. Furthermore, the command-line arguments given after the script filename are the source of assignment values for the positional variables of the subshell. The following example illustrates how command elements are mapped to positional variables:

```
script-name earth wind fire water
    ↑        ↑    ↑    ↑    ↑
   $0       $1   $2   $3   $4
```

This processing can be affected by preference settings for the argument-separator character, by command options for `ksh` (see the description of the `-f` and `-k` options in ''Arguments,'' earlier), by filename generation (see ''Filename Generation,'' earlier), and by escape characters (see ''Argument Parsing'' and ''Escape Characters,'' earlier).

You can change the argument-separator character to something other than the default space and tab. To change the argument separator character, assign the desired separator to the `IFS` variable for the interactive shell you use to invoke the script.

To refer to the script arguments beyond the ninth one, use this format for referring to the variable:

`${`*digit...*`}`

Here is a simple script that displays the first two command-line arguments given to it:

```
echo $2  $1
```

If you placed the preceding command in a file called `swapargs` and established execute permission for the file, you could duplicate the results of the following command line:

```
$ swapargs one two
two one
$
```

The special built-in variables `$*` and `$@`, when not enclosed by quotation characters, cause identical substitutions to occur. They are replaced with the text of each of the positional variables, with space characters (or the character stored in the `IFS` variable) inserted between each variable.

When the same references are enclosed within quotation characters, the replacement values are subject to different interpretation if `ksh` parses arguments on them again. (See ''Argument Parsing,'' earlier.) If you want to obtain the same positional variable assignments after a second argument-parsing process, use `"$@"` to refer to the positional parameters of the subshell. This variable ensures that ''escaped'' spaces that were originally part of discrete arguments remain a part of those arguments, and thereby results in a faithful restoration of the number and composition of positional variables in subshells to a subshell.

Besides allowing the shell to assign positional variables (described in ''Argument Parsing,'' earlier), you can set the positional variables yourself by using the built-in `set` command. (See ''Built-in Commands'' later in this section.)

### Using Repetition and Branching Constructs

While control constructions such as conditional blocks and loops are mostly used in scripts, you can also use them in command lines. If you structure them normally, they will span more than one line. To provide you with an indication that the command entry is not complete, `ksh` switches to a different prompt temporarily. You can change this secondary prompt by using the `PS2` variable.

The usual interpretation of a newline signal is temporarily abandoned when you have entered one of the control keywords but not an associated terminating keyword. This signal permits you to enter any number of lines before you terminate the control construction by using the appropriate keyword (`done`, `esac`, or `fi` for loops, case structures, and conditional command blocks, respectively). For the line in which the terminating keyword is entered, it indicates the end-of-line character received is interpreted normally: the end of the command-block entry and the beginning of its execution.

The following example is a repeat of an earlier one. Shown here are the prompts that would appear before each of the commands or keywords during interactive operation:

```
$ for file in chap1 chap2 chap3
> do
>   pr $file | lpr
>   touch -a $file
> done
$ ■
```

Once all of the command blocks have been closed (all nested loop or branch structures ended with the appropriate terminating keyword), `ksh` checks the command blocks for errors. If there are none, the shell runs the individual commands in each of the blocks one or more times, depending on the constructions used and the conditional expressions that govern their execution.

Usually blocks of commands are entered into a script, which is a file that contains command lines. (See "Command Scripts," earlier.) This permits them to be easily recalled without a lot of typing.

A control structure must contain a conditional expression that yields a Boolean value. Conditional expressions are evaluated upon each iteration of a loop so that the shell can determine when to exit the loop (for instance, after a certain number of iterations). Likewise, conditional expressions control whether a conditional command block executes or is skipped (and optionally whether an alternative, `else`-introduced command block executes).

How are these Boolean values obtained? Each A/UX command returns an exit status value that is habitually not displayed as part of the output text for a command. Nevertheless, this exit value is communicated to the shell that dispatched the command and is accessed through the `$?` variable. If multiple commands are dispatched together as part of a pipe or as part of a sequence of commands separated with ampersand or semicolon metacharacters, the exit status remembered is the exit status of the last command in the series.

**Control Structure Syntax**

You can change the normally sequential flow of shell execution from one command line to the next by using the keywords described in this section.

You use the placeholder *list* to represent one or more commands that are executed under the control of a loop or conditional construction. You can include an arbitrary number of newlines instead of semicolons to delimit commands in *list*.

The notation

`for` *identifier* [`in` *arg*...] `;do` *list* `;done`

is equivalent to

```
for identifier [in arg...]
do
        list
done
```

Here, the argument *identifier* is used to represent a name chosen by you and appearing in the some variations of the `for` loop. (See also ''Lexical Rules for Identifiers,'' later.)

Because the argument *list* is used to represent any sequence of commands, it can also appear at locations where a Boolean result is required. In such cases, the real use of the commands in the list is to generate a Boolean result that in turn dictates whether another block of commands should be run or should be skipped. Most of the time, the built-in command `test` (or an equivalent form of this command) is run as described later in the section ''Built-in Commands.'' The true or false result generated is based upon the exit value of the last command in the list. A true result is generated if the last command has a 0 exit value. A false result is generated otherwise. Only the exit values of command lists that are located immediately after one of the keywords `if`, `elif`, and `while` is subject to this manner of interpretation by `ksh`. Note that the command blocks whose execution is affected are also notated with *list* in the syntax descriptions that follow.

`for` *identifier* [`in` *arg* [*white-space arg*]...]; `do` *list* ;`done`
The commands between the `do` and `done` (represented notationally as *list*) keywords are repeated as many times as there are white-space separated arguments between `in` and `do`. Each time through that set of commands, the variable *identifier* is assigned the value of one of the arguments represented here as *arg* values.

The commands in *list* can contain references to `$`*identifier*, which will contain a value equal to the *n*th *arg* value upon the *n*th iteration of the loop. If no arguments are specified between `in` and `do`, the `for` command executes the commands in *list* once for each positional parameter that is set. (See ''Specialized Command-Line Processing for Scripts,'' earlier.) Iteration ends when the positional parameters have been exhausted.

`select` *identifier* [`in` *arg* ... ] ;`do` *list* ;`done`
Writes to standard error channel (file descriptor 2) each *arg* value given that is preceded by a number. If

`in` *arg*...

is omitted, then the positional parameters are used instead. (See ''Specialized Command-Line Processing for Scripts,'' earlier.) The `PS3` prompt is printed, and a line is read from the standard input. If this line consists of a number referencing one of the enumerated *arg* values, then the value of the variable *identifier* is set to the corresponding *arg* value. If the input line received is empty, the selection list is printed again. If the input value received is out of bounds, the variable *identifier* is set to an empty string. Whether in bounds or not, the input line received is saved in the variable `REPLY`. The commands in *list* are executed for each selection until a break character or an end-of-file character is received.

`case` *word* `in` [*pattern* [ | *pattern*] ... ) *list* ;; ] ... `esac`
Executes the command list associated with the first pattern that matches *word*. The form of *pattern* is the same as that used for filename generation. (See ''Filename Generation,'' earlier.)

`if` *list* ;`then` *list* [`elif` *list* ;`then` *list*] ... [;`else` *list*] ;`fi`
Executes the command list following `if` and, if a 0 exit status is returned, the command list following the first `then` command. Otherwise, a command list following any `elif` is executed and, if a 0 exit status is returned, executes the command list following the associated `then` is executed. Furthermore, when the first command list following an `if` command returns a nonzero exit status, any command list following an `else` command is executed. If the test conditions do not permit any command lists that follow either an

`else` or a `then` command to be executed, then the `if` command returns a 0 exit status.

`while` *list* `;do` *list* `;done`
`until` *list* `;do` *list* `;done`
Repeatedly execute both command lists until the first command list returns a value that terminates the loop. For a `while` loop, the iteration ceases when a nonzero exit value is returned. For an `until` loop, the iteration ceases when a 0 exit value is returned. If no commands in the list following `do` are executed, then the `while` and `until` commands return a 0 exit status; `until` can be used in place of `while` as a negation of the loop-termination test.

`(`*list*`)`
Executes *list* in a separate shell environment. Note that if two adjacent open parentheses are needed for nesting shells, you must insert a space between them to avoid producing a request for arithmetic evaluation as described in the next section.

`{` *list;*`}`
Executes the command block denoted by *list*. Note that `{` represents *keyword* and is not recognized unless it occurs at the beginning of a line or after a semicolon.

`[[`*expression*`]]`
Evaluates *expression* and returns a 0 exit status when *expression* is a test that evaluates as true. This command replaces the `test` command of previous shell versions. (See ''Conditional Expressions,'' later.)

`function` *identifier* `{` *list;* `}`
*identifier*`()` `{` *list;* `}`
Define a function that can then be referenced by the identifier supplied. Upon reference, the command list provided between `{` and `}` is executed. (See ''Functions,'' later.)

`time` *pipeline*
The commands in *pipeline* are executed, and the elapsed time, as well as the user and system time, are printed on standard error.

The following keywords are recognized only when they occur at the beginning of a line or after a semicolon:

```
if then else elif fi case esac for do
while until done { } function select time
```

**Arithmetic Evaluation**

Use the following format to request arithmetic operations:

`((`*variable=arithmetic-expression*`))`

This format is equivalent to

`let "`*variable=arithmetic-expression*`"`

You can use arithmetic expressions to keep track of the number of loop iterations, as in this example:

```
max=7
count=0
while [ $count -lt $max ]
do
    ((count=$count + 1))
    ...
done
```

You can use arithmetic expressions in two other ways. They can appear in an indexing expression that references an element in an array, such as

```
student[$count+1]
```

You can also use them in a `let` command, which you can use to initialize variables:

```
let count=$tests*students
```

Evaluations are performed by means of *long* arithmetic. Constants are expressed in one of the following formats:

*decimal-digit...*
[*base*#]*digits-n-letters...*

In the first format, a decimal value (base 10) is assumed. In the second format, the base is given by *base*. It must be a decimal number between 2 and 36, and *digits-n-letters* specifies a number in that base.

The rules of syntax, precedence, and associativity that apply to expressions in the C language apply here as well. All of the integral operators other than `++`, `--`, `?:`, and `,` are supported.

You can reference variables by name within an arithmetic expression without using the parameter substitution syntax. When you reference a variable, its value is evaluated as an arithmetic expression.

Because many of the arithmetic operators also function as metacharacters, you must suppress their metacharacter interpretation, often by placing them in question marks. An alternative form of the `let` command is provided to eliminate their interpretation as metacharacters. This alternative format

uses opening and closing double parentheses to replace `let`, as described at the beginning of this section.

**Conditional Expressions**

A conditional expression controls the flow of execution through and around blocks of commands bounded by keywords such as those described earlier in "Using Repetition and Branching Constructs." Many times, a test command will be included that generates a Boolean result. You can use the following test-command format. (In this syntax format, brackets are enterable characters — they do not enclose optional elements.)

```
if [[ test-argument... ]]
```

Another (archaic) way to introduce these options and arguments is after the keyword `test`:

```
if test test-argument...
```

You can combine several Boolean conditions by replacing *test-argument* with a number of binary and unary tests joined together by using logical operations such as logical AND (`&&`) and logical OR (`||`). You can invert the Boolean value produced by a test argument by preceding it with a negation operator (`!`).

Argument parsing takes place, consuming the white-space that separates arguments, except when you use escape characters to preserve the literal interpretation of spaces. Filename generation is not performed even when test arguments are represented by *file* as shown in the following list.

The following primitives are available for use as test arguments:

`-a` *file*
: True if *file* exists.

`-b` *file*
: True if *file* exists and is a block special file.

`-c` *file*
: True if *file* exists and is a character special file.

`-d` *file*
: True if *file* exists and is a directory.

`-f` *file*
: True if *file* exists and is an ordinary file.

`-g` *file*
: True if *file* exists and has its setgid bit set.

`-k` *file*
: True if *file* exists and has its sticky bit set.

`-n` *string*
True if the length of *string* is nonzero.

`-o` *option*
True if *option* is on.

`-p` *file*
True if *file* exists and is a First-In-First-Out (FIFO) special file or a pipe.

`-r` *file*
True if *file* exists and is readable by the current process.

`-s` *file*
True if *file* exists and its size is greater than 0.

`-t` *n*
True if file descriptor number *n* is open and associated with a terminal device.

`-u` *file*
True if *file* exists and has its setuid bit set.

`-w` *file*
True if *file* exists and is writable by the current process.

`-x` *file*
True if *file* exists and is executable by the current process. If *file* exists and is a directory, the current process has permission to search in the directory.

`-z` *string*
True if *string* is an empty string.

`-L` *file*
True if *file* exists and is a symbolic link.

`-O` *file*
True if *file* exists and is owned by the effective user ID of this process.

`-G` *file*
True if *file* exists and its group matches the effective group ID of this process.

`-S` *file*
True if *file* exists and is a socket.

*file1* `-nt` *file2*
True if *file1* exists and is newer than *file2*.

*file1* `-ot` *file2*
True if *file1* exists and is older than *file2*.

*file1* `-ef` *file2*
True if *file1* and *file2* exist and refer to the same file.

*string1* `=` *string2*
True if the strings match one another.

*string1* `!=` *string2*
True if the strings do not match one another.

*string1* `<` *string2*
True if *string1* comes before *string2* based on the ASCII value of their characters.

*string1* `>` *string2*
True if *string1* comes after *string2* based on the ASCII value of their characters.

*exp1* `-eq` *exp2*
True if *exp1* is equal to *exp2*.

*exp1* `-ne` *exp2*
True if *exp1* is not equal to *exp2*.

*exp1* `-lt` *exp2*
True if *exp1* is less than *exp2*.

*exp1* `-gt` *exp2*
True if *exp1* is greater than *exp2*.

*exp1* `-le` *exp2*
True if *exp1* is less than or equal to *exp2*.

*exp1* `-ge` *exp2*
True if *exp1* is greater than or equal to *exp2*.

You can construct compound expressions from these primitives, listed in decreasing order of precedence:

`(`*expression*`)`
True if *expression* is true. Used to group expressions.

`!` *expression*
True if *expression* is false.

*expression1* `&&` *expression2*
True if *expression1* and *expression2* are both true.

*expression1* `||` *expression2*
True if either *expression1* or *expression2* is true.

### Additional Korn Shell Metacharacters

The Korn shell treats several two-character tokens as if they were a single metacharacter.

The token `|&` causes asynchronous execution of the preceding command or pipeline as does the ampersand alone, but with a rather unusual redirection of input and output. The input and output are both redirected to the (parent) shell process from which the command was dispatched. To read and write lines of the input so created, you execute the `read` and `print` commands with the `-p` option from the parent shell. These commands are described later in ''Built-in Commands.''

Only one command redirected in this fashion can be active at any given time.

The tokens `&&` and `||` join two commands together, conditionally performing the command that follows them. Here is the format of commands joined this way:

*command* `&&` *conditionally-run-command*

*command* `||` *conditionally-run-command*

When *conditionally-run-command* follows `&&`, it is executed only if the preceding command exits with a zero exit status, which is usually an indication of successful command completion. When *conditionally-run-command* follows the `||` token, it is executed only if the preceding command exits with a nonzero exit status.

The precedence of the metacharacter operators is given in the following list. Those in the first line are given higher precedence than those in the second line.

```
&&  ||
;   &   |&
```

### Variables and Arrays

Variables do not have to be declared before they are used.

Usually the variable name is composed in accordance with the rules for identifiers. At least this is the case for all user-defined variables. For certain built-in variables, names have been previously established that do not meet the criteria for identifiers. Here are some examples:

```
$*
$@
$#
$?
$-
```

```
$$
$!
```

A Korn shell variable has several other attributes besides its associated value. Using the `typeset` command, you can set some of these attributes. For more technical information regarding `typeset`, see ''Built-in Commands,'' later.

Variables with the export attribute set are passed along to subshells and commands as part of their inherited execution environment. The passed variable values, and the passed attribute values (such as export), are those in effect at the time the exported attribute is made active.

Variables can be assigned the string or number value denoted as *value*, as follows:

*name*=*value*
[*name*=*value*]...

To make an assignment by using an arithmetic expression, use one line per assignment in the following format:

`((`*name*=*expression*`))`

For variables with the integer attribute set, arithmetic evaluation is performed on the assignment even if the double open and close parentheses are omitted. By declaring `count` as an integer with the format

```
typeset -i count
```

the result you obtain from the following commands becomes the same result:

```
count="3 + 2"
((count=3 + 2))
```

One way to establish values for positional variables is to use the `set` command. (See the description of `set` in ''Built-in Commands,'' later.)

When you are writing shell script programs, the positional variables are automatically set by the invoking shell to support the passing of the command-line arguments for use inside the program. In such a case, `$0` is set to the script name. See ''Specialized Command-Line Processing for Scripts,'' earlier.

The shell supports one-dimensional arrays. An element of an array variable is referenced by a subscript. A subscript is denoted by an open bracket (`[`), followed by an arithmetic expression, followed by a close bracket (`]`) as in the following example:

```
${student[32]}
```

To assign values to an array, use the format

`set -A` *name value...*

The value of all subscripts must be in the range 0 through 1024. Arrays need not be declared prior to their use. Any reference to a name with a valid subscript is legal, and an array is created if necessary. Referencing an array without an index subscript is equivalent to referencing the first element.

The following list shows the various syntaxes you can use when referring to variables:

`${`*variable-name*`}`
`${`*array-name*`[`*index-expression*`]}`
: Replace the variable or array specified with its value, if any. The braces are required when *variable-name* is followed by a letter, digit, or underscore that is not supposed to be interpreted as part of its name. If *variable-name* is specified as `*` or `@`, all the values of positional variables, starting with `$1`, are used as the substitution text (separated by a field-separator character).

  If the value of *index-expression* is given as `*` or `@`, the value for each of the elements of the array is used as the substitution text (separated by a field-separator character).

`${#`*variable-name*`}`
: Replaces the variable specified with the length of the string stored in *variable-name*. If *variable-name* is specified as an asterisk (`*`) or an "at" sign (`@`), the replacement value is the number of positional parameters that are set.

`${#`*array-name*`[*]}`
: Replaces the array specified with the number of elements in the array.

`${`*variable-name*`:-`*word*`}`
`${`*variable-name*`-`*word*`}`
: Replace the variable specified with its value if it is set and it is not empty; otherwise the replacement value is *word.*

  In the second form, the replacement value is *word* only if the variable is unset.

`${`*variable*`:=`*word*`}`
: Replaces the variable specified with its value if it is set. If it is unset or is empty, this syntax assigns *word* to the variable and also uses *word* as the replacement value. You cannot assign positional parameters this way.

${*variable-name*:?*word*}
${*variable-name*?*word*}
Replace the variable specified with its value if it is set and it is not empty; otherwise print *word* on the standard error and exit from the shell. If *word* is omitted, print a standard message.

In the second form (without a colon preceding the question mark), the message is displayed only if the variable is unset.

${*variable-name*:+*word*}
${*variable-name*+*word*}
Replace the variable specified with its value if it is set and it is not empty; otherwise, substitute nothing.

In the second form, if the value of the variable specified is an empty string, then *word* is used as the replacement value.

${*variable-name*#*pattern*}
${*variable-name*##*pattern*}
Replace the variable specified with its truncated value, if *pattern* matches the beginning of its value. The truncated value is its original value less the characters matched by the search pattern. Otherwise, the replacement value is the complete value of *variable-name*. In the first form, the smallest matching pattern is deleted. In the second form, the largest matching pattern is deleted.

${*variable-name*%*pattern*}
${*variable-name*%%*pattern*}
Replace the variable specified with its truncated value, if *pattern* matches the end of its value. The truncated value is its original value less the characters matched by the search pattern. Otherwise, the replacement value is the complete value of *variable-name*. In the first form, the smallest matching pattern is deleted. In the second form, the largest matching pattern is deleted.

In the preceding variable-referencing syntaxes, the value of *word* is not examined unless it is to be used as the substituted string. Thus, in the following example, `$HOME` is subject to variable substitution only if `d` is not set or is empty:

```
echo ${d:- $HOME}
```

**Functions**

You use the `function` keyword to define shell functions. When a function is declared, its command list is read, processed for substitutions such as aliases, and stored in memory.

You can use either of these syntaxes for a function declaration:

`function` *identifier* `{` *list*`;}`

*identifier*`()` `{` *list*`;}`

The command list is executed whenever the function is referenced.

Inside the declared command list for a function, references to positional variables (`$1`, `$2`, `$3`, and so forth) access the values passed in a parameterized reference to the function:

*function-name* `(`[*positional-arg*]...`)`

The only way to make the original positional parameters available inside the declared commands for the function is to rereference them when calling the function:

*function-name* `(one two "$@" )`

Because of the introduction of two new positional parameters in the preceding reference to a function, the first positional parameter of the shell would have to be referenced in terms of `$3` inside the declared commands for the function *function-name*.

Another common element inside the declared commands for a function is the command `return`, which is used to exit the function and return to the point from which the function was called. See the description of `return` in ''Built-in Commands,'' later, for a description of its argument and its default return value. When the `return` command is not used as the means to exit a function, the value of the function is the exit value of the last command executed within the declared commands for a function.

You can use `typeset` with the `-f` option or the `+f` option to list the functions that have been declared. (See ''Built-in Commands,'' later.) If you use the `-f` option, `typeset` lists the declared commands for the functions as well as the function names. You can undefine functions by using the `unset` command with the `-f` option. (See ''Built-in Commands,'' later.)

Ordinarily, functions are not exported to subshells, such as the subshells that execute shell scripts. However, you can export functions from the parent shell where they were declared to any of its subshells by using `typeset` with the `-x` and `-f` options. To allow functions to remain defined across login sessions, place the function declarations in the file referenced by `ENV`, as described earlier in ''Establishing Preference Settings.''

Errors within functions cause an exit from the function, returning execution to the point from which it was called. For a function call that runs without errors, the last command (possibly a `return`) executes followed by any commands set to run upon the receipt of an `EXIT` signal (see the description of the `trap` command in ''Built-in Commands,'' later).

Functions execute in the same process as the shell in which they were invoked. They share all of the shell's opened input and output streams, all of its traps (other than `EXIT` and `ERR`), and its current working directory. An `EXIT` signal can be specified for a `trap` command placed inside the declared commands for a function. In such a case, the command associated with the trap runs after the function completes.

The shell and the called function ordinarily share variables. However, the `typeset` command can be used within a function to create instances of local variables. The scope of these local variables includes the current function and all of the functions that it calls.

**Lexical Rules for Identifiers**

The names you choose for variables and functions can vary widely, but are subject to certain limitations. Throughout this manual page, the use of the term ''identifier,'' and the use of the placeholder *identifier*, imply adherence to these rules.

An identifier must be a sequence of letters, digits, or underscores, starting with a letter or underscore. Identifiers cannot include shell metacharacters.

**Built-in Commands**

Sometimes it is useful to call a subshell simply for access to the built-in commands that are not available otherwise. For example, the capabilities of the built-in `print` command are not available in other shells, so you may need to call `ksh` solely to use its version of the `print` command. In a case such as this, you may need to use the `-c` option for `ksh` as described in ''Arguments,'' earlier in this manual page.

The commands listed in this section execute within the same process as `ksh` rather than through an independent process based upon an executable A/UX file. For these commands, the usual search moderated by the `PATH` variable need not be performed. For this reason, these commands are called ''built-in'' commands for the `ksh` shell.

Input/output redirection is permitted. Unless otherwise indicated, the output is written on the standard output channel (file descriptor 1).

Error processing differs slightly for commands that are preceded by two daggers (††). When one of these commands produces an error within a shell script, both the shell script and the command are terminated.

Commands that are preceded by one dagger symbol (†) are given the following special treatments:

- One or more variable assignments preceding the command remain in effect after the command runs.
- These commands are executed in a separate process when invoked as part of a command substitution.

†
: [*arg*]...
Runs a null command that returns a 0 exit code. The arguments are processed in terms of argument parsing and variable substitution.

††
. *file* [*positional-value*]...
Reads and executes commands from *file* before returning to the interactive interpretation of command lines. The commands are executed in the current shell environment. The search path specified by PATH is used to find the directory containing *file*. If any arguments are given, they become the values of the positional parameters. Otherwise, the positional parameters are unchanged.

alias
alias *alias-name*
alias [-tx] [*alias-name=command-line*]...
Display or create aliases. To display a list of the aliases that have been set up already, use alias with no other arguments. To display a single alias, use the second form of the command. To set up an alias to refer to a command line, use the third form of the command.

A trailing space in *command-line* causes the first command name to be checked for alias substitution. Use the -t option to set and list tracked aliases. The value of a tracked alias is the full pathname corresponding to the given name. The value becomes undefined when the value of PATH is reset, but the aliases continue to be tracked aliases afterwards. The -x option is used to either establish or print exported aliases, depending on whether you supply an argument string. An exported alias remains defined in subshells, including those needed to run shell scripts. The alias command returns a 0 exit status unless you specify a name for which no alias has been defined.

bg [%*job*]
Places the specified job in the background. The current job is put in the background if *job* is not specified.

†
break [*n*]
Exits from the enclosing for, while, until, or select loop, if

any. If you specify *n*, `break` exits from *n* levels of nested control structures (if they exist).

`cd` [*directory*]
`cd` *old new*
Reset the current working directory. The first form of the command changes the current directory to *directory*. If you specify *directory* as `-`, the directory is changed to the previous directory. The setting for the shell variable `HOME` is used as the default directory when no directory argument is specified. After successful execution, `cd` sets the variable `PWD` to the new current directory.

If the directory can be found in any of the directory paths stored in the variable `CDPATH`, then *directory* is expanded to the first base pathname in `CDPATH` that contains `directory`. For example, if a directory named `/usr/tools` exists and `/usr` is one of the directories stored in `CDPATH`, you can enter

```
cd tools
```

to establish `/usr/tools` as your current directory.

Alternative directory names are separated by colons (`:`). Initially, `CDPATH` is empty so that the current directory is used as the search directory. To ensure that the current directory will continue to be part of the search order, include a null directory in the list of paths:

```
CDPATH=::$HOME/Tools
```

When you specify *directory* with a leading slash (/), no search is performed. Otherwise, each directory in the `CDPATH` list is searched for *directory*.

The second form of `cd` substitutes the string *new* for the string *old* in `PWD`, the current directory name, and tries to change to this new directory.

†
`continue` [*n*]
Resumes the next iteration of the enclosing `for`, `while`, `until`, or `select` loop immediately. If you specify *n*, `continue` resumes execution at the beginning of the *n*th enclosing loop.

`echo` [*arg*]...
Displays arguments after the interpretation of shell metacharacters. The built-in `echo` command writes its arguments (separated by blanks and terminated by a return on the standard output). See `echo`(1) for additional information. The `echo` command is useful for producing diagnostics in shell programs and for writing constant data on pipes. To send diagnostics to the standard error file, use

```
echo string1>&2
```

†

`eval` [*command-producing-arguments*]

Processes its arguments for command and variable substitution and for filename generation before submitting the resulting output to the current shell for further (and similar) processing as a command line. The following command sequence displays your home directory because variable substitution is performed twice:

```
homeprint='echo $HOME'
eval $homeprint
```

†

`exec` *command*

`exec` *redirection-request*

Treat the arguments as command input, but do not return to the parent shell, as if the parent shell had exited upon the completion of the command(s). The command(s) replace the shell without creating a new process. Any previously open file descriptors above 2 are closed when this command invokes another program.

You can use the second form of the command to reset the files (or devices) associated with various file descriptors, such as standard input, standard output, and standard error. (See "Input/Output Redirection," earlier.) Unlike redirections that are requested for other commands, these redirections affect the current shell's execution environment, and, through inheritance, the execution environment of any commands subsequently run.

†

`exit` [*status*]

Causes the shell to exit with the exit status specified by *status*. If *status* is omitted, the exit status is that of the last command executed. An end-of-file character also causes the shell to exit unless you have established the `ignoreeof` option. (See the description of `set`, later in this list.)

††

`export` [*variable*]...

Sets the export attribute for the named variables.

††

`fc` [`-e` *editor*] [*-nlr*] [*lineno*]

`fc` [`-e` *editor*] [*-nlr*] [*start-command*] [*end-command*]

`fc -e -` [*old=new*] [*command*]

`fc -l [rn]` *n-history-lines*

Display, optionally edit, and dispatch previous commands. The redispatched commands, if any, are displayed in their final edited form upon leaving editing mode, then executed.

In any of the command formats, the operation of `fc` depends on reasonable settings for the variables `HISTSIZE` (the maximum number of commands recallable) and `HISTFILE` (the name of the file in which the command history is saved). Refer to the earlier section ''Establishing Preference Settings'' for more introductory information.

If you specify a negative number for *lineno*, *start-command*, *n-history-lines*, or *end-command*, the system calculates the final line number by reducing the most recent command number by an amount equal to the absolute value of the number you supplied.

The arguments *start-command*, *end-command*, and *lineno* can be specified as numbers or as strings. Specify a string to locate the most recent command that begins with a particular string.

If you don't specify *editor*, the value of the variable `FCEDIT` is used. If `FCEDIT` is empty, `/bin/ed` is used.

Use the last form of the command, requiring the `-l` option, to list the given number of recently executed commands along with their line numbers. To reverse the order in which commands are listed, include the `-r` option. To prevent the line numbers from being listed along with the command, include the `-n` option.

In the first form, a preceding command corresponding to *lineno* is rerun after it has been edited with *editor*.

In the second form, a preceding range of commands starting with *start-command* and continuing through *end-command* are rerun after it has been edited with *editor*.

When editing is complete, the edited command(s) are executed. If *end-command* is not specified, only *start-command* is executed.

If you specify neither *start-command* nor *end-command*, the most recent command is used.

Use the third form of the command, requiring the editor name to be supplied as `-`, to skip the editing phase and to reexecute the command. In this case, you can use a substitution of the form *old=new* to modify the command ''on the fly'' before execution.

For example, suppose `r` is aliased to `'fc -e -'`. If you enter

```
r bad=good c
```

the most recent command that starts with the letter `c` is executed once the first occurrence of the string `bad` is replaced with the string `good`.

`fg` [*%job*]
Converts a previously run command from background to foreground mode. If *job* is specified, it is brought to the foreground. Otherwise, the most recent background job is brought to the foreground.

`getopts` *opt-string name* [*arg*]...
`getopts` [`:`][*flag-letter*[`:`]]...*var-name*[*script-opts*]
Parse the string *script-opts* into its component options as does the stand-alone command `getopt`. For each call to `getopts`, the variable *var-name* is assigned the next option letter parsed from the string *script-opts*. If a plus sign precedes the option in *script-opts*, *var-name* will contain a leading plus sign as well as the flag letter. You can specify a trailing colon for any *flag-letter* argument. A trailing colon causes `getopts` to expect that option to be followed by its own argument inside the string *script-opts*. When this parsing mode is established, the argument parsed is stored in the variable `OPTARG`. (The options can be separated from the argument by blanks.) If *script-opts* is omitted, the positional parameters are used.

The parsing rules for `getopts` interpret any letter in *script-opts* preceded by a + or a - as a distinct option. When it parses a substring that doesn't begin with either a plus or a minus sign, or that consists of two hyphens (`--`), `getopts` assumes that it has reached the end of the options list. Any further calls to `getopts` produce an error message and a nonzero exit value. The `getopts` program keeps a count of the items parsed so far. The number of the next item that will be parsed is available in the variable `OPTIND`.

Errors are also reported when an option is parsed that is not recognized as one of the *flag-letter* items specified. A leading `:` before the first *flag-letter* argument changes the mode with which `getopts` handles certain types of errors so that it doesn't report an error message. In this mode, `getopts` stores the letter of an invalid option in `OPTARG`, and stores either `?` or `:` in *var-name*. The value stored in *var-name* will be a question mark when the error is due to receipt of an unexpected option; it will be a colon when a required option argument is missing.

`jobs` [`-l`]
Lists the active jobs. If you specify the `-l` option, the list includes the process ID for each job along with the usual information.

`kill` [*-signal*] *process-no*...
Sends either the `TERM` (terminate) signal or the specified signal to the

specified jobs or processes. Signals are given either by number or by name (as given in `/usr/include/signal.h`, stripped of the prefix `SIG`). Use `kill` with the `-l` option to list the signal numbers and names. If the signal you send to a job that is stopped is `TERM` (terminate) or `HUP` (hangup), `kill` sends a `CONT` (continue) signal. The argument *process-no* can be either a process ID or a job. See also `kill`(1).

`let`[ *variable*=*expr*]...
Evaluates each arithmetic expression, *expr*, assigning the result to the named variable. All calculations are executed with long integers, and no check for overflow is performed. Expressions consist of constants, variables, and operators. The following set of operators, listed in order of decreasing precedence, have been implemented:

`-` unary minus
`!` logical negation
`* / %`
multiplication, division, modulo
`+ -`
addition, subtraction
`<= >= < >`
comparison
`== !=`
equality, inequality

Subexpressions in parentheses are evaluated first and can be used to override the precedence rules listed. The evaluation within a precedence group is from right to left for the = operator and from left to right for the others.

†
`newgrp` [*newgrp-arg*]...
Runs the `newgrp` command in place of the current shell as if the command

```
exec /bin/newgrp...
```

had been used.

`print` [`-nprRs`][`-u`*filedes*] [*arg*]...
Prints its arguments on standard output, separating them with spaces, and normally adding an end-of-line character after the arguments.

The `print` command accepts the following options:

`-p` Causes the arguments to be written onto the pipe of the process spawned with `|&` instead of to standard output.

`-n` Suppresses the addition of an end-of-line character to the end of the output string.

`-r`
`-R` Cause a backslash (/) to be interpreted as data rather than as a metacharacter when it precedes `a`, `b`, `c`, `f`, `n`, `r`, `t`, `v`, `\`, or `0`. The `-R` option also affects the way `print` recognizes command options so that all subsequent options other than `-n` are treated as data rather than as elements of the command.

`-s` Causes the arguments to be written to the history file rather than to standard output.

`-u`*filedes*
Specifies the one-digit file descriptor on which the output will be placed. The default is 1. Using this option is similar to redirecting output in the normal way, except that the `print` command does not cause the file (if any) to be opened and closed or the file descriptor to be duplicated each time.

`pwd`
Displays the currently selected working directory.

`read [-prs][-u`*filedes*`] [`*variable*`?`*prompt*`]`
[*variable*]... Reads a line of text input, assigning it to *variable*. The shell reads one line of input, parses it into fields, using the characters in `IFS` as separators, and assigns the resulting field values to the specified variables, one field per variable. If there are fewer variables specified than fields parsed, the last variable specified is assigned the contents of two or more fields. If no variable names are specified, the input text is stored in the variable `REPLY`. If the shell is running interactively and the first field contains a `?`, the remainder of that field is treated as an input prompt. The return code is 0 unless an end-of-file character is encountered.

The `read` command accepts the following command options:

`-p` Causes the input line to be taken from the input pipe of a process spawned by the shell by means of `|&`. If an end-of-file character is received, it closes the so-redirected process and another so-redirected process can be spawned.

`-r` Reads in a raw character mode so that a backslash (`\`) is treated as data rather than as a metacharacter.

`-s` Saves the input text in the history file.

`-u[`*filedes*`]`
Specifies a one-digit file descriptor, selecting a source of text stream input. You can use the built-in command `exec` to

establish the file or device to be associated with a particular file descriptor. The default value of *n* is 0. If the file descriptor specified is open for writing and is a terminal device, any prompt specified is sent to that terminal instead of the standard error.

††

`readonly` [*variable*=*value*]
`readonly` [*variable*]...

Make the specified variables read-only. These variables cannot be changed by subsequent assignment.

†

`return` [*n*]

Returns execution to the original place in a script where a user-defined function was called, with the return status specified by *n*. If you omit *n*, the return status is that of the last command executed. If you invoke `return` outside the declared commands for a *function*, it has the same effect as the built-in `exit` command.

`set`
`set` [`aefhkmnostuvx`] [`-o` *option*]... [*positional-param*]...
`set -A` *array* [*value*]...
`set +A` *array* [*value*]...

Display or set operating modes or array elements. To display the options that are currently established, use the first form of the command, with no arguments. This format displays the values of all variables affecting shell operation.

To set modes of operation for the shell, use the second form of the command which has many selectable options (`aefhimnostuvx`). See the description of each of these options in the ''Arguments'' section at the beginning of this manual page.

The second form of the command resets the positional parameters. The supplied values are parsed (as described in ''Argument Parsing'') and each resulting argument is assigned to the variables: `$1`, `$2`, and so forth.

Use the third form to nullify (unset) and reset the values stored in *array*, assigning them new values for as many elements as there are values offered. Use the fourth form of the command to add and assign new members to an array. If you specify `+A` rather than `-A`, the old values are not unset first, and the number of array elements is increased by the number of values given.

†

`shift`[*n*]

Reassigns the value of each of the positional parameters according to

the value stored in the parameter *n* positions away. For example, `shift 1` causes `$1` to be assigned the value of `$2`, `$2` to be assigned the value of `$3`, and so forth. The default shift value for *n* is 1. You can replace *n* with any arithmetic expression that evaluates to a nonnegative number less than or equal to the total number of positional parameters set, as given by `$#`.

`test` *argument...*
Evaluates its arguments to produce a Boolean value as described in `test`(1). The same functionality is available through the `[[` *argument* `]]` construct as described earlier in "Conditional Expressions."

†
`times`
Prints the accumulated user and system times for the shell and for processes run from the shell.

†
`trap` *command* [*signal*]...
`trap - ` *signal...*
`trap "" ` *signal...*
`trap`
Execute the command specified when the shell receives the signal(s) named. The value of *signal* can be a number or the name of the signal. Trap commands are executed in order of signal number. When entered in the second format shown, where the command is specified as `-`, `trap` resets the handling of the specified signals to their default treatment. When entered in the third format shown, without any arguments, `trap` prints a list of the already established commands for each signal along with the signal number. When entered in the fourth format shown, where the command is specified with the null string, signals are set to be ignored for the current shell and any subshells. Within subshells, you cannot set a trap on a signal that was set to be ignored by the parent shell.

If *signal* is `ERR`, *command* is executed whenever a command has a nonzero exit code. This trap is not inherited by functions. If *signal* is `0` or `EXIT` and the `trap` statement is executed inside the declared commands for a function, the command is executed after the function completes. If *signal* is `0` or `EXIT` for a `trap` command entered normally (not inside a function), the associated command is executed

upon exit from the shell.

††
`typeset -i[Hrtx][LR[Z]`*width*`]] [`*variable*`[=`*integer*`]]...`
`typeset -i[Hrtx][`*base*`] [`*variable*`[=`*integer*`]]...`
`typeset +i[Hrtx][`*variable*`[=`*integer*`]]...`
`typeset -f[tu] [`*function*`]...]`
`typeset +f[tu] [`*function*`]...]`

Set, unset, or display the attributes and values for shell integer variables or shell functions. All options that are not described here are the same as those described in the next list, where `typeset` is treated more generally. Options described here are either exclusively for integers or functions, or they function differently for integers or functions.

`-f`
`+f` Cause the remaining arguments to be interpreted appropriately for functions. Use `+f` to turn off the trace mode (`-t`), unresolved name (`-u`), and (`-x`) exported attributes by following it with the appropriate option letter.

`-i[`*base*`]`
`+i` Cause the values of the specified variable(s) to be treated as integers, making arithmetic speedier. If *base* is nonzero, it defines the output arithmetic base; otherwise, the first assignment determines the output base. Use `+i` to turn off the read-only (`-r`), tagged (`-t`), and exported (`-x`) attributes by following it with the appropriate option letter.

`-t` *function...*
Specifies that trace mode will be in effect when the specified function is run. To be effective, this option must be preceded by the `-f` option.

`-u` Declares the specified function(s) as currently undefined. The `FPATH` variable is searched to find the function definition when the function is referenced. To be effective, this option must be preceded by the `-f` option.

`-x` Marks the named function(s) for automatic export, allowing the function(s) to remain in effect in subshells in the same process environment. To be effective, this option must be preceded by the `-f` option.

††
`typeset`
`typeset +`

`typeset -`
`typeset -[Hlrtux][LR`*width*`] [`*variable*`[=`*value*`]]...`
`typeset +[Hlrtux] [`*variable*`[=`*value*`]]...`

Set, unset, or display the attributes and values for shell variables. If no arguments are specified (the first command format shown), the names and attributes of all variables are displayed. With no arguments but + (the second command format shown), `typeset` displays the names of all variables, but not their values.

With no arguments but - (the third command format shown), `typeset` displays the names and current values of all variables.

When variables are specified as arguments, `typeset` changes their attributes in accordance with any options you supply. With options that can be toggled on and off, such as `H`, `l`, `r`, `t`, `u`, and `x`, `typeset` activates the setting when you precede the option letter with a hyphen (-), and disables it when you precede the option letter with a plus (+).

When no variables are specified as arguments, yet option letters are present, `typeset` lists the names and values of the variables that have the specified options enabled. In such cases, if you precede the option letters with a plus sign instead of a minus sign, only the names of the variables that have the named options set are displayed.

When this command is invoked inside a function, a local instance of the variable is created. If a global variable by the same name exists, then its value outside of the function corresponds to that of the global variable.

The following options are accepted:

`-H` Provides A/UX-to-host *namefile* mapping on non-UNIX machines.

`-l` Converts all uppercase characters assigned to the named variable(s) to lowercase. Turns off the uppercase option, `-u`.

`-L` Left-justifies and removes leading blanks from *value*. If *width* is nonzero, this option defines the width of the field; otherwise the width is determined by the width of the value first assigned to the variable. When a value is assigned to the variable, it is filled on the right with blanks or truncated, if necessary, to fit into the field. Leading zeros are removed if the `-Z` option is also set. The `-R` option is turned off.

`-r` Makes the named variable(s) read-only. These variables cannot be changed by subsequent assignment.

`-R` Right-justifies and adds leading blanks. If *width* is nonzero, this option defines the width of the field; otherwise the width is

determined by the width of the value first assigned to the variable. The field is filled with blanks or truncated from the end if the variable is reassigned. The `L` option is turned off.

`-t` Sets the ''tagged'' attribute for the named variable(s). Tags are user-definable and have no special meaning to the shell.

`-u` Converts all lowercase characters assigned to the named variable(s) to uppercase. Turns off the lowercase option, `-l`.

`-x` Marks the named variable(s) for automatic export to the environment of subsequently executed commands.

`-Z`
`[-Z]`*width*
Establishes leading zero as the fill character when right-justifying a numeric value in a field *width* characters wide. A variable need only contain a single digit as the first nonblank character to be treated as a numeric value. This option is incompatible with the left-justification (`-L`) option. Using this option is equivalent to using the `-R` and `-Z` options together. The width can also be determined dynamically based on the content first assigned to the variable.

`ulimit` `[acdfmnpstv]`[*limit*]
`ulimit -H[acdfmnpstv]`[*limit*]
`ulimit -S[acdfmnpstv]`[*limit*]
Set or display a resource limit. The available resource limits are listed later in this description. Many systems do not contain one or more of these limits. The limit for a specified resource is set when a *limit* number is specified. Alternatively, you can specify `unlimited` for *limit*. Use the `H` option or the `S` option to specify whether the hard limit or the soft limit is desired. A hard limit cannot be increased once it is set. A soft limit can be increased up to the value of the hard limit. If you specify neither the `H` or the `S` option, the limit applies to both. The current resource limit is printed when *limit* is omitted. In this case the soft limit is printed unless `H` is specified. When more than one resource is specified, the limit name and unit are printed before the value. If no option is given, `-f` is assumed.

`-a` Lists all of the current resource limits.

`-c` Displays the number of 512-byte blocks available for core dumps.

`-d` Displays the number of kilobytes available for the data area.

`-f` Displays the number of 512-byte blocks available for files written by child processes (files of any size may be read).

`-m` Displays the number of kilobytes available for physical memory.

`-n` Displays the number of file descriptors available.

`-p` Displays the number of 512-byte blocks available for pipe buffering.

`-s` Displays the number of kilobytes available for the stack area.

`-t` Displays the number of seconds available for each process.

`-v` Displays the number of kilobytes available for virtual memory.

`umask`
`umask` *complemented-chmod-digits*
`umask` *chmod-opstring*
Set the user file-creation mask to the octal value *complemented-chmod-digits*. (See `umask`(2).) You can also specify a *chmod-opstring* argument as described in `chmod`(1). In that case, the new value of `umask` is recomputed based on the old value and the change in that value requested by *chmod-opstring*. If *umask* is entered with no argument, the current value of the mask is printed.

`unalias` *alias...*
Removes the named aliases.

`unset` [`-f`] *variable...*
Unsets the named variables. Unsetting these variables erases their values and their attributes. Read-only variables cannot be unset. If the `-f` option is set, the names refer to function names. Unsetting `ERRNO`, `LINENO`, `MAILCHECK`, `OPTARG`, `OPTIND`, `RANDOM`, `SECONDS`, `TMOUT`, and `_` removes their special meaning, even if you subsequently assign them values.

`wait` [*pid*]
`wait %`*n*
Wait for the specified child process and report its termination status. If *process* is not given, `wait` suspends shell operation until all currently active child processes terminate. The exit status from this command is the same as that of the process on which it was waiting. See "Controlling Jobs Not in the Foreground," earlier, for a description of the format of *n*.

`whence` [`-pv`] *name...*
For each *name*, displays information about how a command named *name* would be interpreted. The output could include information about command locations on a given system and account, and about current alias settings.

The -v option produces a more verbose report. The -p option causes a path search to take place even when *name* is an alias, a function, or a reserved word.

**Shell-Maintained, Built-in Variables**

The following variables are automatically maintained by ksh. If you unset some of these variables, ksh removes their special meaning even if you subsequently set them.

! Contains the process number of the last background command invoked.

# Contains the number of positional parameters in decimal.

$ Contains the process number of this shell.

- Contains the preferences currently set, whether they were set upon shell invocation or through the set command.

? Contains the decimal exit value returned by the last executed command.

_ (Underscore) Contains the last argument of the previous command. This parameter is not set for asynchronous commands. This parameter is also used to hold the name of the matching MAIL file when the system is checking for mail. Finally, the value of this parameter is set to the full pathname of each program the shell invokes and is passed in the environment. Unsetting this variable removes its special meaning.

A_z
Contains information about exported variables that have special meaning, or that have been made read-only.

ERRNO
Contains the value of *errno* as set by the most recently failed system call. This value is system dependent and is intended for debugging purposes. Unsetting this variable removes its special meaning.

LINENO
Contains the line number of the current line within the script or function being executed. Unsetting this variable removes its special meaning.

OLDPWD
Contains the previous working directory set by the cd command.

OPTARG
Contains the value of the last option argument processed by the getopts built-in command. Unsetting this variable removes its special meaning.

OPTIND
Contains the index of the last option argument processed by the `getopts` built-in command. Unsetting this variable removes its special meaning.

PPID
Contains the process number of the parent of the shell.

PWD
Contains the present working directory set by the `cd` command.

RANDOM
Generates a random integer each time this pseudo variable is referenced. You initiate the sequence of random numbers by assigning a numeric value to `RANDOM`. Unsetting this variable removes its special meaning.

REPLY
Contains the error message set through the `select` command and displayed by the `read` command when no arguments are entered.

SECONDS
Contains the duration of time that the shell has been running, in seconds. If this parameter is assigned a value, the value returned is the value that was assigned plus the number of seconds since the assignment. Unsetting this variable removes its special meaning.

**Other Built-in Variables**

The following variables are used by the shell. In A/UX, the default values shown may have been set (by means of a `.profile` or `.kshrc` file) to different values.

CDPATH
Contains a list of directory paths used by the `cd` command to expand arguments that match the base component of one of the directory paths.

COLUMNS
Defines the width of the edit window, if set. Applies to shell edit modes and to the display of character-oriented menus through the `select` command.

EDITOR
Contains the user preference for choice of command editor. If the value of this variable ends in `emacs`, `gmacs`, or `vi` and the `VISUAL` variable is not set, the corresponding command option is turned on. (See "Built-in Commands," earlier.)

ENV
Contains the pathname for the file from which initial commands are

read and performed by each new ksh process as it starts up. By default, this variable is set to the file .kshrc in your home directory. (Variable substitution is performed again on the value stored in ENV to help generate the final pathname, permitting the use of a reference to yet another variable such as $HOME.)

This ''startup'' file is typically used for alias and function definitions that you want to remain available in any subshells you might run.

FCEDIT
Contains the default editor name for the fc command.

IFS
Contains the characters to be used as field separators. Normally the field separators are the space, tab, and newline characters. Affects command and parameter substitution as well as the built-in command read. When the shell generates certain values such as the value of $*, it uses the first character stored in IFS as the character separating one positional parameter from the next. (Also see ''Filename Generation,'' earlier.)

HISTFILE
Contains the pathname of the file that is used to store previously entered command lines. If this variable is not set, no record of previously entered command lines is kept. (See ''Command Reentry,'' earlier.)

HISTSIZE
Contains the maximum number of previously entered lines that will be available to the command editor, if HISTFILE has been set. The default is 128. Setting this variable to a relatively large value, such as 10000, may result in a delay for each new invocation of ksh.

HOME
Contains the pathname corresponding to the login directory for the current user. When given no arguments, the cd command establishes this directory as the working directory.

LINES
Defines the length of the edit window, if set. Applies to shell edit modes and to the display of character-oriented menus through the select command.

MAIL
Contains the name of your mail file when set. If MAIL is set and if the MAILPATH is not set, the shell notifies you of the arrival of mail in the specified file.

MAILCHECK
Specifies in seconds how often the shell checks for changes in the modification time of any of the files specified by the MAILPATH or MAIL variable. The default value is 600 seconds. When the time has elapsed, the shell checks for mail before issuing the next prompt. Unsetting this variable removes its special meaning.

MAILPATH
Contains a list of filenames separated by colons (:). If this variable is set, the shell informs the user of any modifications to the specified files that have occurred within the last MAILCHECK seconds. Each filename can be followed by a ? and a message that will be printed. The message undergoes parameter and command substitution, and the variable $_ is set to the name of the file that has changed. The default message is produced by the following message:

```
you have mail in $_
```

PATH
Contains a list of the directories to be searched for command files or command scripts.

PS1
Contains the string that the shell displays to prompt you for a command. This string is subject to parameter substitution. By default, it is set to $. The metacharacter ! in the prompt string is replaced by the command number (See "Command Reentry," earlier.) Two successive occurrences of ! produce a single ! when the prompt string is printed.

PS2
Contains the string that the shell displays to prompt you for a block of commands to be executed together. By default, this variable is set to >.

PS3
Contains the string that the shell displays to prompt you for a select choice. By default, this variable is set to #?.

PS4
Contains the string that the shell displays before each line of an execution trace. The value of this variable is expanded for parameter substitution. If PS4 is unset, the execution trace indicator is set to a plus sign (+).

SHELL
Contains the pathname to the login shell preference for a particular account, and is stored in the processing environment. A leading r in

the filename indicates that the login shell is restricted.

TMOUT
Contains the amount of time in seconds that the system will wait for input before exiting. (The shell can be compiled so that it will establish a maximum value for this variable.) Unsetting this variable removes its special meaning.

VISUAL
Contains the user preference for choice of command editor. If the value of this variable ends in `emacs`, `gmacs`, or `vi`, `ksh` turns on the corresponding option regardless of the setting stored in the `EDITOR` variable.

The shell gives default values to `PATH`, `PS1`, `PS2`, `MAILCHECK`, `TMOUT`, and `IFS`. The shell does not set initial values for `ENV` and `MAIL`. Initial values for `HOME`, `MAIL`, and `SHELL` are set by `login` and inherited by the shell as part of its execution environment.

## STATUS MESSAGES AND VALUES

Errors detected by the shell, such as syntax errors, cause the shell to return a nonzero exit status. Otherwise, the shell returns the exit status of the last command executed. (See also the description of the `exit` command in ''Built-in Commands,'' in the ''Description'' section.) If the shell is being used noninteractively, execution of the shell file is abandoned.

The system reports run-time errors for shell scripts by printing the command or function name and the error condition. If the number of the line on which the error occurred is greater than 1, the line number is also printed in square brackets (`[]`) after the command or function name.

## WARNINGS

If a command that is a tracked alias is executed, and then a command with the same name is installed in a directory in the search path prior to the directory where the original command was found, the shell continues to execute the original command. Use the `-t` option of the `alias` command to correct this situation.

Some very old shell scripts use a caret (`^`) as a synonym for the pipe character (`|`). This synonym is not supported in releases of A/UX later than 2.0.

If a command is piped into a shell command, all variables set in the shell command are lost when the command is executed.

Using the `fc` built-in command within a compound command causes the whole command to disappear from the history file.

The built-in dot command ( . ) reads the whole file named *file* before any commands are executed. Thus, for

. *file*

any `alias` and `unalias` commands in *file* will not be available within *file*.

Traps are not processed while a job is waiting for a foreground process. Thus a trap on `CHLD` won't be executed until the foreground job terminates.

Unsetting some special variables removes their special meaning, even if they are subsequently set.

When you log in over a serial line, the command-input editing options may require specific settings of the configuration switches of the associated terminal device.

**FILES**

`$HOME/.profile`
User-specific `ksh` startup settings file

`/bin/ksh`
Executable file

`/etc/passwd`
Password and login-account information file

`/etc/profile`
System-wide `ksh` startup-settings file

`/etc/suid_profile`
File from which startup settings are obtained for subshells invoked to run for a script that has setuid or setgid permission

`/tmp/ksh*`
Temporary file

**SEE ALSO**

`cat`(1), `chmod`(1), `CommandShell`(1), `csh`(1), `echo`(1), `ed`(1), `env`(1), `getopt`(1), `kill`(1), `launch`(1), `login`(1), `newgrp`(1), `nice`(1), `printenv`(1), `ps`(1), `sh`(1), `startmac`(1), `stty`(1), `tee`(1), `vi`(1)

`dup`(2), `exec`(2), `fork`(2), `ioctl`(2), `lseek`(2), `pipe`(2), `ulimit`(2), `umask`(2), `wait`(2), `signal`(3), `rand`(3C), `a.out`(4), `passwd`(4), `profile`(4), `termcap`(4), `terminfo`(4), `environ`(5) in

"Korn Shell Reference" in *A/UX Shells and Shell Programming*

Bolsky, Morris, and David Korn. *The KornShell Command and Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1989.

## NAME

last — displays login and logout times for each user of the system

## SYNOPSIS

last [*name*]... [*tty*]...

## ARGUMENTS

*name*
Specifies the names of users who used the system last.

*tty* Specifies the terminals that were used on the system.

## DESCRIPTION

last will look back in the wtmp file which records all logins and logouts for information about a user, a terminal or any group of users and terminals. Arguments specify names of users or terminals of interest. Names of terminals may be given fully or abbreviated. For example, last 0 is the same as last tty0. If multiple arguments are given, the information which applies to any of the arguments is printed. For example, last root console would list all of ''root's'' sessions as well as all sessions on the console terminal.

The last command reports the sessions of the specified users and terminals, most recent first, indicating start times, duration, and terminal for each. If the session is still continuing or was cut short by a reboot, last so indicates.

The last command with no arguments prints a record of all logins and logouts, in reverse order. Since last can generate a great deal of output, piping it through the more program for screen viewing is advised.

If last is interrupted with an Interrupt signal, (generated by CONTROL-C) it indicates how far the search has progressed in wtmp. If interrupted with a quit signal (generated by a CONTROL-\), last exits and dumps core.

CONTROL-D (EOF) signal does nothing. Therefore exit gracefully from last with an interrupt signal.

## EXAMPLES

The command:

```
last reboot
```

will give an indication of mean time between reboots of the system.

## FILES

/usr/bin/last
Executable file
/etc/wtmp
Temporary file

**SEE ALSO**

acct(1M) in *A/UX System Administrator's Reference*

utmp(4) in *A/UX Programmer's Reference*

## NAME

launch — runs a Macintosh binary application in A/UX

## SYNOPSIS

launch [-adr] *application* [*document*]...

launch -p [*adr*] *application document*...

## ARGUMENTS

-a Runs the Macintosh application asynchronously.

*application*
Specifies the name of the application file to be run.

-d Performs a launch operation that is compatible with applications requiring 24-bit addressing mode. (Such applications are not 32-bit clean.)

*document*
Specifies an individual document to be opened.

-p Prints the specified document. To use the -p option, you must supply a document name in the command line. Using the -p option is equivalent to selecting a document through the Macintosh Finder and then choosing Print from the File menu.

-r Enables certain preprocessing and postprocessing of the standard input and standard output so that carriage return characters are mapped to newlines upon reading input and mapped backed to carriage returns upon writing output.

## DESCRIPTION

launch runs the Macintosh binary application specified. The *application* and *document* arguments act much as do icons selected through the Macintosh Finder.

If your application is in a pair of AppleDouble files, the two files must be in the same directory. You do not specify both filenames; launch automatically looks for the associated header file when you launch an AppleDouble data file.

## EXAMPLES

This command runs the Macintosh binary application MacPaint:

```
launch macpaint
```

This command runs MacPaint and opens the document demo:

```
launch macpaint demo
```

A simpler way to run a Macintosh application from the command line is to enter its name in place of launch. For this method to work, the Macintosh application must be located within an A/UX file system in a

directory specified as one of the search paths in the PATH variable. For example, consider an application named xyz. If it is in AppleDouble format, the header file for xyz has the A/UX filename %xyz. To launch xyz, enter this command:

```
xyz
```

To launch xyz and open the document file abc, enter this command:

```
xyz abc
```

You can nevertheless use the launch command options when invoking a Macintosh application this way—without a direct reference to launch.

**FILES**

/mac/bin/launch
Executable file

**NAME**

lav — displays load average statistics

**SYNOPSIS**

lav

**DESCRIPTION**

lav displays the average number of jobs in the run queue over the last 1, 5, and 15 minutes.

**FILES**

/usr/bin/lav
Executable file

**SEE ALSO**

ruptime(1N), uptime(1)

**NAME**

ld — invokes the link editor for common object files

**SYNOPSIS**

ld [-a*factor*] [-e *epsym*] [-f *fill*] [-ild] [-l*x*] [-m] [-o *outfile*] [-r] [-s] [-t] [-u *symname*] [-x] [-z] [-F] [-L*dir*] [-M] [-N] [-S] [-V] [-VS *num*] *file*...

**ARGUMENTS**

-a*factor*
Specifies the expansion factor to be used to increase the size of the default symbol table.

-e *epsym*
Sets the default entry point address for the output file to be that of the symbol *epsym*.

-f *fill*
Sets the default fill pattern for holes within an output section as well as initialized bss sections. The argument *fill* is a 2-byte constant.

*file* Specifies the file to be processed by ld.

-ild
Generates the sections reserved for use by the incremental loader and retains relocation entries in the new object file (as does the -r option).

-l*x* Searches a library lib*x.a*, where *x* is a string of up to seven characters. A library is searched when its name is encountered, so the placement of this argument is significant. The default library location is /lib.

-m Produces a map or listing of the input/output sections on the standard output.

-o *outfile*
Produces an output object file with the name *outfile*. The name of the default object file is a.out.

-r Retains relocation entries in the output object file. Relocation entries must be saved if the output file is to become an input file in a subsequent ld run. The link editor does not complain about unresolved references.

-s Strips line-number entries and symbol table information from the output object file.

-t Turns off the warning about multiply defined symbols that are not the same size.

-u *symname*
Specifies an undefined symbol in the symbol table. This option is

useful for loading entirely from a library, because initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine. Replace *symname* with the name of an undefined symbol.

`-x` Causes the system not to preserve local (nonglobal) symbols in the output symbol table. Enter external and static symbols only. This option saves some space in the output file.

`-z` Loads the text segment at an offset from 0 so that null-pointer references generate a segmentation violation.

`-F` Creates a demand-paged executable.

`-L`*dir*
Changes the algorithm of searching for `lib`*x*`.a` to look in *dir* before looking in `/lib` and `/usr/lib`. This option is effective only if it precedes the `-l` option on the command line.

`-M` Produces an output message for each multiply defined external definition. However, if the objects being loaded include debugging information, extraneous output is produced. (See the description of `-g` option in `cc`(1).)

`-N` Puts the data section immediately following the text in the output file. Note that the `-N` option must be used either with `/usr/lib/unshared.ld` or with a user-supplied `.ld` file.

`-S` Suppresses the display of progress and error messages unless an error message occurs that results in the termination of the program.

`-V` Produces an output message giving information about the version of `ld` being used.

`-VS` *num*
Causes `ld` to use *num* as a decimal version stamp identifying the `a.out` file that is produced. The version stamp is stored in the optional header.

**DESCRIPTION**

`ld` combines several object files into one, performs relocation, resolves external symbols, and supports symbol table information for symbolic debugging. In the simplest case, the names of several object programs are given, and `ld` combines them, producing an object module that can either be executed or used as input for a subsequent `ld` run. The output of `ld` is left in `a.out`. This file is executable if no errors occurred during the load. If any input file, *filename*, is not an object file, `ld` assumes it is either a text file containing link editor directives or an archive library.

If any argument is a library, it is searched exactly once at the point at which it is encountered in the argument list. Only routines that define an unresolved external reference are loaded. The library (archive) symbol table (described in ar(4)) is searched sequentially with as many passes as are necessary to resolve external references that can be satisfied by library members. Thus, the ordering of library members is unimportant.

The following information about section alignment and MMU requirements should be considered at system installation.

The default section-alignment action for ld on M68000 systems is to align the code (.text) and data (.dataand.bsscombined) separately on 512-byte boundaries. Since MMU requirements vary from system to system, this alignment is not always desirable. This version of ld provides a mechanism to allow the specification of different section alignments for each system, so that you can align each section separately on *n*-byte boundaries, where *n* is a multiple of 512. The default section-alignment action for ld on this system is to align the code (.text) at byte 0 and the data (.dataand.bsscombined) at the 4-megabyte boundary (byte 10487576).

When all input files have been processed (and if no override is provided), ld searches the list of library directories (as with the -l option) for a file named default.ld. If this file is found, it is processed as an ld instruction file (or *ifile*). The default.ld file should specify the required alignment as outlined here. If it does not exist, the default section-alignment action is taken.

The default.ld file should appear as follows, with <*alignment*> replaced by the alignment requirement in bytes:

```
SECTIONS {
        .text : {}
        GROUP ALIGN(<alignment>) : {
                        .data : {}
                        .bss  : {}
                        }
        }
```

*Note:* This system requires a data rounding that is an even multiple of 1 megabyte. (1 megabyte is the segment size.)

For example, a default.ld file of the following form would provide the same alignment as the default (512-byte boundary):

```
SECTIONS {
        .text : {}
        GROUP ALIGN(512) : {
```

```
                        .data : {}
                        .bss  : {}
                        }
        }
```

To get alignment on 2 kilobyte boundaries, you should specify the following `default.ld` file should be specified:

```
SECTIONS {
        .text : {}
        GROUP ALIGN(2048) : {
                        .data : {}
                        .bss : {}
                        }
         }
```

Note that this system requires a data rounding that is an even multiple of 1 megabyte. (1 megabyte is the segment size.)

For more information about the format of `ld` instruction files or the meaning of the commands, see "`ld`Reference" in *A/UX Programming Languages and Tools, Volume 1.*

**WARNINGS**

Through its options and input directives, the common link editor gives you great flexibility; however, if you use the input directives, you must assume some added responsibilities. Input directives should ensure the following properties for programs:

- C defines a zero pointer as null. A pointer to which zero has been assigned must not point to any object. To satisfy this requirement, you must not place any object at virtual address zero in the data space.
- When you call the link editor through `cc`(1), a startup routine is linked with your program. This routine calls `exit()` (see `exit`(2)) after execution of the main program. If you call the link editor directly, you must ensure that the program always calls `exit()` rather than falling through the end of the entry routine.

**FILES**

`/bin/ld`
: Executable file

`/lib/*`
: Various library files and directories

`/usr/lib/*`
: Various library files and directories

`a.out`
: Default output file

**SEE ALSO**

as(1), cc(1)

a.out(4), ar(4) in *A/UX Programmer's Reference*

"ldReference" in *A/UX Programming Languages and Tools, Volume 1*

**NAME**

leave — reminds you when you have to leave

**SYNOPSIS**

leave [*hhmm*]

**ARGUMENTS**

*hhmm*

Specifies the time of day. Replace *hh* with the hour of the day (on a 12 or 24 hour clock) and *mm* with the minutes. All times are converted to a 12 hour clock, and assumed to be in the next 12 hours.

**DESCRIPTION**

leave waits until the specified time, then reminds you that you have to leave. You are reminded 5 minutes and 1 minute before the actual time, at the time, and every minute thereafter. When you log off, leave exits just before it would have printed the next message.

If no argument is given, leave prompts with

```
When do you have to leave?
```

A reply of newline causes leave to exit, otherwise the reply is assumed to be a time. This form is suitable for inclusion in a .login or .profile.

The leave command ignores interrupts, quits, and terminates. It sends messages while other programs are running. To get out of leave, you should either log off or use kill -9, giving its process ID.

**FILES**

/usr/ucb/leave

Executable file

**SEE ALSO**

calendar(1)

## NAME

lex — generates programs for simple lexical tasks

## SYNOPSIS

lex [-c] [-n] [-t] [-v] [*file*]...

## ARGUMENTS

-c Indicates C actions and is the default.

*file* Specifies the input file to be used by lex. Multiple files are treated as a single file. If no files are specified, standard input is used.

-n Does not print out the summary.

-t Causes the lex.yy.c program to be written instead to standard output.

-v Provides a one-line summary of statistics of the machine generated.

## DESCRIPTION

lex generates programs to be used in simple lexical analysis of text.

The input *files* (standard input default) contain strings and expressions to be searched for, and C text to be executed when strings are found.

A file lex.yy.c is generated which, when loaded with the library, copies the input to the output except when a string specified in the file is found; then the corresponding program text is executed. The actual string matched is left in yytext, an external character array. Matching is done in order of the strings in the file. The strings may contain square brackets to indicate character classes, as in [abx-z] to indicate a, b, x, y, and z; and the operators *, +, and ? mean, respectively, any nonnegative number of, any positive number of, and either zero or one occurrence of, the previous character or character class. Thus [a-zA-Z]+ matches a string of letters. The character . is the class of all ASCII characters except newline. Parentheses for grouping and vertical bar for alternation are also supported. The notation *r*{*d*,*e*} in a rule indicates between *d* and *e* instances of regular expression *r*. It has higher precedence than |, but lower than *, ?, +, and concatenation. The character ^ at the beginning of an expression permits a successful match only immediately after a newline, and the character $ at the end of an expression requires a trailing newline. The character / in an expression indicates trailing context; only the part of the expression up to the slash is returned in yytext, but the remainder of the expression must follow in the input stream. An operator character may be used as an ordinary symbol if it is within " symbols or preceded by \.

Three subroutines defined as macros are expected: input() to read a character; unput(*c*) to replace a character read; and output(*c*) to place an output character. They are defined in terms of the standard streams, but you can override them. The program generated is named

`yylex()`, and the library contains a `main()` which calls it. The action `REJECT` on the right side of the rule causes this match to be rejected and the next suitable match executed; the function `yymore()` accumulates additional characters into the same `yytext`; and the function `yyless(`*p*`)` pushes back the portion of the string matched beginning at *p*, which should be between `yytext` and `yytext+yyleng`. The macros *input* and *output* use files `yyin` and `yyout` to read from and write to, defaulted to `stdin` and `stdout`, respectively.

Any line beginning with a blank is assumed to contain only C text and is copied; if it precedes `%%`, it is copied into the external definition area of the `lex.yy.c` file. All rules should follow a `%%`, as in YACC. Lines preceding `%%` which begin with a nonblank character define the string on the left to be the remainder of the line; it can be called out later by surrounding it with `{}`. Note that curly brackets do not imply parentheses; only string substitution is done.

The external names generated by `lex` all begin with the prefix `yy` or `YY`.

Certain table sizes for the resulting finite state machine can be set in the definitions section:

`%p` *n*
: number of positions is *n* (default 2000)

`%n` *n*
: number of states is *n* (500)

`%t` *n*
: number of parse tree nodes is *n* (1000)

`%a` *n*
: number of transitions is *n* (3000)

The use of one or more of the above automatically implies the `-v` option, unless the `-n` option is used.

**EXAMPLES**

The following is output that was generated by `lex`:

```
D       [0-9]
%%
if      printf("IF statement\n");
[a-z]+ printf("tag, value %s\n",yytext);
0{D}+   printf("octal number %s\n",yytext);
{D}+    printf("decimal number %s\n",yytext);
"++"    printf("unary op\n");
"+"     printf("binary op\n");
"/*"    {    loop:
        while (input() != '*');
```

```
switch (input())
            {
            case '/': break;
            case '*': unput('*');
            default: go to loop;
            }
    }
```

**LIMITATIONS**

When given an illegal option, `lex` reports the fact that it has been given an illegal option but then continues to execute with the default options, rather than stopping the execution and printing a usage statement.

**FILES**

`/usr/bin/lex`
Executable file

**SEE ALSO**

`awk`(1), `grep`(1), `sed`(1), `yacc`(1)

`malloc`(3X) in *A/UX Programmer's Reference*

"`lex` Reference" in *A/UX Programming Languages and Tools, Volume 2*

## NAME

`line` — reads one line from the standard input

## SYNOPSIS

`line` *input*

## ARGUMENTS

*input*

Specifies the standard input. Replace *input* with a line of text.

## DESCRIPTION

`line` copies one line (up to a newline) from the standard input and writes it on the standard output. It returns an exit code of 1 on `EOF` and always prints at least a newline. It is often used within shell files to read from the user's terminal.

## EXAMPLES

If you enter:

```
line
Hello world
```

this command will return:

```
Hello world
```

When using the Bourne shell (`sh`(1)), the command:

```
a='line'
hi there
echo $a
```

will return:

```
hi there
```

In the C-shell (`csh`(1)), the command:

```
set a='line'
bye bye
echo $a
```

will return:

```
bye bye
```

## FILES

`/bin/line`

Executable file

**SEE ALSO**

csh(1), ksh(1), sh(1)

read(2) in *A/UX Programmer's Reference*

## NAME

lint — invokes a C program checker

## SYNOPSIS

lint [-a] [-b] [-D*name*[=*def*]] [-h] [-I*dir*] [-l*x*] [-n] [-o *lib*] [-p] [-u] [-U*name*] [-v] [-x] *file...*

## ARGUMENTS

-a Suppresses complaints about assignments of long values to variables that are not long.

-b Suppresses complaints about break statements that cannot be reached. (Programs produced by lex or yacc will often result in many such complaints.)

-D*name*[=*def*]
Defines *name* as if by a #define directive. If no =*def* if given, *name* is defined as 1.

*file* Specifies the file to be checked.

-h Does not apply heuristic tests that attempt to intuit bugs, improve style, and reduce waste.

-I*dir*
Searches for #include files (whose names do not begin with /) in *dir* before looking in the directories on the standard list. When this option is used, #include files whose names are enclosed in double quotes are searched for first in the directory of the *ifile* argument, then in directories named in -I options, and last in directories on a standard list, which, at present, consists of /usr/include. If the -Y option (see below) is specified, the standard list is not searched. For #include files whose names are enclosed in <>, the directory of the *ifile* argument is not searched, unless -I. is specified.

-l*x* Includes an additional lint library, llib-l*x*.ln. For example, you can include a lint version of the Math Library llib-lm.ln by inserting -lm on the command line. This argument does not suppress the default use of llib-lc.ln.

These lint libraries must be in the assumed directory. This option can be used to reference local lint libraries and is useful in the development of multifile projects. To generate llib-lX.ln from llib-lX, use

```
cc -E -C -Dlint llib-lX | \
      /usr/lib/lint1 -vx -H/tmp/lint$$ > llib-lX.ln
rm -f /tmp/lint$$
```

`-n` Does not check compatibility against either the standard or the portable `lint` library.

`-o` *lib*
Causes `lint` to create a new `lint` library that has the name `llib-l`*lib*`.ln`. The `lint` library produced is the input that is given to the second pass of `lint`.

This option simply causes this file to be saved in the named `lint` library. To produce a `llib-l`*lib*`.ln` without extraneous messages, use of the `-x` option is suggested.

The `-v` option is useful if the source file(s) for the `lint` library are just external interfaces (for example, the way the file `llib-lc` is written). These option settings are also available through the use of `lint` *comments* (as shown later in this section).

`-p` Attempts to check portability to other dialects (IBM and GCOS) of C. Along with stricter checking, this option causes all nonexternal names to be truncated to eight characters and all external names to be truncated to six characters and one case.

`-u` Suppresses complaints about functions and external variables used and not defined, or defined and not used. (This option is suitable for running `lint` on a subset of files of a larger program.)

`-U`*name*
Removes any initial definition of *name*, where *name* is a reserved symbol that is predefined by the particular preprocessor. The list of reserved symbols is shown below:

operating system:
unix

hardware:
m68k

UNIX System varient:
_SYSV_SOURCE
_BSD_SOURCE
_AUX_SOURCE

`-v` Suppresses complaints about unused arguments in functions.

`-x` Does not report variables referred to by external declarations but never used.

**DESCRIPTION**

`lint` attempts to detect features of the C program files that are likely to be bugs, nonportable, or wasteful. It also checks type usage more strictly than the compilers. Features currently detected include unreachable statements,

loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, function usage is checked to find functions that return values in some places and not in others, functions that are called with varying numbers or types of arguments, and functions whose values are not used or whose values are used but not returned.

Arguments whose names end with `.c` are taken to be C source files. Arguments whose names end with `.ln` are taken to be the result of an earlier invocation of `lint` with the `-o` option used. The `.ln` files are analogous to `.o` (object) files that are produced by the `cc` command when given a `.c` file as input. Files with other suffixes are warned about and ignored.

The `lint` command will take all the `.c`, `.ln`, and `llib-l`*x*`.ln` (specified by `-l`*x*) files and process them in command line order. By default, `lint` appends the standard C `lint` library (`llib-lc.ln`) to the end of the list of files. However, if the `-p` option is used, the portable C `lint` library (`llib-port.ln`) is appended instead. The second pass of `lint` checks this list of files for mutual compatibility.

Any number of `lint` options may be used, in any order, intermixed with filename arguments. The `-a`, `-b`, `-h`, `-u`, `-v`, and `-x` options are used to suppress certain kinds of complaints.

The `-g` and `-O` options are ignored, but, by recognizing them, the behavior of `lint` is closer to that of the `cc`(1) command. Other options are warned about and ignored. The pre-processor symbol `lint` is defined to allow certain questionable code to be altered or removed for `lint`. Therefore, the symbol `lint` should be thought of as a reserved word for all code that is planned to be checked by `lint`.

Certain conventional comments in the C source will change the behavior of `lint`.

`/*NOTREACHED*/`
: Stops comments about unreachable code at appropriate points. (This comment is typically placed just after calls to functions like `exit`(2).)

`/*VARARGS`*n*`*/`
: Suppresses the usual checking for variable numbers of arguments in the function declaration that follows it. The data types of the first *n* arguments are checked; a missing *n* is assumed to be 0.

`/*ARGSUSED*/`
: Turns on the `-v` option for the next function.

`/*LINTLIBRARY*/`
: Shuts off (at the beginning of a file) complaints about unused

functions and function arguments in this file. This is equivalent to using the -v and -x options.

lint produces its first output on a per-source-file basis. Complaints pertaining to included files are collected and printed after all source files have been processed. Finally, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a complaint stems from a given source file or from one of its included files, the source filename will be printed followed by a question mark.

**EXAMPLES**

The command:

```
lint -b myfile.c
```

checks the consistency of the file myfile.c. The -b option indicates that unreachable break statements are not to be checked. This option might well be used on files that lex generates.

**LIMITATIONS**

exit(2), longjmp(3C), and other functions that do not return are not understood; this causes various lies.

**FILES**

/usr/bin/lint
: Executable file

/usr/lib
: Directory where the lint libraries specified by the -l*x* option must exist

/usr/lib/lint[12]
: File containing first and second passes

/usr/lib/llib-lc.ln
: Declarations for C Library functions (binary format)

/usr/lib/llib-port.ln
: File containing declarations for portable functions (binary format)

/usr/lib/llib-lm.l
: File containing declarations for Math Library functions (binary format)

/usr/tmp/*lint*
: Temporary files

**SEE ALSO**

cc(1), cpp(1), make(1)

"lint Reference," in *A/UX Programming Languages and Tools, Volume 1*

## NAME

ln — makes links

## SYNOPSIS

ln [-s] *file1* [*file2*]

ln *file... directory*

ln -f *directory1 directory2*

## ARGUMENTS

-f Causes ln to make a hard link to an existing directory. Only the superuser is permitted to use this option.

*directory*
Specifies the directory to which the *file* is linked.

*directory1*
Specifies the directory that will be hard linked to *directory2*.

*directory2*
Specifies the directory that will be hard linked to *directory1*.

*file* Specifies the file that will be linked to the current directory (*directory*).

*file1*
Specifies the file that will be symbolically linked to file2.

*file2*
Specifies the file that will be symbolically linked to *file1*.

-s Causes ln to create symbolic links.

## DESCRIPTION

A link is a directory entry referring to a file; the same file (together with its size, all its protection information, and so forth) may have several links to it.

There are two kinds of links: hard links and symbolic links. By default ln makes hard links. A hard link to a file is indistinguishable from the original directory entry; any changes to a file are effective, independent of the name used to reference the file. Hard links may not span file systems and (unless created with the -f option by the superuser) may not refer to directories.

A symbolic link contains the name of the file to which it is linked. The referenced file is used when an open operation is performed on the link. A stat on a symbolic link will return the linked-to file; an lstat must be done to obtain information about the link. The readlink call may be used to read the contents of a symbolic link. Symbolic links may span file systems and may refer to directories.

The `ln` command may be invoked with one, two, or more than two arguments. If given one argument, `ln` creates a link in the current directory to *file1*. The file named by *file1* must not already exist in the current directory, or `ln` will exit with the message *file1*: `File exists`.

Given two arguments, `ln` creates a link to an existing file *file1* having the name *file2*. The argument *file2* may also be a directory in which to place the link. If only the directory is specified, the link will be made to the last component of *file1*. If *file1* is not found, `ln` will so indicate and no link will be created. If *file2* already exists, it will not be overwritten.

Given more than two arguments, `ln` makes links to all the named files in the named directory. The links made will have the same name as the files being linked to.

Any files or directories located in *directory1* will also be found in *directory2*. Moreover, new files created in either directory will appear in the other.

**FILES**

`/bin/ln`
Executable file

**SEE ALSO**

`cp`(1), `mv`(1), `rm`(1)

`link`(2), `stat`(2), `readlink`(2), `stat`(2), `symlink`(2) in *A/UX Programmer's Reference*

## NAME

login — signs you on a terminal session

## SYNOPSIS

login [*name* [*env-var...*]]

## ARGUMENTS

*env-var*

Specifies the environment variable you wish to add to the default "environment." This option may take either the form *xxx* or *xxx=yyy*. If this option is used without an equal sign, the variable is placed in the environment as

L*n*=xxx

where *n* is a number starting at 0 and is incremented each time a new variable name is required. Variable definitions containing an = are placed into the environment without modification. If they already appear in the environment, then they replace the older value.

*name*

Specifies the name of the person who is logging in to the system.

## DESCRIPTION

login is used at the beginning of each terminal session and allows you to identify yourself to the system. It may be invoked as a command or by the system when a connection is first established. Also, it is invoked by the system when a previous user has terminated the initial shell by typing a CONTROL-D to indicate an "end-of-file".

If login is invoked as a command, it must replace the initial command interpreter. This is accomplished by typing

```
exec login
```

from the initial shell, if it is the Bourne shell, sh(1). For the C shell, csh(1), and the Korn shell, ksh(1), you may just type:

```
login [user]
```

The login command asks for your user name (if not supplied as an argument), and, if appropriate, your password. Echoing is turned off (when possible) during the typing of your password, so it will not appear on the written record of the session.

At some installations, an option may be invoked that will require you to enter a second dialup password. This will occur only for dialup connections, and will be prompted by the message:

```
dialup password:
```

Both passwords are required for a successful login.

If you do not complete the login successfully within a certain period of time (for example, one minute), you are likely to be disconnected silently. Note that login does a sleep to settle the line and waits for a few seconds before accepting your input. If it misses the first character of your input, type it slower.

After a successful login, accounting files are updated, the procedure /etc/profile is performed for users whose login shell is either sh or ksh, and the message-of-the-day, if any, is printed. Then, the user ID, the group ID, the working directory, and the command interpreter are initialized, according to specifications found in the /etc/passwd file entry for the user. If the command interpreter is sh, the file .profile, if it exists, in the initial working directory is executed. To indicate that this invocation of the command interpreter is the login shell, the name of the interpreter is prefixed with a minus sign (-), (for example, -sh). If the last field in the password file is empty, then the default command interpreter, the Bourne shell (/bin/sh) is used. If the last field is *, then a chroot is done to the directory named in the directory field of the entry. At that point login is re-executed at the new level, which must have its own root structure, including /etc/login and /etc/passwd.

The basic ''environment'' (see environ(5)) is initialized to

HOME=*your-login-directory*
PATH=:/bin:/usr/bin
SHELL=*last-field-of-passwd-entry*
MAIL=/usr/mail/*your-login-name*
TZ=*timezone-specification*

The environment may be expanded or modified by supplying additional arguments to login, either at execution time or when login requests your login name. The login command will not change the variables PATH and SHELL in order to prevent users from spawning secondary shells with fewer security restrictions. Both login and getty understand simple single-character quoting conventions. Typing a backslash in front of a character quotes it and allows the inclusion of such things as spaces and tabs.

**EXAMPLES**

At the beginning of each terminal session, the following sort of message is displayed on the screen

```
Apple Computer A/UX

login:
```

to which a user name is the appropriate response.

**STATUS MESSAGES AND VALUES**

`Login incorrect`
: If the user name or the password cannot be matched.

`No shell`
`cannot open password file`
`no directory`
: Consult a system administrator.

`No utmp entry.`
`You must exec login from the`
: If you attempted to execute `login` as a command without using the shell's `exec` internal command (`sh`(1) only) or from other than the login shell (`sh`(1) and `ksh`(1)).

**FILES**

`/bin/login`
: Executable file

`/etc/utmp`
: Accounting file

`/etc/wtmp`
: Accounting file

`/etc/motd`
: File containing message-of-the-day entries

`/etc/passwd`
: Password file

`/etc/profile`
: Systemwide personal profile files for (`sh`(1) and `ksh`(1))

`/etc/cshrc`
: Systemwide personal `csh` startup file for (`csh`(1))

`$HOME/.profile`
: Personal profile file for (`sh`(1) and `ksh`(1))

`$HOME/.login`
: Personal file for `csh` startup used at login time (`csh`(1))

`$HOME/.cshrc`
: Personal `csh` startup file for (`csh`(1))

`$HOME/.logout`
: Personal *csh* logout file used at logout time for (`csh`(1))

`/usr/mail/`*name*
: Mailbox file for user *name*

**SEE ALSO**

`csh`(1), `ksh`(1), `mail`(1), `newgrp`(1), `rlogin`(1N), `sh`(1), `su`(1)
`getty`(1M), `init`(1M) in *A/UX System Administrator's Reference*

passwd(4), profile(4), environ(5) in *A/UX Programmer's Reference*

*A/UX Essentials*

*A/UX Shells and Shell Programming*

**NAME**

logname — gets the login name

**SYNOPSIS**

logname

**DESCRIPTION**

logname returns the contents of the environment variable $LOGNAME, which is set when a user logs into the system.

**EXAMPLES**

The command:

```
logname
```

displays the $LOGNAME of the user logged into the system on the current port.

**FILES**

/bin/logname
: Executable file

/etc/profile
: File containing the user's login profile

**SEE ALSO**

env(1), login(1), printenv(1)

logname(3X), environ(5) in *A/UX Programmer's Reference*

## NAME

lookbib — finds references in a bibliography

## SYNOPSIS

lookbib [-n] *database*

## ARGUMENTS

*database*
Specifies the database to be searched.

-n Turns off the prompt for instructions.

## DESCRIPTION

lookbib uses an inverted index made by indxbib(1) to find sets of bibliographic references. A bibliographic reference is a set of lines, constituting fields of bibliographic information. Each field starts on a line beginning with a %, followed by a key-letter, then a blank, and finally the contents of the field, which may continue until the next line starting with %.

The lookbib command reads keywords typed after the > prompt on the terminal and retrieves records containing all these keywords. If nothing matches, nothing is returned except another > prompt.

The lookbib command will ask if you need instructions and will print some brief information if you reply y.

It is possible to search multiple databases, as long as they have a common index made by indxbib. In that case, only the first argument given to indxbib is specified to lookbib.

If lookbib does not find the index files (the .i[abc] files), it looks for a reference file with the same name as the argument, without the suffixes. It creates a file with a .ig suffix, suitable for use with fgrep. It then uses this fgrep file to find references. This method is simpler to use, but the .ig file is slower to use than the .i[abc] files, and does not allow the use of multiple reference files.

## FILES

/usr/ucb/lookbib
Executable file

*file*.ia
Output file where *file* is the name of the index

*file*.ib
Output file where *file* is the name of the index

*file*.ic
Output file where *file* is the name of the index

*file*.ig
Output file where *file* is the name of the index

**SEE ALSO**

addbib(1), indxbib(1), refer(1), roffbib(1), sortbib(1)

**NAME**

lorder — finds the ordering relation for an object library

**SYNOPSIS**

lorder *file...*

**ARGUMENTS**

*file* Specifies the object or library archive file.

**DESCRIPTION**

lorder produces a global cross-reference, given a list of object modules (.ofiles), which can then be passed to tsort to produce a properly ordered archive file. The input is one or more object or library archive *files* (see ar(1)). The standard output is a list of pairs of object filenames, meaning that the first file of the pair refers to external identifiers defined in the second. The output may be processed by tsort to find an ordering of a library suitable for one-pass access by ld .

> *Note:* The link editor ld is capable of multiple passes over an archive in the portable archive format (see ar) and does not require that lorder be used when building an archive.

Use of the lorder command may, however, allow for a slightly more efficient access of the archive during the link edit process.

**EXAMPLES**

The command:

```
ar cr library 'lorder *.o | tsort'
```

builds a new library from existing .o files.

**LIMITATIONS**

Object files whose names do not end with .o, even when contained in library archives, are overlooked. Their global symbols and references are attributed to some other file.

**FILES**

/bin/lorder
: Executable file

*symref
: Executable file

*symdef
: Executable file

**SEE ALSO**

ar(1), ld(1), tsort(1)

ar(4) in *A/UX Programmer's Reference*

**NAME**

lp — spools print requests to printers

**SYNOPSIS**

lp [-c] [-d*dest*] [-m] [-n*number*] [-o*option*] [-s] [-t*title*] [-w] [*file*]...

**ARGUMENTS**

-c Makes copies of the files specified by *files* immediately after you enter the lp command. Normally, the system doesn't copy files, but links files whenever possible. If you don't include this option, be careful not to remove any of the files being printed until all printing is complete. Also note that without this option, any changes you make to the files after you enter the lp command and before printing is complete will appear in the printed output.

-d*dest*
Specifies the printer or class of printers to use when printing particular jobs. If *dest* specifies a printer, then the system uses that specific printer. If *dest* specifies a class of printers, then the system prints on the first available printer that is a member of the class. Under certain conditions (printer unavailability, insufficient file space, and so forth), the system can not accept requests for specific destinations (see accept(1M) and lpstat(1)). By default, the system uses *dest* from the environment variable LPDEST (if it is set). Otherwise, lp goes to the default destination (if one exists) for the system you are using. Destination names vary between systems (see lpstat(1)).

*file* Specifies the file to be spooled.

-m Sends mail by means of mail(1) after the files have been printed. By default, the system sends no mail upon normal completion of the print request.

-n*number*
Specifies the number of copies to be printed for particular jobs. The default is 1.

-o*option*
Specifies the printer-dependent or class-dependent options. You can specify multiple options by using the key character -o more than once. For more information about valid options, see "Models" in lpadmin(1M).

-s Suppresses messages from lp such as request id is *id*.

-t*title*
Specifies the title that prints on the banner page for particular jobs.

-w Writes a message on your terminal after the the system prints the files. If you aren't logged on at the time the message is written, the system

sends mail instead.

**DESCRIPTION**

`lp` spools the named files (or standard input if it is used at the end of a pipe) for printing.

If you don't include filenames, the `lp` command waits to receive text data typed on the standard input followed by an end-of-file character. You can also use a hyphen (`-`) on the command line (with or without filenames) to represent the standard input, which could be a pipe rather than text typed at the keyboard. (See the ''Limitations''section later in this manual page.) The `lp` command prints the files (including the standard input) in the same order as that in which they appear on the command line.

The `lp` command associates a unique job identification number with each request and displays that number as part of its status message (sent to the standard output). You can use this ID to stop a job which is printing or is scheduled to print (see `cancel`(1)).

**LIMITATIONS**

Any files specified must be readable by the `lp` user account because `/usr/bin/lp` changes the effective user ID to `lp`. If the file permissions assigned to *files* don't allow `lp` to read them, you must use a pipe to direct the files to `lp`, as shown here. (Besides `cat`, other frequently used printer-formatting utilities are `pr` and `troff`.)

`cat` *files* | `lp`

**FILES**

`/usr/bin/lp`
Executable file

`/usr/spool/lp/*`
Print job information files

**SEE ALSO**

`cancel`(1), `enable`(1), `lpq`(1), `lpr`(1), `lpstat`(1), `mail`(1)

`accept`(1M), `lpadmin`(1M), `lpsched`(1M) in *A/UX System Administrator's Reference*

## NAME

lpq — queries the print spooler for progress information

## SYNOPSIS

lpq [+[*sleep-interval*]] [-l] [-P*printer*] [*jobno*]... [*user*]...

## ARGUMENTS

+[*sleep-interval*]
: Displays the spool queue until the last job has printed. If desired, you can replace *sleep-interval* with the number of seconds lpq should sleep between scans of the queue.

*jobno*
: Limits the query to information about a specific job or several jobs. See the ''Examples'' section later in this manual page.

-l
: Prints additional information about the file or files that have been sent to the print spooler as one print job. Normally, only as much information as fits on one line is displayed. Job ordering depends on the algorithm used to scan the spooling directory and is supposed to be FIFO (First In First Out). The filenames for a job may be unavailable when lpr is used as the last command in a pipeline, in which case the file is identified as standard input.

-P*printer*
: Limits the query to information about jobs that are destined for a particular printer. If this option is not specified, the default line printer (or the value of the PRINTER variable in the environment) is used.

*user*
: Limits the query to information about jobs that belong to the user or users specified.

## DESCRIPTION

lpq responds to your queries about print jobs by examining particular files and directories that are used by lpd. The lpd program normally runs continuously in order to service print requests (see lpr(1)).

By running lpq without any arguments, you can obtain a report describing all the print jobs currently in the queue.

For each print job that remains to be done, lpq reports the user's name, the current rank in the queue, the names of files included in the job, the job identifier, and the total size in bytes. You can also supply the job number as an argument to lprm to remove a job before it is printed (see lprm(1)).

## EXAMPLES

In the following processing request, a job-specific query is made to see the status of job number 286.

```
% lpq 286
cashew is ready and printing
Rank   Owner    Job  Files          Total Size
active root     286  /etc/passwd    742 bytes
% 
```

**STATUS MESSAGES AND VALUES**

Beyond the normal information reported by `lpq` regarding the status of print jobs, certain error messages may also be provided.

If `lpq` warns that no daemon is present because of some malfunction, you can use the `lpc` command to restart the printer daemon (see `lpc`(1M)).

The `lpq` program may also report that it is unable to open various files.

**LIMITATIONS**

Because of the dynamic nature of the information in the spooling directory, `lpq` may not reliably report information concerning newly arriving or newly dispatched print jobs.

Output formatting depends upon the line length of the terminal. The line length can result in widely spaced columns.

**FILES**

`/etc/printcap`
: File containing printer capabilities

`/etc/termcap`
: File containing terminal capabilities

`/usr/spool/*`
: Directory used by a variety of spooling utilities for configuration and data files

`/usr/spool/*/cf*`
: Control files specifying jobs

`/usr/spool/*/lock`
: Printer lock files

**SEE ALSO**

`lpr`(1), `lprm`(1)

`lpc`(1M), `lpd`(1M) in *A/UX System Administrator's Reference*

**NAME**

`lpr` — spools print requests to printers

**SYNOPSIS**

`lpr` [`-#` *copies*] [`-C` *class*] [`-h`] [`-i` [*indent-cols*]] [`-J` *cover-title*] [`-l`] [`-m`] [`-p`] [`-P` *printer*] [`-r`] [`-s`] [`-T` *title*] [`-w`*page-width*] [*file*]...

**ARGUMENTS**

`-#`*copies*
: Specifies the number of copies to be printed of each of the named files.

`-C` *class*
: Specifies a particular class of print job for routing to a particular class of printers. For example,

```
lpr -C Postscript foo.c
```

causes the file `foo.c` to be sent to a PostScript®-class printer.

*file*
: Specifies the name of the file to be sent to the printer.

`-h`
: Suppresses the printing of the burst page.

`-i` [*indent-cols*]
: Specifies the number of columns each line is indented from the left margin. If no argument is supplied, 8 space characters are printed before each line.

`-J` *cover-title*
: Specifies how the job is identified on the cover page that appears before the print job. If this option is not specified, the name of the first file is used.

`-l`
: Causes `lpr` to use a filter that allows control characters to be printed and suppresses page breaks.

`-m`
: Sends mail upon completion.

`-p`
: Causes `lpr` to use `pr` to format the files (equivalent to `print`).

`-P` *printer*
: Specifies the name of the printer to which the job is sent.

`-r`
: Removes the file upon completion of spooling or upon completion of printing (when used with the `-s` option).

`-s`
: Causes symbolic links to be created in the spooler directories to conserve file-system space. If this option is not specified, files to be printed are copied into the spooler directories. Be careful not to modify or remove the files submitted for printing until they have completed printing.

-T *title*
Specifies how the job is identified on the header portion of each page of the print job. This option requires the use of the -p option, which invokes pr; together these options operate like the -h option of the pr command.

-w Specifies the page width in columns. This number is used by the pr command. To invoke pr, you must also supply the -p option. Together these options operate like the -w option of the pr command.

## DESCRIPTION

lpr uses a spooling daemon to print the named files when facilities become available. If no files appear, the standard input is assumed.

You can used the -P option to force output to a specific printer. Normally, the default printer is used (which is site dependent), or the value of the environment variable PRINTER is used.

## EXAMPLES

To print three copies of the file foo.c, followed by three copies of the file bar.c, followed by three copies of more.c, enter

```
lpr -#3 foo.c bar.c more.c
```

To obtain three copies of the combined text of the same three files, enter

```
cat foo.c bar.c more.c | lpr -#3
```

## LIMITATIONS

If you try to spool a file that is too large, it will be truncated.

The lpr command can not be used to print files containing binary codes.

## STATUS MESSAGES AND VALUES

Error messages will be produced if lpr finds binary files among the files to be printed.

If a user other than root prints a file and spooling is disabled, lpr prints a message saying so and does not put jobs in the queue.

If a connection to lpd on the local computer cannot be made, lpr prints a message saying that the daemon cannot be started.

Diagnostics may be printed in the daemon's log file, regarding missing spool files by lpd.

## FILES

/etc/passwd
Personal identification file

/etc/printcap
Printer-capabilities database file

`/usr/lib/lpd*`
Files containing line printer daemons
`/usr/spool/*`
Directories used for spooling
`/usr/spool/*/cf*`
Daemon control files
`/usr/spool/*/df*`
Data files specified in `cf*` files
`/usr/spool/*/tf*`
Temporary copies of `cf*` files

**SEE ALSO**

`lpq`(1), `lprm`(1), `pr`(1)

`lpc`(1M), `lpd`(1M) in *A/UX System Administrator's Reference*

`symlink`(2), `printcap`(4) in *A/UX Programmer's Reference*

## NAME

lprm — removes jobs from the line printer spooling queue for a Berkeley file system (4.2)

## SYNOPSIS

lprm [-P*printer*] [-] [*jobno*]... [*user*]...

## ARGUMENTS

-
: Removes all jobs that a user owns. If the superuser employs this option, the spool queue is emptied entirely. The owner is determined by the user's login name and host name on the machine where the lpr command was invoked.

*jobno*
: Specifies the job number that is to be removed.

-P*printer*
: Specifies the queue associated with a specific printer; otherwise the default printer, or the value of the PRINTER variable in the environment is used.

*user*
: Specifies the name of the user who owns the job that is being removed.

## DESCRIPTION

lprm removes a job, or jobs, from a printer spool queue. Since the spooling directory is protected from users, using lprm is normally the only method by which a user may remove a job.

The lprm command without any arguments deletes the currently active job if it is owned by the user who invoked lprm.

Specifying a user's name or list of users' names causes lprm to attempt to remove any queued jobs belonging to that user (or users). This form of invoking lprm is useful only to the super-user.

A user may remove an individual job from a queue by specifying its job number. This number may be obtained from the lpq(1) program, for example,

```
% lpq -l

ken : 1st          [job 013ucbarpa]
(standard input)   100 bytes
% lprm 13
```

The lprm command announces the names of any files it removes and is silent if there are no jobs in the queue that match the request list.

The lprm command kills off an active daemon, if necessary, before removing any spooling files. If a daemon is killed, a new one is automatically restarted upon completion of file removals.

**STATUS MESSAGES AND VALUES**

A ''Permission denied'' is received if the user tries to remove files other than his own.

**LIMITATIONS**

Since there are race conditions possible in the update of the lock file, the currently active job may be incorrectly identified.

**FILES**

/etc/printcap
: Printer characteristics file

/usr/spool/*
: Spooling directories

/usr/spool/*/lock
: Lock file used to obtain the process ID of the current daemon and the job number of the currently active job

**SEE ALSO**

lpr(1), lpq(1)

lpd(1M) in *A/UX System Administrator's Reference*

## NAME

lpstat — prints lp status information

## SYNOPSIS

lpstat [-a[*list*]] [-c[*list*]] [-d] [-o[*list*]] [-p[*list*]] [-r] [-s] [-t] [-u[*list*]] [-v[*list*]]

## ARGUMENTS

-a[*list*]
: Prints acceptance status (with respect to lp) of destinations for requests. Replace *list* with a list of intermixed printer names and class names.

-c[*list*]
: Prints the class names and their members. Replace *list* with a list of class names.

-d
: Prints the system default destination for lp.

-o[*list*]
: Prints the status of output requests. Replace *list* with a list of intermixed printer names, class names, and request IDs.

-p[*list*]
: Prints the status of printers. Replace *list* with a list of printer names.

-r
: Prints the status of the lp request scheduler.

-s
: Prints a status summary, including the status of the line printer scheduler, the system default destination, a list of class names and their members, and a list of printers and their associated devices.

-t
: Prints all status information.

-u[*list*]
: Prints status of output requests for users. Replace *list* with a list of login names.

-v[*list*]
: Prints the names of printers and the pathnames of the devices associated with them. Replace *list* with a list of printer names.

## DESCRIPTION

lpstat prints information about the current status of the lp line printer system.

If no options are given, then lpstat prints the status of all requests made to lp by the user. Any arguments that are not options are assumed to be request IDs (as returned by lp). The lpstat command prints the status of such requests.

Some of the options may be followed by an optional *list* that can be in one of two forms: a list of items separated from one another by a comma, or a list of items enclosed in double quotes and separated from one another by a comma and/or one or more spaces. For example:

```
-u user1, user2, user3
```

The omission of a *list* following such options causes all information relevant to the options to be printed, for example:

```
lpstat -o
```

prints the status of all output requests.

**FILES**

`/usr/bin/lpstat`
Executable file

`/usr/spool/lp/*`
Spooler files

**SEE ALSO**

`enable`(1), `lp`(1), `lpq`(1)

**NAME**

ls — lists the contents of a directory

**SYNOPSIS**

ls [-a] [-b] [-c] [-C] [-d] [-F] [-g] [-i] [-l] [-L] [-m] [-n] [-o] [-p] [-q] [-r] [-R] [-s] [-t] [-u] [-x] [*names*]

**ARGUMENTS**

-a Lists all entries. Usually entries whose names begin with a period (.) are not listed.

-b Forces printing of nongraphic characters to be in the octal *ddd* notation.

-c Uses the time of the last modification of the i-node (file created, mode changed, and so forth) for sorting (-t) or printing (-l).

-C Specifies multicolumn output with entries sorted vertically.

-d Lists the directory name only, not its contents. This option is often used with the -l option to get the status of a directory. This option does not apply if the a file is specified.

-F Puts a slash (/) after each filename if that file is a directory, an asterisk (*) after each filename if that file is executable, and an (@) after each filename if that file is a symbolic link.

-g Acts the same as the -l option except that the owner is not printed.

-i Prints the i-number in the first column of the report, for each file.

-l Lists in long format, giving mode, number of links, owner, group, size in bytes, and time of last modification for each file (see below). If the file is a special file, the size field will contain the major and minor device numbers, instead of a size. If the file is a symbolic link, the pathname of the linked-to file is printed preceded by ->.

-L Lists the file's or directory's (if it is a symbolic link) link references rather than the link itself.

-m Specifies stream output format.

-n Acts the same as the -l option, except that the owner's user ID and group's group ID numbers, rather than the associated character strings, are printed.

*names*
Specifies the files or directories to be listed.

-o Acts the same as the -l option except that the group is not printed.

-p Puts a slash (/) after each filename if that file is a directory.

`-q` Forces printing of nongraphic characters in filenames as the character (`?`).

`-r` Reverses the order of sort to get reverse alphabetic or oldest first, as appropriate.

`-R` Recursively lists subdirectories encountered.

`-s` Gives size in 512-byte blocks, including indirect blocks, for each entry.

`-t` Sorts by time modified (latest first), instead of by name.

`-u` Uses the time of the last access, instead of the last modification, for sorting (with the `-t` option) or printing (with the `-l` option).

`-x` Specifies multicolumn output with entries sorted horizontally, rather than down the page.

**DESCRIPTION**

For each directory argument, `ls` lists the contents of the directory; for each file argument, `ls` repeats the filename and any other information requested. The output is sorted alphabetically by default. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but with file arguments appearing before directory arguments and their contents.

There are three major listing formats. The default format is to list one entry per line, the `-C` and `-x` options enable multicolumn formats, and the `-m` option enables stream output format, in which files are listed across the page, separated by commas. In order to determine output formats for the `-C`, `-x`, and `-m` options, `ls` uses an environment variable, `COLUMNS`, to determine the number of character positions available on one output line. If this variable is not set, the `terminfo` database is used to determine the number of columns, based on the environment variable `TERM`. If this information cannot be obtained, 80 columns are assumed.

The mode printed under the `-l` option consists of 10 characters that are interpreted below. The first character can be one of the following:

`d` if the entry is a directory

`b` if the entry is a block special file

`c` if the entry is a character special file

`l` if the entry is a symbolic link

`p` if the entry is a fifo (named pipe) special file

`-` if the entry is an ordinary file

The next 9 characters are interpreted as three sets of three bits each. The first set refers to the owner's permissions; the next to permissions of others in the user-group of the file; and the last to all others. Within each set, the three characters indicate permission to read, to write, and to execute the file as a program, respectively. For a directory, ''execute'' permission is interpreted to mean permission to search the directory for a specified file.

The permissions are indicated as follows:

`r` if the file is readable;

`w` if the file is writable;

`x` if the file is executable;

`-` if the indicated permission is not granted.

The group-execute permission character is given as `s` if the file has set-group-ID mode; likewise, the user-execute permission character is given as `s` if the file has set-user-ID mode. The last character of the mode (normally `x` or `-`) is `t` if the 1000 (octal) bit of the mode is on; see `chmod`(1) for the meaning of this mode. The indications of set-ID and 1000 bits of the mode are capitalized (`S` and `T`, respectively) if the corresponding execute permission is not set.

When the sizes of the files in a directory are listed, a total count of blocks, including indirect blocks, is printed.

**EXAMPLES**

The command:

```
ls -l /etc
```

will list all entries in `/etc` in long format, as, for example,

```
-rw-r--r- 1 root bin  115  Mar 17 1986  mtab
```

where the fields represent the file's permissions, number of links, owner, group, size in bytes, date of last modification, and name, respectively.

**LIMITATIONS**

Unprintable characters in filenames may confuse the columnar output options.

**FILES**

`/bin/ls`
: Executable file

`/etc/passwd`
: File to get user IDs for `ls -l` and `ls -o`

`/etc/group`
: File to get group IDs for `ls -l` and `ls -g`

/usr/lib/terminfo/*
Files to get terminal information

**SEE ALSO**
chown(1), chmod(1), find(1)

## NAME

m4 — processes macros for C and other languages

## SYNOPSIS

m4 [-B*int*] [-e] [-H*int*] [-s] [-S*int*] [-T*int*] [-D*name*[=*val*]] [-U*name*] [*file*]...

## ARGUMENTS

-B*int*
: Changes the size of the push-back and argument collection buffers from the default of 4096.

-D*name*[=*val*]
: Defines *name* to *val* or to null in the absense of *val*.

-e
: Causes m4 to operate interactively. Interrupts are ignored and the output is unbuffered.

*file*
: Specifies the file to be processed. If this argument is not specified, or if a dash (-) is used as the filename, the standard input is read.

-H*int*
: Changes the size of the symbol table hash array from the default of 199. The size should be prime.

-s
: Enables line sync output for the C preprocessor (#line...).

-S*int*
: Changes the size of the call stack from the default of 100 slots. Macros take 3 slots, and nonmacro arguments take 1.

-T*int*
: Changes the size of the token buffer from the default of 512 bytes.

-U*name*
: Undefines *name*.

## DESCRIPTION

m4 is a macro processor intended as a front end for C and other languages. Each of the argument files is processed in order. The processed text is written on the standard output.

To be effective, the options must appear before any filenames and before any -D or -U options.

Macro calls have the form:

*name*(*arg1*,*arg2*,...,*argn*)

The right parenthesis, (, must immediately follow the name of the macro. If the name of a defined macro is not followed by a (, it is deemed to be a call of that macro with no arguments. Potential macro names consist of alphabetic letters, digits, and underscore (_), where the first character is not

a digit.

Leading unquoted blanks, tabs, and newlines are ignored while collecting arguments. Left and right single quotes are used to quote strings. The value of a quoted string is the string stripped of the quotes.

When a macro name is recognized, its arguments are collected by searching for a matching right parenthesis. If fewer arguments are supplied than are in the macro definition, the trailing arguments are taken to be null. Macro evaluation proceeds normally during the collection of the arguments, and any commas or right parentheses that happen to turn up within the value of a nested call are as effective as those in the original input text. After the argument collection, the value of the macro is pushed back onto the input stream and rescanned.

**Built-in macros**

The `m4` program makes available the following built-in macros. They may be redefined, but once this is done the original meaning is lost. Their values are null unless otherwise stated.

`define`
: Installs the second argument as the value of the macro whose name is the first argument. Each occurrence of `$`*n* in the replacement text, where *n* is a digit, is replaced by the *n*th argument. Argument 0 is the name of the macro; missing arguments are replaced by the null string; `$#` is replaced by the number of arguments; `$*` is replaced by a list of all the arguments separated by commas; `$@` is like `$*`, but each argument is quoted (with the current quotes).

`undefine`
: Removes the definition of the macro named in the argument.

`defn`
: Returns the quoted definition of the argument(s). This macro is useful for renaming macros, especially built-in macros.

`pushdef`
: Acts similarly to `define`, but also saves any previous definition.

`popdef`
: Removes the current definition of the argument(s), exposing the previous one, if any.

`ifdef`
: Installs the second argument as its value, if the first argument is defined; otherwise, install the third argument. If there is no third argument, the value is null. The word `unix` is predefined on the UNIX® system versions of `m4`.

`shift`
Returns all but the first argument. The other arguments are quoted and pushed back with commas in between. The quoting nullifies the effect of the extra scan that is subsequently performed.

`changequote`
Changes quote symbols to the first and second arguments. The symbols may be up to five characters long. `changequote` without arguments restores the original values, that is, ``''.

`changecom`
Changes left and right comment markers from the default # and newline. With no arguments, the comment mechanism is effectively disabled. With one argument, the left marker becomes the argument and the right marker becomes newline. With two arguments, both markers are affected. Comment markers may be up to five characters long.

`divert`
Changes the current output stream to its (digit-string) argument. `m4` maintains 10 output streams, numbered 0-9. The final output is the concatenation of the streams in numerical order; initially stream 0 is the current stream. Output diverted to a stream other than 0 through 9 is discarded.

`undivert`
Causes immediate output of text from diversions named as arguments, or all diversions if there is no argument. Text may be undiverted into another diversion. Undiverting discards the diverted text.

`divnum`
Returns the value of the current output stream.

`dnl`
Reads and discards characters up to and including the next newline.

`ifelse`
Provides three or more arguments. If the first argument is the same string as the second, then the value is the third argument. If not, and if there are more than four arguments, the process is repeated with arguments 4, 5, 6, and 7. Otherwise, the value is either the fourth string or, if it is not present, null.

`incr`
Returns the value of the argument incremented by 1. The value of the argument is calculated by interpreting an initial digit-string as a decimal number.

`decr`
Returns the value of the argument decremented by 1.

`eval`
Evaluates the argument as an arithmetic expression, using 32-bit arithmetic. Operators include +, -, *, /, %, ^ (exponentiation), bitwise &, |, ^, and ~ as well as relationals and parentheses. Octal and hexidecimal numbers may be specified as in C. The second argument specifies the radix for the result; the default is 10. The third argument may be used to specify the minimum number of digits in the result.

`len`
Returns the number of characters in the argument.

`index`
Returns the position in the first argument where the second argument begins (zero-origin), or -1 if the second argument does not occur.

`substr`
Returns a substring of its first argument. The second argument is a zero-origin number selecting the first character; the third argument indicates the length of the substring. A missing third argument is taken to be large enough to extend to the end of the first string.

`translit`
Transliterates the characters in the first argument from the set given by the second argument to the set given by the third. No abbreviations are permitted.

`include`
Returns the contents of the file named in the argument.

`sinclude`
Acts the same as `include`, except that nothing is returned if the file is inaccessible.

`syscmd`
Executes the system command given in the first argument. No value is returned.

`sysval`
Specifies the return code from the last call to `syscmd`.

`maketemp`
Fills in a string of `XXXXX` in the argument with the current process ID.

`m4exit`
Causes immediate exit from `m4`. Argument 1, if given, is the exit code. The default is 0.

`m4wrap`
Pushes back argument 1 at final EOF. An example is: `m4wrap('cleanup()')`.

`errprint`
Prints the argument on the diagnostic output file.

`dumpdef`
Prints current names and definitions for the named items, or for all items if no arguments are given.

`traceon`
Turns on tracing for all macros including built-ins, with no arguments. Otherwise, turns on tracing for named macros.

`traceoff`
Turns off tracing globally and for any macros specified. Macros specifically traced by `traceon` can be untraced only by specific calls to `traceoff`.

**EXAMPLES**

The command:

```
m4 file1 file2 > outputfile
```

runs the `m4` macro processor on the files `file1` and `file2` and redirects the output into `outputfile`.

**FILES**

`/usr/bin/m4`
Executable file

**SEE ALSO**

`cc`(1), `cpp`(1)

"`m4` Reference," in *A/UX Programming Languages and Tools, Volume 2*

*See* machid(1)

## NAME

m68k, pdp11, u3b, u3b2, u3b5, u3b15, vax — provide truth values about processor type

## SYNOPSIS

```
m68k

pdp11

u3b

u3b2

u3b5

u3b15

vax
```

## DESCRIPTION

The following commands (corresponding to programs) will return a true value (exit code of 0) if program currently runs on a processor that is indicated by the command name.

m68k
: True if program currently runs on a 680x0.

pdp11
: True if program currently runs on a PDP-11/45 or PDP-11/70.

u3b
: True if program currently runs on a 3B 20S.

u3b2
: True if program currently runs on a 3B 2 computer.

u3b5
: True if program currently runs on a 3B 5 computer.

u3b15
: True if program currently runs on a 3B 15 computer.

vax
: True if program currently runs on a VAX-11/750 or VAX-11/780.

The commands that do not apply will return a false (nonzero) value. These commands are often used within make(1) makefiles and shell procedures to increase portability.

## FILES

/bin/m68k
: Executable file

/bin/pdp11
: Executable file

`/bin/u3b`
Executable file
`/bin/u3b2`
Executable file
`/bin/u3b5`
Executable file
`/bin/u3b15`
Executable file
`/bin/vax`
Executable file

**SEE ALSO**

`csh`(1), `ksh`(1), `make`(1), `sh`(1), `test`(1), `true`(1)

**NAME**

macref — produces a cross-reference listing of macro files

**SYNOPSIS**

macref [-n] [-s] [-t] [--] *file...*

**ARGUMENTS**

-- Delimits the end of options.

*file* Specifies the macro file.

-n Causes one line to be printed for each reference to a symbol.

-s Causes symbol-use statistics to be printed.

-t Causes a macro table of contents to be printed.

**DESCRIPTION**

macref reads the named files (which are assumed to be nroff(1)/troff(1) input) and produces a cross-reference listing of the symbols in the input.

The options may be grouped behind one dash (-). The macref program does not accept - as standard input.

The default output is a list of the symbols found in the input, each accompanied by a list of all references to that symbol. The macref command lists the symbols alphabetically in the leftmost column, with the references following to the right. Each reference is given in the form:

[ [ (*NMname*)] *Mname*—] *type lnum* [#]

where the fields have the following meanings:

*Mname*
Specifies the name of the macro within which the reference occurs. This field is missing if the reference occurs outside a macro. Any names listed in the *NMname* part are macros within which *Mname* is defined.

*type* Specifies the type associated, by context, with this occurrence of the symbol. The types may be:

r request

m macro

d diversion

s string

n number register

p parameter (for example, \$*x* is a parameter reference to *x*. Note that parameters are never modified, and that the only valid

parameter symbol names are 1, 2, ... 9).

*lnum*
Specifies the line number on which the reference occurred.

*#* Modifies the value of the symbol.

Generated names are listed under the artificial symbol name `~sym`.

**FILES**

`/usr/bin/macref`
Executable file

**SEE ALSO**

`nroff`(1), `troff`(1)

## NAME

mactoiso, isotomac — convert between Macintosh encoding and International Standards Organization (ISO) encoding

## SYNOPSIS

mactoiso [-c *char*] [*file*]

isotomac [-c *char*] [*file*]

## ARGUMENTS

-c *char*
: Specifies the use of a character other than the default (blank).

*file* Specifies a file from which input is to be read.

## DESCRIPTION

mactoiso reads, from the file specified by the optional *file* argument, or from standard input, characters encoded according to the Macintosh character set, converts them to the ISO 8859-1 character-set-encoding scheme, and writes them to standard output.

isotomac reads characters from the ISO character set as input, converts them to the Macintosh character set, and writes them to standard output.

Each character set contains characters that are not represented in the other character set. By default, the isotomac and mactoiso commands place a blank character in place of any character that is not represented in the character set to which they are converting the file.

## STATUS MESSAGES AND VALUES

The exit status is 0 upon successful completion, and 1 for a usage error.

## SEE ALSO

charcvt(3C), iso(5) mac(5), in *A/UX Programmer's Reference*

## NAME

mail — send mail to users or read mail

## SYNOPSIS

mail [-e] [-f*file*] [-p] [-q] [-r] [-t] *address...*

## ARGUMENTS

*address*
Specifies the address where the mail is to be sent.

-e Causes mail not to be printed. An exit value of 0 is returned if the user has mail; otherwise, an exit value of 1 is returned.

-f*file*
Causes mail to use *file* (e.g., mbox) instead of the default "mailfile."

-p Causes all mail to be printed without prompting for disposition.

-q Causes mail to terminate after interrupts. Normally an interrupt only causes the termination of the message being printed.

-r Causes messages to be printed in first-in, first-out order.

-t Causes the message to be preceded by all *address*es the mail is sent to. An *address* is usually a user name recognized by login(1). If an *address* being sent mail is not recognized, or if mail is interrupted during input, the file dead.letter will be saved to allow editing and resending. Note that this is regarded as a temporary file in that it is recreated every time needed, erasing the previous contents of dead.letter.

## DESCRIPTION

mail without arguments, prints a user's mail, message-by-message, in last-in, first-out order. For each message, the user is prompted with a ?, and a line is read from the standard input to determine the disposition of the message:

*newline*
Goes on to next message.

\+ Acts the same as newline.

d Deletes the message and goes on to next message.

p Prints the message again.

\- Goes back to the previous message.

s [*file*]...
Saves the message in the named file (mbox is default).

`w` [*file*]...
Saves the message, without its header, in the named file (`mbox` is default).

`m` [*address*]...
Mails the message to the named *address*es (yourself is default).

`q` Puts undeleted mail back in the mailfile and stops.

EOT (CONTROL-d)
Acts the same as the `q` option.

`x` Puts all the mail back in the mailfile unchanged and stops.

`!`*command*
Escapes to the shell to perform *command*.

`*` Prints a command summary.

When *address* is named, `mail` takes the standard input up to an end-of-file (or up to a line consisting of just a `.`) and adds it to each *address*'s mailfile. The message is preceded by the sender's name and a postmark. Lines that look like postmarks in the message, (i.e., `From...`) are preceded with a `>`.

To denote a recipient on a remote system, prefix *address* by the system name and exclamation mark (see `uucp`(1C)). Everything after the first exclamation mark in *address* is interpreted by the remote system. In particular, if *address* contains additional exclamation marks, it can denote a sequence of machines through which the message is to be sent on the way to its ultimate destination. For example, specifying `a!b!cde` as a recipient's name causes the message to be sent to user `b!cde` on system `a`. System `a` will interpret that destination as a request to send the message to user `cde` on system `b`. This might be useful, for instance, if the sending system can access system `a` but not system `b`, and system `a` has access to system `b`. `mail` will not use `uucp` if the remote system is the local system name (i.e., `localsystem!user`).

The mailfile may be manipulated in two ways to alter the function of `mail`. The other permissions of the file may be read-write, read-only, or neither read nor write to allow different levels of privacy. If changed to other than the default, the file will be preserved even when empty to perpetuate the desired permissions. The file may also contain the first line:

`Forward to` *address*

which causes all mail sent to the owner of the mailfile to be forwarded to *address*. This is especially useful to forward all of a user's mail to one machine in a multiple machine environment. In order for forwarding to work properly, the mailfile should have `mail` as group ID, and the group permission should be read-write.

When a user logs in, the presence of mail, if any, is indicated. Also, notification is made if new mail arrives while using `mail`.

**EXAMPLES**

The command:

```
mail cj
```

accepts whatever message is typed up to an EOF. The user `cj` will be notified that he has mail the next time he logs in.

If you want to read mail that has been sent to you, simply type

```
mail
```

**LIMITATIONS**

Conditions sometimes result in a failure to remove a lock file.

After an interrupt, the next message may not be printed; printing may be forced by typing a `p`.

**FILES**

`/bin/mail`
: Executable file

`/etc/passwd`
: File containing user addresses

`/usr/mail/`*user*
: File containing incoming mail for *user*; i.e., the mailfile

`$HOME/mbox`
: File containing saved mail

`$MAIL`
: File containing pathname of mailfile

`/tmp/ma*`
: Temporary file

`/usr/mail/*.lock`
: Lock file for mail directory

`$HOME/dead.letter`
: Unmailable text file

**SEE ALSO**

`biff`(1), `login`(1), `mailx`(1), `uucp`(1C), `write`(1)

**NAME**

mailx — enables you to send and receive messages electronically

**SYNOPSIS**

mailx [-d] [-e] [-f [*filename*]] [-F] [-h *number*] [-H] [-i] [-n] [-N] [-r *address*] [-s *subject*] [-u *user*] [-U] [*name*]...

**ARGUMENTS**

-d Turns on debugging output; neither particularly interesting nor recommended.

-e Tests for the presence of mail. The mailx program prints nothing and exits with a successful return code if there is mail to read.

-f [*filename*]
Read messages from *filename* instead of /usr/mail/*login-name*. If no *filename* is specified, the default is used, $HOME/mbox or $MBOX is used. (See "Environment Variables.")

-F Records the message in a file named after the first recipient. Overrides the record variable, if set. (See "Environment Variables.")

-h *number*
Specifies the number of network "hops" made so far. This is provided for network software to avoid infinite delivery loops.

-H Prints the header summary only.

-i Ignores interrupts. See also ignore in "Environment Variables."

-n Does not initialize from the system default /usr/lib/mailx/mailx.rc.

-N Does not print the initial header summary.

*name*
Specifies the login name of the user you wish to send mail to.

The name specification can take several forms: login names, shell commands, or alias groups. Login names may alternately be any network address, including mixed network addressing. If a *name* replacement begins with a pipe symbol (|), the remainder of the name must be a shell command through which the message can be routed. This provides an automatic interface with any program that reads the standard input. For example, |lp can be specified as one of the destination names so that the outgoing mail message is also printed. Alias groups are set by the a[lias] command (see "Commands," later in this section). The alias command establishes lists of recipients of any type.

`-r` *address*
Passes *address* to network delivery software. All tilde commands are disabled.

`-s` *subject*
Sets the Subject header field to *subject*.

`-u` *user*
Reads any incoming mail for a user with the specified login name. This is only effective if the mailbox file associated with the user account is not read-protected.

`-U` Converts `uucp` style addresses to internet standards. Overrides the `conv` environment variable.

**DESCRIPTION**

`mailx` provides a flexible environment for sending and receiving messages electronically. To select a send or receive operating mode, `mailx` examines its command-line arguments. Any arguments supplied that are not command options select a send mode. In send mode, each *name* argument is interpreted as a mail destination (recipient). If no *name* arguments are supplied, `mailx` selects read-incoming-mail mode. The incoming mail messages are read from a mailbox file (see the `-f` option).

When reading mail, `mailx` provides commands to facilitate saving, deleting, and responding to messages. When sending mail, `mailx` allows editing, reviewing, and other modifications of the message as it is entered.

In send mode, you are expected to continue typing in lines once the command line is accepted. Each line you type is treated as a portion of the message you are sending. To specify the end of a mail message, type an end-of-file character, or enter a single period at the start of a new line.

The mail messages sent by you become incoming mail for the specified recipients. Incoming mail messages are stored in a standard file for each user, called the system mailbox for that user, usually named `/usr/mail/`*login-name*. (You may alter this default by using the `-f` option, as shown later.) When a `mailx` command is specified to read messages, the mailbox is the default place `mailx` expects to find them. As messages are read, they are usually marked to be moved to a secondary file for longer-term storage. This secondary file is named `mbox` and is normally located in the user's home directory (see ''Environment Variables,'' later, for a description of this file). Messages remain in this file until you remove them using an A/UX provision other than one of the mail commands (such as the `rm` command or an editor such as `TextEditor`).

When reading mail, mailx expects you to enter mail-handling commands in response to the command prompt (normally the ? character). A summary of the incoming mail messages is displayed before the command prompt. This ''header'' summary can be redisplayed at any time using the appropriate (h) command.

When sending mail, mailx is in input mode. In input mode, the characters you type are treated as part of the outgoing message, unless you force mailx to treat subsequent text as a command by typing the escape character. If no subject is specified on the command line, a prompt for the subject is printed. As the message is typed, mailx will read the message and store it in a temporary file. Commands may be entered by beginning a line with the escape character (tilde (~) by default) followed by a single command letter and optional arguments. See ''Tilde Escapes'' later in this section, for a summary of these commands.

At any time, the behavior of mailx is governed by a set of environment variables. These are flags and valued parameters that are set and cleared via the se[t] and uns[et] commands. See ''Environment Variables,'' later, for a summary of these parameters.

At startup time, mailx reads commands from a system-wide file (/usr/lib/mailx/mailx.rc) to initialize certain parameters, then from a private startup file ($HOME/.mailrc) for personalized variables. Most regular commands are legal inside startup files, the most common use being to set up initial display options and alias lists. The following commands are not legal in the startup file: !, C[opy], e[dit], fo[llowup], F[ollowup], ho[ld], m[ail], pre[serve], r[eply], R[eply], sh[ell], and v[isual]. Any errors in the startup file cause the remaining lines in the file to be ignored.

**Command Interpretation**

If you press RETURN without an accompanying command while in command mode, the first incoming mail message is displayed. Thereafter (or any time after any message has been displayed), pressing RETURN with no accompanying command displays the next message.

Each message is assigned a sequential number, which makes it possible to visit messages in any order, not just newest-to-oldest order. At any time, mailx keeps track of the ''current'' message, which is the message most recently displayed. In a ''message header'' report (see the h command) the current message is preceded by a > symbol in the one-line-per-message summary.

When you do not supply a message number with those interactive commands that can accept a *msglist* argument, such commands operate on the current message.

A *msglist* is one or more message specifications separated by spaces. A message specification may take any of the forms shown in the following list. (Note that the command context affects whether the message specification you supply for *msglist* can be honored or not.)

*number*
Specifies the message number.

. Specifies the current message.

^ Specifies the first undeleted message.

$ Specifies the last message.

* Specifies all messages.

*n-m*
Specifies an inclusive range of message numbers.

*user*
Specifies all messages from *user*.

*/string*
Specifies all messages with *string* in the subject line (case ignored).

*:char*
Specifies all messages of type *char*, where *char* is one of the following:

d Specifies all deleted messages.

n Specifies all new messages.

o Specifies all old messages.

r Specifies all previously read messages.

u Specifies all previously unread messages.

The overall format of the commands is shown here:

*command* [*msglist*] [*arguments*]

The commands and their functions are described in the following subsection.

Other arguments are usually arbitrary strings whose usage depends on the command involved. Filenames, where expected, are expanded via the normal shell conventions (see csh(1)). Special characters are recognized by certain commands and are documented with the commands later.

**Commands**

The following is a complete list of mailx commands:

*!shell-command*
Escapes to the shell. (See SHELL under "Environment Variables.")

# *comment*
Specifies a comment. This may be useful in .mailrc files.

\+ Displays the next message (the same as the n command with no arguments).

\- Displays the previous message.

= Prints the current message number.

? Prints a summary of commands.

a[lias] *alias name...*
g[roup] *alias name...*
Declares an a[lias] or g[roup] for the given names. The names will be substituted when *alias* is used as a recipient. Useful in the .mailrc file.

alt[ernates] *name...*
Declares a list of alternate names for your login. When responding to a message, these names are removed from the list of recipients for the response. With no arguments, alt[ernates] prints the current list of alternate names. (See also allnet under "Environment Variables.")

cd [*directory*]
ch[dir] [*directory*]
Changes directory. If *directory* is not specified, $HOME is used.

c[opy] [filename]
c[opy] [msglist] *filename*
Copies messages to the file without marking the messages as saved. Otherwise equivalent to the s[ave] command.

C[opy] [*msglist*]
Saves the specified messages in a file whose name is derived from the author of the message to be saved, without marking the messages as saved. Otherwise equivalent to the S[ave] command.

d[elete] [*msglist*]
Deletes messages from the mailbox. If autoprint is set, the next message after the last one deleted is printed. (See "Environment Variables.") Otherwise, the next message after the last one deleted is made the current message, but it is not automatically displayed. (So without autoprint, after you use the delete command then press RETURN with no accompanying command, mailx attempts to display the second message beyond the deleted message.)

di[scard] [*header-file*]

ig[nore] [*header-file*]
: Suppresses printing of (discards or ignores) the specified header fields when displaying messages on the screen. Examples of header fields to ignore are status and cc. The fields are included when the message is saved. The P[rint] and T[ype] commands override this command.

dp[*msglist*]
dt[*msglist*]
: Deletes the specified messages from the mailbox and prints the next message after the last one deleted. Roughly equivalent to a d[elete] command followed by a p[rint] command.

ec[ho] *string...*
: Echos the given strings (like echo(1)).

e[dit] [*msglist*]
: Edits the given messages. The messages are placed in a temporary file and the EDITOR variable is used to get the name of the editor (See ''Environment Variables.'') Default editor is ed(1).

ex[it]
x[it]
: Exits from mailx, without changing the mailbox. No messages are saved in the mbox (see also q[uit]).

fi[le] [*filename*]
fold[er] [filename]
: Quits the current file of messages and read-in the specified file (folder). Several special characters are recognized when used as filenames, with the following substitutions:

  % the current mailbox.

  %*user*
  : the mailbox for *user*.

  # the previous file.

  & the current mbox.

  Default file is the current *mailbox*.

folders
: Prints the names of the files in the directory set by the folder variable. (See ''Environment Variables.'')

fo[llowup] [*message*]
: Responds to (follows up on) a message, recording the response in a file whose name is derived from the author of the message. Overrides the record variable, if set. See also the F[ollowup], S[ave], and

C[opy] commands and outfolder under ''Environment Variables.''

F[ollowup] [*msglist*]
Responds to (follows up on) the first message in the *msglist*, sending the message to the author of each message in the *msglist*. The subject line is taken from the first message and the response is recorded in a file whose name is derived from the author of the first message. See also the fo[llowup], S[ave], and C[opy] commands and outfolder under ''Environment Variables.''

f[rom] [*msglist*]
Prints the header summary (''from'' portion) for the specified messages.

g[roup]*alias name...*
a[lias]*alias name...*
Declares an a[lias] or g[roup] for the given names. The names will be substituted when *alias* is used as a recipient. Useful in the .mailrc file.

h[eaders] [*message*]
Prints the page of headers which includes the message specified. The screen variable sets the number of headers per page. (See ''Environment Variables.'') See also the z command.

hel[p]
Prints a summary (help list) of commands.

ho[ld] [*msglist*]
pre[serve] [*msglist*]
Holds (preserves) the specified messages in the mailbox.

i[f] *s | r*
*mail-commands*
el[se]
*mail-commands*
en[dif]
Conditional executions, where *s* will execute following *mail-commands*, up to an el[se] or en[dif], if the program is in *send* mode (that is, not receiving or reading mail), and *r* causes the *mail-commands* to be executed only in *receive* mode. Useful in the .mailrc file.

ig[nore] *header-file...*
di[scard] *header-file...*
Suppresses printing of (ignores or discards) the specified header fields when displaying messages on the screen. Examples of header fields to ignore are status and cc. All fields are included when the message

is saved. The P[rint] and T[ype] commands override this command.

l[ist]
Prints (lists) all commands available. No explanation is given.

m[ail] *name...*
Mails a message to the specified users.

mb[ox] [*msglist*]
Arranges for the given messages to end up in the standard mbox save file when mailx terminates normally. See mbox under "Environment Variables" for a description of this file. See also the ex[it] and q[uit] commands.

n[ext] [*message*]
Displays the next message matching *message*. A *msglist* may be specified, but in this case, the first valid message in the list is the only one used. This is useful for jumping to the next message from a specific user, since the name would be taken as a command in the absence of a real command. See the discussion of *msglist*s, preceding, for a description of possible message specifications.

pi[pe] [*msglist*] [*shell-command*]
| [*msglist*] [*shell-command*]
Pipes the message through the given *shell-command*. The message is treated as if it were read. If no arguments are given, the current message is piped through the command specified by the value of the cmd variable. If the page variable is set, a formfeed character is inserted after each message. (See "Environment Variables.")

pre[serve] [*msglist*]
ho[ld] [msglist]
Preserves (holds) the specified messages in the mailbox.

P[rint] [*msglist*]
T[ype] [*msglist*]
Prints (types) the specified messages on the screen, including all header fields. Overrides suppression of fields by the ig[nore] command.

p[rint] [*msglist*]
t[ype] [msglist]
Prints (types) the specified messages. If crt is set, the messages longer than the number of lines specified by the crt variable are paged through the command specified by the PAGER variable. The default command is pg(1). (See "Environment Variables.")

q[uit]
: Exits (quits) from mailx, storing messages that were read in mbox and unread messages in the mailbox. Messages that have been saved explicitly in a file are deleted.

R[eply] [*msglist*]
R[espond] [msglist]
: Responds to the author of each message in the *msglist*. The subject line is taken from the first message. If record is set to a filename, the response is saved at the end of that file. (See ''Environment Variables.'')

r[eply] [*message*]
r[espond] [*message*]
: Replies to the specified message, including all other recipients of the message. If record is set to a filename, the response is saved at the end of that file. (See ''Environment Variables.'')

S[ave] [*msglist*]
: Saves the specified messages in a file whose name is derived from the author of the first message. The name of the file is taken to be the author's name with all network addressing stripped off. See also the C[opy], fo[llowup], and F[ollowup] commands and outfolder under ''Environment Variables.''

s[ave] [*filename*]
s[ave] [*msglist*] *filename*
: Appends the specified messages to the end of the given file. The file is created if it does not exist. The message is deleted from the mailbox when mailx terminates unless keepsave is set. (See also ''Environment Variables'' and the ex[it] and q[uit] commands.) mbox is the default filename.

se[t]
se[t] *name*
se[t] *name=string*
se[t] *name=number*
: Defines (sets) a variable called *name*. The variable may be given a null, string, or numeric value. se[t] by itself prints all defined variables and their values. See ''Environment Variables'' for detailed descriptions of the mailx variables.

sh[ell]
: Invokes an interactive shell. (See also SHELL under ''Environment Variables.'')

si[ze] [*msglist*]
: Prints the size in characters of the specified messages.

so[urce] *filename*
Reads (sources) commands from the given file and returns to command mode.

to[p] [*msglist*]
Prints the top few lines of the specified messages. If the toplines variable is set, it is taken as the number of lines to print. (See "Environment Variables.") The default is 5.

tou[ch] [msglist]
Touches the specified messages. If any message in *msglist* is not specifically saved in a file, it will be placed in the mbox upon normal termination. See ex[it] and q[uit].

T[ype] [*msglist*]
P[rint] [*msglist*]
Prints (types) the specified messages on the screen, including all header fields. Overrides suppression of fields by the ig[nore] command.

t[ype] [*msglist*]
p[rint] [*msglist*]
Prints (types) the specified messages. If crt is set, the messages longer than the number of lines specified by the crt variable are paged through the command specified by the PAGER variable. The default command is pg(1). (See "Environment Variables.")

u[ndelete] [*msglist*]
Restores (undeletes) the specified deleted messages. Will restore only those messages deleted in the current mail session. If autoprint is set, the last message of those restored is printed. (See "Environment Variables.")

uns[et] *name...*
Causes the specified variables to be erased (unset). If the variable was imported from the execution environment (i.e., a shell variable) then it cannot be erased.

ve[rsion]
Prints the current version and release date.

v[isual] [*msglist*]
Edits the given messages with a (visual) screen editor. The messages are placed in a temporary file and the VISUAL variable is used to get the name of the editor. (See "Environment Variables.")

w[rite] [*msglist*] *filename*
Writes the given messages on the specified file, minus the header and trailing blank line. Otherwise equivalent to the s[ave] command.

x[it]
ex[it]
Exits from mailx, without changing the mailbox. No messages are saved in the mbox (see also q[uit]).

z[+|-]
Scrolls the header display forward or backward one screenful. The number of headers displayed is set by the screen variable. (See "Environment Variables.")

**Tilde escapes**

The following commands may be entered only from input mode, by beginning a line with the escape character (tilde (~) by default). See escape under "Environment Variables" for changing this special character.

~!*shell-command*
Escapes to the shell.

~. Simulates end-of-file (terminates message input).

~:*mail-command*
~_*mail-command*
Performs the command-level request. Valid only when sending a message while reading mail.

~? Prints a summary of tilde escapes.

~A Inserts the autograph string Sign into the message. (See "Environment Variables.")

~a Inserts the autograph string sign into the message. (See "Environment Variables.")

~b *name...*
Adds the *name*s to the blind carbon copy (bcc) list.

~c *name...*
Adds the *name*s to the carbon copy (cc) list.

~d Reads in the dead.letter file. See DEAD under "Environment Variables" for a description of this file.

~e Invokes the editor on the partial message. See also EDITOR under "Environment Variables."

~f [*msglist*]
Forwards the specified messages. The messages are inserted into the message, without alteration.

~h Prompts for Subject line and To, Cc, and bcc lists. If the field is displayed with an initial value, it may be edited as if you had just

typed it.

~i *string*
Inserts the value of the named variable into the text of the message. For example, ~A is equivalent to ~i Sign.

~m [*msglist*]
Inserts the specified messages into the letter, shifting the new text to the right one tab stop. Valid only when sending a message while reading mail.

~p Prints the message being entered.

~q Quits from input mode by simulating an interrupt. If the body of the message is not null, the partial message is saved in dead.letter. (See DEAD "Environment Variables" for a description of this file.)

~r *filename*
~< *filename*
~< ! *shell-command*
Reads in the specified file. If the argument begins with an exclamation point (!), the rest of the string is taken as an arbitrary shell command and is executed, with the standard output inserted into the message.

~s *string...*
Sets the subject line to *string*.

~t *name...*
Adds the given *name* to the To list.

~v Invokes a preferred (visual) screen editor on the partial message. See also VISUAL under "Environment Variables."

~w *filename*
Writes the partial message onto the given file without the header.

~x Exits as with ~q except the message is not saved in the DEAD file.

~| *shell-command*
Pipes the body of the message through the given *shell-command*. If the *shell-command* returns a successful exit status, the output of the command replaces the message.

**Environment variables**

The following are environment variables taken from the execution environment and cannot be altered within mailx.

HOME= *directory*
Specifies the user's base of operations.

MAILRC= *filename*
Specifies the name of the startup file. The default is $HOME/.mailrc.

The following variables are internal mailx variables. They may be imported from the execution environment or set via the se[t] command at any time. The uns[et] command may be used to erase variables.

allnet
Specifies all network names whose last component (login name) matches are treated as identical. This causes the *msglist* message specifications to behave similarly. The default is noallnet. See also the alt[ernates] command and the metoo variable.

append
Appends messages to the end of mbox file instead of at the top of mbox, upon termination. The default is noappend (i.e., by default mailx saves messages at the top of mbox on exit).

askcc
Prompts for the Cc list after message is entered. The default is noaskcc.

asksub
Prompts for subject if it is not specified on the command line with the -s option. This option is enabled by default.

autoprint
Enables automatic printing of messages after d[elete] and u[ndelete] commands. The default is noautoprint.

bang
Enables the special-casing of exclamation points (!) in shell escape command lines, as in vi(1). The default is nobang.

cmd=*shell-command*
Sets the default command for the pi[pe] command. There is no default value.

conv=*conversion*
Converts uucp addresses to the specified address style. The only valid conversion now is internet, which requires a mail delivery program conforming to the RFC822 standard for electronic mail addressing. Conversion is disabled by default. See also sendmail and the -U option.

crt=*number*
Pipes messages having more than *number* lines through the command specified by the value of the PAGER variable (pg(1) by default). This option is disabled by default.

DEAD=*filename*
Specifies the name of the file in which to save partial letters in case of untimely interrupt or delivery errors. The default is $HOME/dead.letter.

debug
Enables verbose diagnostics for debugging. Messages are not delivered. The default is nodebug.

dot
Takes a period on a line by itself during input from a terminal as end-of-file. The default is nodot.

EDITOR=*shell-command*
Specifies the command to run when the e[dit] or ~e command is used. The default is ed(1).

escape=*c*
Substitutes *c* for the ~ escape character.

folder=*directory*
Specifies the directory for saving standard mail files. User-specified filenames beginning with a plus (+) are expanded by preceding the filename with this directory name to obtain the real filename. If *directory* does not start with a slash (/), $HOME is prefixed to it. In order to use the plus (+) construct on a mailx command line, folder must be an exported sh environment variable. There is no default for the folder variable. See also outfolder, later.

header
Enables printing of the header summary when entering mailx. This command is enabled by default.

hold
Preserves all messages that are read in the mailbox instead of putting them in the standard *mbox* save file. The default is nohold.

ignore
Ignores interrupts while entering messages. Handy for noisy dial-up lines. The default is noignore.

ignoreeof
Ignores end-of-file during message input. Input must be terminated by a period (.) on a line by itself or by the ~. command. The default is noignoreeof. See also dot, above.

keep
Truncates the mailbox to zero length instead of removing it, when the mailbox is empty. The option is disabled by default.

keepsave
: Keeps messages that have been saved in other files in the mailbox instead of deleting them. The default is nokeepsave.

MBOX=*filename*
: The name of the file in which to save messages that have been read. The x[it] command overrides this function, as does saving the message explicitly in another file. The default is $HOME/mbox.

metoo
: Does not delete your login from the list, if your login appears as a recipient. The default is nometoo.

LISTER=*shell-command*
: Specifies the command (and flag options) to use when listing the contents of the folder directory. The default is ls(1).

onehop
: Disables alteration of the recipients' addresses, improving efficiency in a network where all machines can send directly to all other machines (i.e., one hop away). When responding to a message that was originally sent to several recipients, the other recipient addresses normally are forced to be relative to the originating author's machine for the response.

outfolder
: Causes the files used to record outgoing messages to be located in the directory specified by the folder variable, unless the pathname is absolute. The default is nooutfolder. See folder above and the S[ave], C[opy], fo[llowup], and F[ollowup] commands.

page
: Inserts a formfeed after each message sent through the pipe. This option is used with the pi[pe] command. The default is nopage.

PAGER=*shell-command*
: Specifies the command to use as a filter for paginating output. This can also be used to specify the flag options to be used. The default is pg(1).

prompt=*string*
: Sets the command mode prompt to *string*. The default is ?.

quiet
: Refrains from printing the opening message and version when entering mailx. The default is noquiet.

record=*filename*
: Records all outgoing mail in *filename*. This option is disabled by default. See also outfolder, above.

save
Enables the saving of messages in dead.letter on interrupt or delivery error. See DEAD for a description of this file. This option is enabled by default.

screen=*number*
Sets the number of lines in a screenful of headers for the h[eaders] command.

sendmail=*shell-command*
Alternates command for delivering messages. The default is mail(1).

sendwait
Waits for background mailer to finish before returning. The default is nosendwait.

SHELL=*shell-command*
Specifies the name of a preferred command interpreter. The default is sh(1).

showto
Prints the recipient's name instead of the author's name, when displaying the header summary and the message is from you.

sign=*string*
Specifies the variable inserted into the text of a message when the ~a (autograph) command is given. This option has no default (see also ~i (in ''Tilde Escapes'')).

Sign=*string*
Specifies the variable inserted into the text of a message when the ~A command is given. This option has no default (see also ~i (''Tilde Escapes'')).

toplines=*number*
Specifies the number of lines of header to print with the to (top) command. The default is 5.

VISUAL=*shell-command*
Specifies the name of a preferred screen editor. The default is vi(1).

**LIMITATIONS**

Where *shell-command* is shown as valid, arguments are not always allowed. Experimentation is recommended.

Internal variables imported from the execution environment cannot be unset (uns[et]).

The full internet addressing is not fully supported by mailx. The new standards need some time to settle down.

Attempts to send a message having a line consisting only of a . are treated as the end of the message by mail(1) (the standard mail delivery program).

**FILES**

/usr/bin/mailx
: Executable file

/usr/lib/mailx
: Executable file

$HOME/.mailrc
: Personal startup file

$HOME/mbox
: Secondary storage file

/usr/mail/*
: Post office directory

/usr/lib/mailx/mailx.help*
: Help message files

/usr/lib/mailx/mailx.rc
: Global startup file

/tmp/R[emqsx]*
: Temporary files

**SEE ALSO**

biff(1), csh(1), ksh(1), ls(1), mail(1), pg(1), sh(1)

*A/UX Essentials*

**NAME**

make — maintains, updates, and regenerates groups of files

**SYNOPSIS**

make [-a] [-b] [-B] [-d*digits*] [-e] [-f *description-file*] [-g] [-G] [-i] [-k] [-K] [-M] [-n] [-p] [-P] [-q] [-r] [-s] [-t] [-u] [-V] [*target*]...

**ARGUMENTS**

-a Updates all targets as if they were all out-of-date. This option is useful for completely rebuilding all files.

-b Uses compatibility mode for old description files. This mode is on by default.

-B Turns off compatibility mode.

-d*digits*
Specifies debug mode. If you specify this option without a *digits* argument, full debug mode is invoked. If you specify this option with a single digit, the specified debug subset is invoked; this option invokes each specified subset. Currently, subsets 0 and 1 are implemented.

-e Causes environment variables to override macro definitions within the description file.

-f *description-file*
Uses the description file specified by *description-file*. A description file of - (hyphen) denotes the standard input.

-g Turns on additional capabilities to automatically check-out Source Code Control System (SCCS) files. See ''SCCS File Handling'' in the ''Description'' section, later in this manual page.

-G Enables the Dynamic Include File Dependency Generation (DIFDG). You can also enable DIFDG by defining the variable MAKEDIFDGSUFFIXES as a list of legal suffixes for the source files to be searched.

-i Ignores any error code that might be returned by a shell command. This mode can also be entered if the target .IGNORE: appears in the description file. (See ''Built-in Targets'' in the ''Description'' section, later in this manual page.)

-k Causes make to abandon work on the current target, but continues to process other targets that do not depend on the abandoned target, if a shell command returns a nonzero status.

-K Turns off the -k option. This option is on by default. This option is most often used in a description file that invokes make, which is a member of a multilevel make hierarchy, and that is invoked by a top-

level make command with the -k option.

-M Stores the dependency map for all object files in a file. The default name for this file is ._Make_State. (You can change the default name by changing the value in the MAKEDEPFILE variable to the desired name.) You can also enable this option by defining the variable MAKEDEPFILE as the name of the file in which you want to store the map.

-n Prints the commands in the description file as they would be executed, but does not actually execute them. Even lines beginning with an @ (at sign) are printed. (See ''Targets and Dependency Statements'' in the ''Description'' section, later in this manual page.) However, if a command line has the string $(MAKE) in it, the line is always executed. (See discussion of the $MAKEFLAGS macro in ''Environment Variables and Macros,'' in the ''Description'' section.)

-p Prints out the built-in rules of make, including a complete set of macro definitions.

-P Searches for Pre and Post files in the directory /usr/lib. (See ''*MakeFile* Preprocesing and Postprocessing'' in the ''Description'' section.) For example, for a description file named x.mk, make will search for and read /usr/lib/x.mkPre and /usr/lib/x.mkPost.

-q Specifies a question. The make command returns a zero or a nonzero status code depending on whether or not the target file is up-to-date.

-r Causes make not to use its built-in rules. To do useful work, this option must be accompanied by an appropriate description file.

-s Specifies silent mode. Does not print command lines before executing them. This mode is also entered if the target .SILENT: appears at any place in the description file.

-t Touches the target files (causing them to be up-to-date) without executing any commands.

*target*
Specifies the program to be run.

-u Looks for makecomm and Makecomm files in the user's home directory, as specified by the $HOME environment variable, and in the current directory. The search order is $HOME/makecomm, $HOME/Makecomm, ./makecomm, and ./Makecomm. At most, one file from each directory is read by make. These files are read before any description files and can be used to define macros and rules.

`-V` Displays the current version of `make`.

**DESCRIPTION**

`make` is used to maintain, update, and regenerate groups of files. The `make` program was designed to manage the systematic regeneration of programs and is typically used for, but is not limited to, that purpose.

The actions of `make` are governed by a set of built-in rules. You can supplement or replace these rules by providing an appropriate description-file.

**Suffix List and Built-in Rules**

The `make` command uses a suffix list and a set of built-in rules to determine how to regenerate a file. The suffix list and the built-in rules are based on the file-naming requirements of the various software generation tools in the A/UX environment. For example, a file whose suffix is `.s` is typically an assembly-language program that is processed by the `as` command. A file whose suffix is `.c` is typically a C program that is processed by the `cc` command.

For example, if `make` is requested to regenerate a file called `x.o`, `make` examines the name of each file in the current directory and looks for all files that have a base name of `x` and a suffix. In this case, `make` finds the file `x.c` and then extracts and saves the suffix, `.c`. `make` then prepends the suffix to each member of the default suffix list, one at a time, and attempts to match the resulting string against each built-in rule, from the first to the last. If no match is found, `make` prepends the suffix to the next element in the suffix list. If no match is ever found, `make` concludes that it does not know how to regenerate the requested file. When a match is found, `make` executes the commands that are associated with the matched rule. In this case, the string `.c.o` matches the `.c.o` built-in rule. For the `.c.o` rule, the associated command is `cc -O -c` *base-name*`.c`. The `make` command then executes the command, which in this case generates the requested file, `x.o`.

For this version of `make`, the suffix list is as follows, reading across the columns:

```
.obj        .obj~       .for        .for~
.pas        .pas~       .f          .f~
.o          .c          .c~         .y
.y~         .l          .l~         .s
.s~         .sh         .sh~        .h
.h~         .i
```

For this version of `make`, the built-in rules are as follows, reading across the columns:

```
.c          .c~         .sh         .sh~
```

```
.c.o        .c~.o       .c~.c       .s.o
.s~.o       .c.i        .c~.i       .c.s
.y.o        .y~.o       .l.o        .l~.o
.y.c        .y~.c       .l.c        .c.a
.c~.a       .s~.a       .h~.h       .f.o
.f~.o       .p.o        .p~.o       .for.obj
.for~.obj   .pas.obj    .pas~.obj
```

The suffix list and built-in rules demonstrate three important features of `make`. First, the order of the suffix list and the built-in rules is extremely important because the order of both governs which rule will be used to process a file.

Second, the built-in rules demonstrate that the rightmost suffix member of a rule can be empty. This is true for the first four built-in rules. For example, the `.c` rule allows `make` to regenerate the file `x`, which has no suffix, from the file `x.c`.

Third, both the suffix list and the built-in rules contain the tilde (`~`) character. To `make`, the tilde character indicates an SCCS file (see `sccsfile`(4)). Because `make` was designed to parse suffixes, and SCCS files are identified by their `s.` prefix, `make` internally converts references of the form `s.`*filename* to *filename*`~`. Thus, the rule `.c~.o` would transform an SCCS C source file into an object file (`.o`).

By definition, a rule starts with a period (`.`) and cannot contain a slash (`/`). The format of a rule is:

*.target*`:`[`:`] [*dependency-list*]
<tab>[*shell-command*]
.
.
.

where *target* and at least one colon are required. Items enclosed in square brackets (`[]`) are optional.

A built-in rule cannot rely on another built-in rule to resolve a dependency. Only explicit dependencies can be listed in the dependency list of a rule.

### Description Files

The built-in rules are often supplemented or overridden by the contents of a user-written description file. If the `-f` option is not present, the search order for description files is as follows:

```
./makefile
./Makefile
./MakeFile
./s.makefile
```

```
./SCCS/s.makefile
./s.Makefile
./SCCS/s.Makefile
./s.MakeFile
./SCCS/s.MakeFile
```

The new description file, `MakeFile`, is described in ''`MakeFile` Preprocessing and Postprocessing'' later in the ''Description'' section. If the description file is `-`, the standard input is taken. More than one `-f` *description-file* argument pair can appear on the command line.

**Include Files**

If the string `include` or `Include` appears as the first seven characters of a line in a description file and is followed by a blank or tab character, `make` assumes that the rest of the line is the name of a description file, which is read by the current invocation of `make`, after macro substitution. The difference between `include` and `Include` is that if `include` does not exist, `make` will terminate with an error message. If `Include` does not exist, `make` will continue processing and will not issue an error message.

**Targets and Dependency Statements**

Although `make` can use its built-in rules to perform simple regeneration tasks, `make` requires direction from the user to accomplish more complicated tasks, such as regenerating a program that comprises multiple source files. You provide that direction in a description file from which `make` reads and processes user-written dependency statements to update one or more targets. The target is usually, but does not have to be, a program. If a dependency statement is incomplete, `make` uses its built-in rules to supplement the dependency statement.

The format for a dependency statement is nearly identical to the format for a built-in rule; a distinction is maintained so that if you want to completely replace an existing built-in rule, you can do so by providing the new rule in the description file. The format of a dependency statement is as follows:

*target* [*target*] : [ : ] [*dependency-list*] [;]
<tab>*shell-command*
.
.
.

Each *target* argument is an alphanumeric string separated by a blank. A target name cannot begin with a tab or a period ( . ) and cannot contain a colon ( : ) or a semicolon ( ; ). At least one colon is required; a second colon is optional. The colon must be preceded by at least one target. The *dependency-list* argument is a blank-separated list of items on which the

target depends. The items can be filenames or other targets. The *shell-command* arguments are the commands that will be executed to update the target. Each shell command must be preceded by a tab character.

If a shell command extends over two or more lines, you can escape the newline character, which is automatically placed at the end of the each line by the standard text editors, by preceding it with a backslash (\).

Command lines are executed one at a time, each by its own shell. To have a series of commands executed by the same shell, append a semicolon (`;`) to the end of each shell command and escape any newlines, as just described. This treatment of shell commands is particularly important for any command that is executed directly by the shell and whose result is effective only for the lifetime of the shell, such as the `cd` command.

By default, `make` hands shell commands to the Bourne shell, but if the `$SHELL` variable is set and exported in the user's environment, `make` hands shell commands to the specified shell.

The `make` command interprets the number sign (`#`) as the beginning of a comment. When parsing lines in a description file, `make` determines that the first line that does not begin with a tab or a # begins a new dependency statement or macro definition. (Macros are discussed later in the "Description" section.)

The following example illustrates many of the concepts described here. In this dependency statement, the target is a program called `pgm`. `pgm` depends on two files, `a.o` and `b.o`, and they in turn depend on their corresponding source files, `a.c` and `b.c` and a common file, `incl.h`:

```
#
# Making pgm.
#
pgm: a.o b.o
     cc a.o b.o -o pgm
a.o: incl.h a.c
     cc -c a.c;\
     echo "Done compiling a.c"
b.o: incl.h b.c
     cc -c b.c;\
     echo "Done compiling b.c"
```

The `make` command updates a target only if its dependents are newer than the target. For this example, `make` will generate a new `a.o` file if the modification time of either `incl.h` or `a.c` is newer than the modification time of `a.o` or if `a.o` does not exist. The same is true of `b.o`.

Each line in a description file is terminated by a newline character. In the previous example, a semicolon is appended to the end of the line that invokes the C Compiler and the newline is escaped by a backslash character to pass both the invocation of the C Compiler and the `echo` command to the same instance of the shell.

Because the built-in rules can be used to determine that `a.c` and `b.c` can be used to generate `a.o` and `b.o`, respectively, the dependency statement for making `pgm` can be written more briefly:

```
pgm:  a.o b.o
      cc a.o b.o -o pgm
a.o b.o:    incl.h
```

When parsing shell commands, `make` first prints the command and then passes everything except the initial tab character directly to the shell as is. The following example uses "<tab>" to represent the tab character.

```
<tab>echo a\
<tab>b
```

When this example is processed by `make`, the following output will be produced:

```
<tab>echo a\
<tab>b
ab
```

The first and second lines are printed by `make` before the initial tab character is stripped, and the third line is printed by the `echo` command.

You can turn off printing of the command before execution by preceding each command with the `@` character. For example, if the description file contains

```
@echo a\
b
```

the output of `make` will be

```
ab
```

### Single-Colon and Double-Colon Targets

As described in the format for a dependency statement earlier, a dependency statement must have at least one colon, and can have an optional second colon. For the dependency statement

```
a:    b.o
a:    c.o
a:    d.o
```

the `make` command concludes that `a` depends on `b.o`, `c.o`, and `d.o`; that

is, the preceding dependency statements are equivalent to

```
a:      b.o c.o d.o
```

The make command treats dependency statements that have two colons differently, so that a description file that contains

```
a::     b.o
a::     c.o
a::     d.o
```

contains three separate dependency statements. If you use the single-colon rule, the target is updated when either of these conditions is true:

- Any dependent is newer than the target.
- The target does not exist.

If you use the double-colon rule, the target is updated when any of these conditions is true:

- Any dependent is newer than the target.
- The target does not exist.
- The target does not have a dependency list.

Double-colon dependency statements are useful for situations where a single target name is desired, but depending on the context, different commands need to be executed to update the target. For example:

```
a::     a.sh
        cp a.sh a

a::     a.c
        cc -o a a.c
```

The first dependency statement is executed if the current directory contains the file a.sh, and the second dependency statement is executed if the current directory contains the file a.c. If both a.sh and a.c exist in the current directory, one or both target statements could be executed depending on whether the target is older than the dependents. If the target is missing, the target is considered to be older than the dependents.

Double-colon dependency statements are also useful for situations in which a target has the same name as a subdirectory in the current directory. For example, if the dependency statement is

```
mail:
        cd mail;\
        cc mail.o -o mail
```

and the current directory contains a directory called mail, the commands to update the target will never be invoked. This is because make assumes

that if the target `mail` exists (even if it is a directory) and has no dependency list, the target is up-to-date. As a result, the target name in a single-colon dependency statement should never be the name of a subdirectory in the current directory. Often, however, assigning the target the same name as the directory that contains the files on which the target depends makes the description file more meaningful to the user. The `make` command processes double-colon dependency statements differently, so that

```
mail::
        cd mail;\
        cc mail.o -o mail
```

works as desired. This is because if a double-colon dependency statement has no dependency list, `make` processes the commands that update the target even if the target already exists. The caveat is, however, that `make` always recompiles `mail.c` even when the executable `mail` file is newer than `mail.c`. A better solution to this problem is described under ''Attributes,'' later in the ''Description'' section.

As mentioned earlier, you can replace a built-in rule by providing a new rule of the same name in the description file. You can also disable a built-in rule, as shown by the following example:

```
.c.a:;
```

The `make` command interprets a semicolon (`;`) that is not preceded by a command as a null command, which has the effect of disabling the specified rule.

Just as the built-in rules can be replaced, so can the default suffix list. The following line in a description file clears the suffix list:

```
.SUFFIXES:
```

The following line appends additional suffixes to the end of the existing suffix list:

```
.SUFFIXES: .n .x
```

Multiple suffix lists accumulate until cleared, as just shown, or until `make` terminates.

**Built-in Targets**

The targets described in this section are actually built-in rules that you can enable by including them in a description file. If present, they modify the default behavior of `make`. Because `make` reads the entire description file before beginning to process dependency statements, the following built-ins, which must appear at the beginning of a line, are processed first, whether they appear at the beginning, middle, or end of the description file.

`.DEFAULT:`
If a file must be made but there are no explicit shell commands or relevant built-in rules, the shell commands listed under `.DEFAULT:` are used.

`.IGNORE:`
If present, `.IGNORE` has the same effect as the `-i` option, which is to cause `make` to ignore nonzero return codes from commands.

`.MAKESTOP [` *exit-code* `]:`
If present, `.MAKESTOP:` causes `make` to exit. `.MAKESTOP:` is useful in a multilevel directory and description file hierarchy for quickly bypassing a `make` command in a particular directory or directories. The *exit-code* argument is optional. If you do not specify a value, the exit code defaults to 0. If no exit code is specified or if the specified exit code is zero, `make` exits silently. If a nonzero exit code is specified, `make` prints a warning message.

`.PRECIOUS:`
The default behavior of `make` is to remove a target and its dependents when a quit or interrupt signal is received while `make` is processing the commands that update the target. Because the actions of `make` depend in large part on the mere existence of a file, removal of potentially incomplete files helps ensure that the proper files are regenerated each time. You can avoid removal by making specific files dependent on `.PRECIOUS:`. See ''Error Handling,'' later in the ''Description'' section, for further details.

`.SILENT:`
If present, `.SILENT:` has the same effect as the `-s` command option.

**Environment Variables and Macros**

The documentation for `make` uses the term ''macro'' to name the entities that the shell documentation calls ''environment variables.'' A macro is a variable whose value is set in a description file and can be overridden from the `make` command line. Although `make` and the shell use these entities in nearly identical ways, there are differences, which are described in this section.

The sample shell script

```
NAME=Joe
echo NAME
echo $NAME
```

produces the following result:

```
NAME
```

```
Joe
```

The difference between the first and second `echo` commands is that the first simply requests that the string `NAME` be echoed, while the second, because of the prepended dollar sign (`$`), requests that the contents of `NAME` be echoed. Such a request is called ''expansion.''

Expansion is handled differently by `make`. The example description file

```
NAME=joe

all:
        echo NAME
        echo $NAME
```

produces the following result

```
NAME
AME
```

This result is produced because `make` requires that macro names that are longer than one character be enclosed in parentheses or braces for expansion to occur. In this case, `make` sees the `$` and attempts to expand a variable named `N`. No such variable is set, so nothing is echoed and the `echo` command completes by echoing `AME`. The following description file produces the desired result:

```
NAME=joe

all:
        echo NAME
        echo $(NAME)
```

The use of braces is equivalent to the use of parentheses, so that `${NAME}` is equivalent to `$(NAME)`.

Each time `make` evaluates a macro, `make` strips one dollar sign (`$`) from it. Therefore, an extra dollar sign should be prepended to any macro that is part of a shell command line. When `make` is invoked, `make` reads the user's environment and makes all the variables found there available for modification by the description file.

Environment variables are processed before any description file and after the built-in rules; macro definitions in a description file override environment variables of the same name. The `-e` option causes environment variables to override macro definitions of the same name in a description file.

The formal definition of a macro is

*macro-name* = *string2*

By convention, macro names are uppercase. The *macro-name* argument is an alphanumeric string that cannot contain a colon (:) or a semicolon (;). The equal sign (=) can be surrounded by spaces or tabs. The *string2* argument is defined as all characters up to a comment character or an unescaped newline.

The make command provides several built-in macros. Here is a description of each macro.

MAKECDIR
: A read-only macro that expands into the full pathname of the current directory.

MAKEFLAGS
: If the MAKEFLAGS macro is not present in the environment, make creates and assigns to it the options with which make was invoked. MAKEFLAGS is processed by make as containing any legal input option (except -f, -p, -P, -r, and -u). Thus, MAKEFLAGS always contains the current input options. This macro proves very useful for "super-makes." In fact, as noted earlier, when the -n option is used, the command $(MAKE) is executed anyway; hence, you can perform a make -n operation recursively on a whole software system to see what would have been executed. This is because the -n operation is put in MAKEFLAGS and passed to further invocations of $(MAKE). This is one way of debugging all of the description files for a software project without actually causing the execution of update commands.

MAKELEVEL
: If the MAKELEVEL macro is not present in the environment, make creates it, assigns an initial value of zero, and exports it. If it is already present in the environment, make increments the value of MAKELEVEL by one. In this way, each subordinate invocation of make can know its level in a multilevel make hierarchy. This macro is read-only and cannot be modified by the description file.

MAKEBDIR
: If the MAKEDIR command is not present in the environment, make creates it, and assigns to it the absolute pathname of the current directory. If it is already present in the environment, the value of MAKEBDIR is not changed. MAKEBDIR provides a way for each subordindate invocation of make to obtain the pathname of the top-level make command.

MAKEGOALS
: For every invocation of make, make creates the MAKEGOALS macro

and assigns to it the targets that were specified on the command line. For the command line

```
$ make clean all clobber
```

MAKEGOALS will be set to "clean all clobber." If the current invocation of make invokes make, the invocation can be made as shown in the following example:

```
MAKE=make
cd dir; $(MAKE) $(MAKEGOALS)
```

In this way, the same command-line arguments can be passed to subordinate invocations of make.

VPATH

This version of make supports special processing of the macro VPATH, if set. VPATH is useful for processing files that are located in a directory other than the current directory. In the following example, main.c is located in the current directory. func1.c is located in ../common, and func2.c is located in ../incl. The make command searches the directories specified by the VPATH variable for any dependencies that are not in the current directory. Consider the following example.

```
VPATH=../common:../incl
main: main.o func1.o func2.o
      cc -o $@ $>
```

In this example, $@ (described later in this section) expands to the target name and $> (described later in this section) expands to the list of dependencies on the current target. If main.c, func1.c, or func2.c is not present in the current directory, make uses its built-in rules to search for SCCS versions of the files in the current directory. (See "SCCS File Handling," later in the "Description" section.) If SCCS versions of the files are not found, make searches the pathnames specified by VPATH.

The following built-in macros define values for common software generation programs or options to those programs. Description files can replace or supplement the values of these macros to change the way in which the built-in rules work.

AR This macro is defined as ar.

AS This macro is defined as as.

ASFLAGS

This macro is defined as null and is provided as an argument to the assembler.

CC This macro is defined as `cc`.

CFLAGS
This macro is defined as `-O` and is provided as an argument to the C Compiler.

CHMOD
This macro is defined as `chmod`.

CP This macro is defined as `cp`.

F77
This macro is defined as `f77`.

F77FLAGS
This macro is defined as null and is provided as an argument to the Fortran Compiler.

FORTRAN
This macro is defined as `fortran`.

FORTRANFLAGS
This macro is defined as null and is provided as an argument to the Fortran Compiler.

GET
This macro is defined as `get` and is used to get SCCS versions of files.

GFLAGS
This macro is defined as null and is provided as an argument to `get`.

LD This macro is defined as `ld`.

LDFLAGS
This macro is defined as null and is provided as an argument to `ld`.

LEX
This macro is defined as `lex`.

LFLAGS
This macro is defined as null and is provided as an argument to `lex`.

MAKE
This macro is defined as `make`.

MV This macro is defined as `mv`.

PASCAL
This macro is defined as `pascal`.

PASCALFLAGS
This macro is defined as null and is provided as an argument to `pc`.

PC This macro is defined as pc.

PCFLAGS
This macro is defined as null and is provided as an argument to pc.

RM This macro is defined as rm.

YACC
This macro is defined as yacc.

YFLAGS
This macro is defined as null and is provided as an argument to yacc.

The following six built-in macros have special expansion capabilities that are useful for writing shell commands.

* The * macro stands for the filename part of the current dependent with the suffix deleted. It is evaluated only for built-in rules.

@ The @ macro stands for the full target name of the current target. It is evaluated only for explicitly named dependencies.

< The < macro is evaluated only for built-in rules or the .DEFAULT rule. It is the module that is out-of-date with respect to the target (that is, the ''manufactured'' dependent filename). Thus, in the .c.o rule, the < macro would evaluate to the .c file. Here is an example for making optimized .o files from .c files:

```
.c.o:
cc -c -O $*.c
```

Here is another example:

```
.c.o:
cc -c -O $<
```

? The ? macro is evaluated when explicit rules from the description file are evaluated. It is the list of prerequisites that are out-of-date with respect to the target; essentially, those modules must be rebuilt.

% The % macro is evaluated only when the target is an archive library member of the form lib(file.o). In this case, @ evaluates to lib and % evaluates to the library member, file.o.

> The > macro is expanded to list all of the dependencies on the current rule.

These six macros can have alternative forms. When an uppercase D or F is appended to any of the six macros, the meaning is changed to ''directory part'' for D and ''file part'' for F. Thus, $(@D) refers to the directory part of the string @. If there is no directory part, ./ is generated.

The following description file demonstrates the use of the ? and > macros in their standard and alternative forms:

```
pgm:
        @echo "? = $?"
        @echo "?D = $(?D)"
        @echo "?F = $(?F)"
        @echo "> = $>"
        @echo ">D = $(>D)"
        @echo ">F = $(>F)"

pgm:  dir/a.o
pgm:  dir/b.o
pgm:  dir/c.o
```

When `a.o` is the only object that is newer than `pgm`, the following data is output from `make`:

```
? = dir/a.o
?D = dir
?F = a.o
> = dir/a.o dir/b.o dir/c.o
>D = dir dir dir
>F = a.o b.o c.o
```

**Macro Substitution**

The contents of a macro can be substituted as shown here:

`$(`*macro-name* [`:` *subst1* =[ *subst2*]]`)`

The contents of a macro are replaced by *string2*, which is delimited by blanks, tabs, newline characters, or the beginning of a line. A substitute sequence can replace only the trailing characters of *subst1*. For example,

```
SAMPLE=/a/b/file.test

all:
        @echo "1    $(SAMPLE:file=FILE)"
        @echo "2    $(SAMPLE:test=TEST)"
        @echo "3    $(SAMPLE:a/=A/)"
        @echo "4    $(SAMPLE:b/file.test=K)"
        @echo "5    $(SAMPLE:a=A)
```

has the following output:

```
1       /a/b/file.test
2       /a/b/file.TEST
3       /a/b/file.test
4       /a/K
```

```
5       /a/b/file.test
```

In this example, only the second and fourth commands are successful. The other commands fail because they do not substitute the trailing characters of the expanded macro.

The following example demonstrates the usefulness of string substitution:

```
all:  /u/test/a.o
      cc -S $(?:a=.c)
      mv $(?:.o=.s) .tmp
      sed "s/text/data/" > $(?:.o=.s) < .tmp
      as -o $@ $(?:.o=.s)
      rm .tmp
```

This example uses the `@` macro (expand to the full target name of the current target) and the `?` macro (expand to the list of out-of-date dependencies) macros to produce the assembly-language file for each dependent of the `all` target, change each occurrence of `text` to `data` by using `sed`, and assemble each resulting `.s` file.

Substitution works only on macros that are part of shell command lines. This version of `make` does not support substitutions of macros that are part of dependency lists.

**Macro Testing**

This version of `make` supports the testing of macros, where the format is

`$(`*macro-name*`:`*test-operator*`)`

The macro name can be set or unset and with or without an assigned value. The *test-operator* argument can be one of the following values:

`L` The macro is expanded to the length of its contents. An empty or null value expands to zero. This test operator is useful for determining whether or not to examine the contents of a macro.

`V` If the macro is set and has a non-null value, the macro is expanded to null; otherwise, the macro is expanded to `#`. This test can be used to control the execution of command lines, as shown here:

`$(`*macro-name*`:V)` *conditional-command*

If the macro is not set, the macro is expanded to `#`, which causes `make` to evaluate the line as a comment. As a result, *conditional-command* is not executed.

`N` If the macro is set and has a non-null value, the macro is expanded to `#`; otherwise, the macro is expanded to null. This macro is the opposite of the `V` test operator and is used in the same way as the `V` test operator.

For example, assume that you want to have a target called `clean` if the macro `$CLNFILES` is set. The dependency statement would remove the files expanded from this macro. Here is how the dependency statement would look:

```
$(CLNFILES:V)clean:
$(CLNFILES:V)     @echo "Removing: $(CLNFILES)";\
        rm -f $(CLNFILES)
```

If the `$CLNFILES` macro is set and contains a non-null value, the `$(CLNFILES:V)` macro will be ignored when `make` reads the description file, and the line will be processed just as if the description file contained the following code:

```
clean:
        @echo "Removing: $(CLNFILES)";\
        rm -f $(CLNFILES)
```

The `$(CLNFILES)` macro is expanded just before the command line is executed. Macros that have test operators are expanded during the parsing of the command line. This means that the order of macros that have test operators is significant, which is unlike the normal behavior of macros that do not have test operators. Normal macros are expanded after all description files have been read and command-line execution has begun. You can delay the expansion of macros that have test operators by prefixing more `$` characters, just as you can with normal macros.

In the example preceding, notice that `$(CLNFILES:V)` does not appear in front of each line. This is because a single command line was used, and that command line extended over two lines, with the newline escaped by the backslash (`\`) character. If there had been multiple command lines, a `$(CLNFILES:V)` macro would have been required at the beginning of each line.

**Global Macros**

This version of `make` provides special handling of user-defined macros that begin with `G_`. Such macros are automatically exported to the environment, so they can be easily passed to subordinate invocations of `make`.

**Default Macro Values**

This version of `make` supports a default expansion value for nonexistent or null-valued macros. The default value is specified as follows:

```
$(XYZ::default)
```

In this case, if the macro `XYZ` is not defined or has a null value, it is expanded to `default`. The default value is restricted to a single word, so the following example would not have the intended result:

```
$(XYZ::This is the default)
```

Instead, `XYZ` would bd expanded to `This`. But the following example would work:

```
DEFAULT = This is the default

all:
        @echo "$(NOTDEFINED::$(DEFAULT))"
```

**Pattern Matching on Macros**

This version of `make` supports limited shell-style pattern-matching on macros. Here is an example:

```
cc -o $@ $(>:=*.o)
```

Each word in the expanded `>` macro is tested (shell-style) against the asterisk (`*`) pattern and compiled when there is a match.

**Attributes**

This version of `make` understands attributes, which can be placed before or after the dependents in a dependency list as shown here:

*target*: [*attributes*] [*dependents*] [*attributes*]

You can use any of the following attributes:

`.FAKE`
: If the target exists and has no dependents, the normal behavior of `make` for single-colon dependency statements is to do nothing. The addition of the `.FAKE` attribute to the dependency statement requires `make` to treat the target as if it did not exist. This, in turn, forces `make` to execute the associated commands.

`.CURTIME`
: This attribute causes `make` to use the current time rather than the most recent modification time of the target, even if the target does not exist. This attribute is used with the `.FAKE` attribute to prevent the associated dependency statement from being invoked unless the dependents have been updated with a newer time.

`.IDEBUG`*n*
: If present, `.IDEBUG`*n* tells `make` not to display debugging information about this target at the desired debugging level. The variable *n* is set to a debugging level in the range of 0 to 9. For example, to prevent this target from showing up in your debugging sections at levels 0 and 1, you would specify `.IDEBUG0` and `.IDEBUG1`.

`.IGNORE`
: This attribute causes errors from any command of the target to be

ignored.

.MAIN
The normal behavior of make when invoked without a target name on the command line is to search the description file for the first target, process the target, and then terminate. The addition of .MAIN to a dependency statement causes make to treat the associated dependency statement as if it were the first dependency statement in the description file.

.NOMESS
If present, .NOMESS causes make not to echo commands or issue any warning or error messages from commands. This attribute is useful in Pre and Post files if you do not want to see messages from these files.

.PRE
This attribute informs make that the associated target is to be made before any others, including .MAIN. You can use this attribute to place initialization commands. Because the entire description file is read before the targets are processed, the placement of this attribute is position-independent within the description file.

.POST
This attribute informs make that the associated target is to be processed after all others. You can use this attribute to place cleanup commands.

.PRECIOUS
With this attribute, the document is considered ''precious.''

.KEEPTIME
This attribute causes make to maintain the target's original modification time, even after the target has been regenerated.

.OLDTIME
This attribute causes make to ignore the target's modification time and apply a modification time of 0 for the purpose of determining if the target should be updated. After the target is regenerated, make sets the correct modification time.

.SILENT
With this attribute, the commands of this target are not echoed before execution.

.NOVPATH
This attribute causes make to ignore the $VPATH macro for the associated target.

If targets that have .MAIN, .PRE, and .POST attributes are dependents of other targets, the targets are made in the order dictated by the dependencies, not in the order dictated by the attributes.

Attributes in double-colon dependency statements apply to all of them as a unit.

**Libraries**

If a target or dependency name contains parentheses, it is assumed to be an archive library, the string within parentheses referring to a member within the library. Thus lib(file.o) and $(LIB)(file.o) both refer to an archive library that contains file.o. (Assuming that the LIB macro has been previously defined.) The expression $(LIB)(file1.o file2.o) is not legal. The built-in rules pertaining to archive libraries have the form .*suffix*.a where *suffix* is the suffix from which the archive member is to be made. An unfortunate by-product of the current implementation requires *suffix* to be different from the suffix of the archive member. Thus, you cannot have lib(file.o) depend upon file.o explicitly.

Here is an example of the most common use of the archive interface. This example assumes that the source files are all C-type source:

```
lib:
     lib(file1.o) lib(file2.o) lib(file3.o)
     @echo lib is now up-to-date
.c.a:
     $(CC) -c $(CFLAGS) $<
     ar rv $@ $*.o
     rm -f $*.o
```

In fact, the .c.a rule listed in this example is built into make and need not actually be specified. Here is a more interesting, but more limited, example of an archive library maintenance construction:

```
lib:
     lib(file1.o) lib(file2.o) lib(file3.o)
     $(CC) -c $(CFLAGS) $(?:.o=.c)
     ar rv lib $?
     rm $?
     @echo lib is now up-to-date
.c.a:;
```

Here the substitution mode of the macro expansions is used. The $? list is defined to be the set of object-file names (inside lib) whose C source files are out-of-date. The substitution mode translates the .o to .c. (Unfortunately, you cannot as yet transform to .c~; however, this capability may become possible in the future.) Note also the disabling of

the .c.a: built-in rule, which would have created each object file, one by one. This particular construction speeds up archive library maintenance considerably. This type of construction becomes very cumbersome if the archive library contains a combination of assembly-language programs and C programs.

**MakeFile Preprocessing and Postprocessing**

If, by employing its search rules, make finds a description file named MakeFile or SCCS/s.MakeFile, the files /usr/lib/MakeFilePre and /usr/lib/MakeFilePost, if present, are read. If, for example, the description file is named x.mk and the -P option is specified, make will read /usr/lib/x.mkPre and /usr/lib/x.mkPost, if present.

The preprocessing and postprocessing files are description files that can be used to store global environmental settings and rules for events that you want to occur prior to and after processing of a description file.

If the macro G_SGS_ROOT is present in the environment, make considers this macro to be the root for finding the preprocessing and postprocessing files. For example, if G_SGS_ROOT is set to newroot, make will look for the preprocessing and postprocessing files in the directory /newroot/usr/lib.

**SCCS File Handling**

As described in "Description Files" earlier in the "Description" section, make can checkout read-only copies of makefile, Makefile, or MakeFile from an SCCS version of the file in the current directory or an SCCS version of the file located in a subdirectory named SCCS. This ability is separate from the built-in rules that govern the check-out of SCCS versions of dependents and is not turned off by the -r option to make.

The built-in rules for dependents can checkout only a file for which there is an SCCS version in the current directory; the built-in rules cannot checkout an SCCS version of a file located in a subdirectory named SCCS.

If the -g option is used, however, and the file does not exist in the current directory, make will first use its built-in rules to check-out the file in the current directory. If this attempt fails because the SCCS version does not exist, the -g option causes make to search the current directory again and then search a subdirectory named SCCS, if present, and check out the file if found. Note that the current directory is searched twice, once by the built-in rules and once because the -g option was specified. Searching the current directory makes the -g option especially useful with the -r option. The -r option turns off the built-in rules; thus, when -r is used with the -g option, the current directory is searched just once.

If a VPATH variable is present and set, make uses the built-in rules to search the specified directories for SCCS versions of the file. If the -g option is specified, any subdirectories named SCCS in the VPATH directories are also searched.

The built-in rules for checking out dependents from their SCCS versions in the current directory are not used to process include files. (See ''Include Files'' earlier in the ''Description'' section.) If an include file exists only in its SCCS version in the current directory, make will not check it out. If make is invoked with the -g option, however, make will check out an include file if it is present in an SCCS version in the current directory or in a subdirectory named SCCS.

In no case does make check out a copy of a description file, an include file, or a dependent file if the file already exists.

If a file is checked out by means of the built-in rules, make does not remove the checked-out copy. If a file is checked out by the action of the -g option, the checked-out copy is automatically removed when no longer needed.

**Dynamic Include File Dependency Generation (DIFDG)**

The make command can examine selected source files for #include directives. These include files are added to the target's dependency list. This feature, called DIFDG, relieves you from having to set up and create the include-file dependency list.

The only disadvantage to having make create the dependency list is that some include files may be placed on the target's dependency list that would normally be left out during the compile because of an #ifdef statement. However, this behavior does not cause any problems; the target is still updated properly. The make utility does not add an include file to the target's dependency list unless that include file really exists, so no damage can result.

You enable DIFDG by defining the _MAKE_DIFDG_SUFFIXES variable with a list of source-file suffixes to be searched, as in this example, or by specifying the -G (generate) option to make:

```
_MAKE_DIFDG_SUFFIXES= .c .s .f .p .l
```

The _MAKE_DIFDG_SUFFIXES variable must contain at least one suffix to enable DIFDG. An empty variable here does not have added meaning. The suffixes .o, .h, and .a are ignored.

You can specify the list of directories to search for these include files by using the _MAKE_DIFDG_INCDIRS variable. The order is important because make searches each directory for include files until the files are found. Like cpp(1), make looks for include files of the form header.h first in the same directory as the source file (not always the current

directory), then in the directories listed in the _MAKE_DIFDG_SUFFIXES variable. If an include file has the form <header.h>, the only directories searched are those listed in the variable. If this variable is defined but not assigned a value, the only directory that will be searched is the source-file directory. This means that <header.h> forms will always fail because there will not be a directory to search. If this variable is not defined, make uses the default /usr/include directory.

You can add a prefix to each include file dependent whose full pathname starts with /usr/include, by using the _MAKE_DIFDG_PREFIX variable. Use this variable only when you have specified the -M (map) option to request that the include-file dependencies be written. There is no default. Here is an example:

```
_MAKE_DIFDG_PREFIX= $$(SGS_INCDIR)
```

You can also use the c preprocessor, cpp, to create include-file dependency files. This method is much slower than that used by make. The last variable for DIFDG, _MAKE_DIFDG_CPPFLAGS, is defined with the flags to be passed to cpp. Merely defining this variable enables the cpp method of finding include files. If you do not define this variable, make uses the faster method. When you assign a value to this variable, keep in mind that only words that start with a hyphen are passed to cpp as in this example. (It is assumed that DEFINES is a variable that contains -D style macros.)

```
_MAKE_DIFDG_CPPFLAGS= -Y $(DEFINES)
```

If the _MAKE_DIFDG_FILE variable is set and non-null, and DIFDG is enabled, the DIFDG include-file dependencies will be written to the variable it when make exits. If the -G option is used, the defaults are as follows:

```
_MAKE_DIFDG_SUFFIXES= .c .l .y
_MAKE_DIFDG_INCDIRS= /usr/include
_MAKE_DIFDG_CPPFLAGS=
_MAKE_DIFDG_PREFIX=
```

### Error Handling

Shell commands that return a nonzero status normally terminate make. You can modify this behavior in a number of ways:

-i If present on the command line, this option causes make to ignore any nonzero status that is returned by a shell command and to continue processing the current description file.

.IGNORE:
See "Built-in Targets," earlier in the "Description" section.

`-k` If present on the command line, this option causes `make` to abandon work on the current target if a shell command returns a nonzero status. The `make` command will continue to work on other targets in the description file that do not depend on the target for which the error was received.

`-` If a hyphen is prepended to any shell command, `make` will not terminate on an error that might occur as a result of executing the command, but will continue processing the description file. The `-` can be combined with the previously described `@` symbol, described in "Section Name" earlier in the "Description" section, which suppresses the printing of the command before it is passed to the shell. The combination can be either `-@` or `@-`. Both error messages returned by the shell and `make` commands standard error messages are still printed.

As mentioned earlier, `make` removes a target and its dependents when a quit or interrupt signal is received. The following conditions apply:

- The dependency statement must be a single-colon dependency statement. If the dependency statement is a double-colon dependency statement, `make` does not remove files.
- The target must have existed before processing of the dependency statement began.
- The target must not be a dependent of the built-in target `.PRECIOUS:`.
- The `make` must not have been invoked with the `-n` option.
- The `make` command must not have been invoked with the `-t` option.

An example of the standard termination message generated because of a nonzero status code is shown here. In this case, `make` terminates because the description file, `x.mk`, does not exist in the directory `/dir`.

```
$ make -f x.mk
Make: Cannot read description file /dir/x.mk
Make: Stopped in directory /dir.
```

**LIMITATIONS**

The syntax `(lib(file1.o file2.o file3.o)` is illegal.

You cannot build `lib(file.o)` from `file.o`.

The macro `$(a:.o=.c~)` does not work.

**FILES**

`$HOME/[Mm]akecomm`
File

`/bin/make`
Executable file

`[Mm]akecomm`
File

`s.[Mm]ake[Ff]ile`
File

`SCCS/s.[Mm]ake[Ff]ile`
File

**SEE ALSO**

`cd`(1), `csh`(1), `ksh`(1), `lex`(1), `sh`(1), `touch`(1), `yacc`(1)

`printf`(3S), `sccsfile`(4) in *A/UX Programmer's Reference*

"`make` Reference" in *A/UX Programming Languages and Tools, Volume 2*

**NAME**

makedev — prepares troff description files

**SYNOPSIS**

makedev *files*

**ARGUMENTS**

*files* Specifies the description file to be read.

**DESCRIPTION**

makedev reads description files about a particular device and converts them into a form suitable for reading by troff(1). Input to makedev is in the format described in font(5).

**FILES**

/usr/bin/makedev
Executable file

**SEE ALSO**

troff(1)

font(5) in *A/UX Programmer's Reference*

*A Typesetter-Independent* troff, Brian W. Kernighan (Bell Laboratories, 1982)

*Adventures with Typesetter-Independent* troff, Mark Kahrs and Lee Moore (University of Rochester Technical Report 159, 1985)

## NAME

makekey — generates an encryption key

## SYNOPSIS

makekey

## DESCRIPTION

makekey improves the usefulness of encryption schemes depending on a key by increasing the amount of time required to search the key space. It reads 10 bytes from its standard input, and writes 13 bytes on its standard output. The output depends on the input in a way intended to be difficult to compute (i.e., to require a substantial fraction of a second).

The first eight input bytes (the input key) can be arbitrary ASCII characters. The last two (the salt) are best chosen from the set of digits, ., /, and uppercase and lowercase letters. The salt characters are repeated as the first two characters of the output. The remaining 11 output characters are chosen from the same set as the salt and constitute the output key.

The transformation performed is essentially the following: the salt is used to select one of 4,096 cryptographic machines all based on the National Bureau of Standards DES algorithm, but broken in 4,096 different ways. Using the input key as key, a constant string is fed into the machine and recirculated a number of times. The 64 bits that come out are distributed into the 66 output key bits in the result.

makekey is intended for programs that perform encryption (e.g., ed(1) and crypt(1)). (The encryption scheme provided by them is not secure.) Usually, makekey's input and output will be pipes.

## EXAMPLES

The first line that follows invokes makekey, the second line is the input to makekey, and the third is the new key generated by makekey.

```
makekey
abcdefgh23
23xq5GyrhLTCA
```

## FILES

/usr/lib/makekey
Executable file

## SEE ALSO

crypt(1), ed(1), ex(1)

passwd(4) in *A/UX Programmer's Reference*

**NAME**

man — displays the named manual page entries

**SYNOPSIS**

man [-c] [-d] [-T*term*] [-w] [[*section*] *name*]...

**ARGUMENTS**

-c Causes man to invoke col(1). Note that col(1) is invoked automatically by man unless *term* is one of the standard terminal types: 300, 300s, 450, 37, 4000a, 382, 4014, tek, 1620, or X.

-d Searches the current directory, rather than /usr/catman; requires the full filename (for example, cu.1c, rather than just cu).

*name*

Specifies the name of the manual page you wish to display.

*section*

Specifies the section number of the manual page you wish to display. If *section* is not specified, the whole manual is searched for *name* and all occurrences of it are printed.

*Note:* If you specify a section number, only one section can be searched at a time.

This argument may be changed before each *name*.

-T*term*

Prints the entry as appropriate for terminal type *term*. For a list of recognized values of *term*, type help term2 . The default value of *term* is 450.

-w Prints on the standard output only the *pathnames* of the entries, relative to /usr/catman, or to the current directory for -d option.

**DESCRIPTION**

man locates and prints an entry in the *A/UX Command Reference*, the *A/UX System Administrator's Reference*, or the *A/UX Programmer's Reference*. The name (*name*) of the entry is entered in lowercase. The section number (*section*) number may not have a letter suffix.

The man command examines the environment variables $TERM and $PAGER (see environ(5)) and attempts to select options that adapt the output to the terminal being used. The $PAGER variable defaults to more if not set otherwise. The user may select pg with the appropriate options. The -T*term* option overrides the value of $TERM; in particular, one should use -Tlp when sending the output of man to a line printer.

**EXAMPLES**

The command:

```
man man
```

would reproduce this entry on the terminal, as well as any other entries named `man` that may exist in other sections of the manual.

The command:

```
man sync
```

searches through all sections to find the entry for `sync`. Since there is a `sync`(1) and a `sync`(2), both entries are provided. If you are looking only for the `sync` system call (found in section 2), specify the section number as follows:

```
man 2 sync
```

**LIMITATIONS**

The `man` command prints manual entries that were formatted by `nroff` and are printed using the correct terminal filters as derived from the `-T`*term* and `$TERM` settings.

Typesetting or other nonstandard printing of manual entries is not supported.

**FILES**

`/usr/bin/man`
: Executable file

`/usr/catman/?_man/man[1-8]/*`
: Preformatted manual entry files

**SEE ALSO**

`term`(4) in *A/UX Programmer's Reference*

## NAME

merge — merges three files into one

## SYNOPSIS

merge [-p] *file1 file2 file3*

## ARGUMENTS

*file1*
: Specifies the file into which the other files will be merged, if the -p option is not given.

*file2*
: Specifies a file to be merged into *file1*.

*file3*
: Specifies a file to be merged into *file1*.

-p
: Causes the results of the merge to go to the standard output. If this option is not given, the results go into *file1*.

## DESCRIPTION

merge incorporates all changes that lead from *file2* to *file3* into *file1*. merge is useful for combining separate changes to an original. Suppose *file2* is the original, and both *file1* and *file3* are modifications of *file2*. Then merge combines both changes.

An overlap occurs if both *file1* and *file3* have changes in a common segment of lines. merge prints information on how many overlaps occurred and includes both alternatives in the result. The alternatives are delimited as follows:

<<<<<<< *file1*
*lines in file1*
=======
*lines in file3*
>>>>>>> *file3*

If there are overlaps, the user should edit the result and delete one of the alternatives.

## NOTES

Author: Walter F. Tichy, Purdue University, West Lafayette, IN 47907.


## SEE ALSO

co(1), diff(1), diff3(1), rcsmerge(1)

**NAME**

mesg — permits or denies the receipt of messages

**SYNOPSIS**

mesg [*choice*]

**ARGUMENTS**

*choice*

Specifies whether you want messages printed or not. Replace *choice* with either y or n.

**DESCRIPTION**

mesg permits or denies receipt of messages sent by another user via write(1). With no argument, mesg reports the current permission state.

The mesg command with choice n forbids messages by revoking nonuser write permission on the user's terminal. The mesg command with argument y reinstates permission.

**EXAMPLES**

To change the permission to "yes," enter:

```
mesg y
```

The system displays:

```
Is Yes; Was No
```

or whatever is the current and former state of your message permission.

**STATUS MESSAGES AND VALUES**

Exit status is 0 if messages are receivable, 1 if not, 2 on error.

**FILES**

/bin/mesg

Executable file

/dev/tty*

File containing terminal types

**SEE ALSO**

talk(1N), write(1)

## NAME

mkdir — creates a directory

## SYNOPSIS

mkdir *dirname...*

## ARGUMENTS

*dirname*

Specifies the directory to be created.

## DESCRIPTION

mkdir creates specified directories in mode 777 (possibly altered by the user's umask (see sh(1), csh(1) and ksh(1)). Standard entries, ., for the directory itself, and .., for its parent, are made automatically. These and other directories beginning with . are not visible in listings unless you use the -a flag option to ls.

mkdir requires write permission in the parent directory.

## EXAMPLES

To create a directory called letters as a subdirectory of the current directory at the time you employ the command, enter:

```
mkdir letters
```

## STATUS MESSAGES AND VALUES

The mkdir command returns exit code 0 if all directories were successfully made; otherwise, it prints a diagnostic and returns nonzero.

## FILES

/bin/mkdir

Executable file

## SEE ALSO

chmod(1), csh(1), ksh(1), rm(1), sh(1)

**NAME**

mkshlib — creates a shared library

**SYNOPSIS**

mkshlib -s *specs* [-n] -t *target* [-h *host*]

**ARGUMENTS**

-h *host*
Specifies the name of the host shared library, *host*. If this option is not specified, then the command does not produce the host shared library.

-n Does not generate a new target shared library. Use this option if you want to update the host shared library only. You still need to include the -t option and the target library name, because the system needs a version of the target shared library in order to build the host shared library.

-s *specs*
Specifies the name of the shared-library specification file, which contains the information necessary to build the shared library. Its contents include a list of the object files to be included in the shared library, the branch-table specifications for the target library, the pathname directing where to install the target library, and the start addresses of text and data sections for the target library. This file includes initialization specifications for imported variables, if necessary. Imported variables are addresses external to the target shared library, such as the addresses of routines that the library can call upon. See the ''Description'' section for details on the shared-library specification file.

-t *target*
Specifies the name of the target shared library to be produced. The specification file includes the location at which the target library is to be installed. (See the description of the #target directive in ''Shared Library Specification File'' in the ''Description'' section.)

**DESCRIPTION**

mkshlib builds and maintains shared libraries. A shared library is similar in function to a normal, nonshared library. The primary differences become evident when the program is executed. More than one program can use the code in shared-library routines simultaneously. The executable code for a shared library is in Common Object File Format (COFF). The applications that call shared-library routines access the executable code by means of a special addressing structure that is provided at link-edit time.

(In contrast to the programs that use a shared library, each program that uses a nonshared library gets a private copy of any library routines required.)

The shared library consists of two files (two sublibraries) containing source archives and executable object files. These files are referred to as the host shared library (host) and the target shared library (target), respectively. The host and target sublibraries can be on different systems. A host shared library is an archive that provides information used during link-edit (see `ld`(1) and `ar`(4)). The name of the host shared library is included on the `cc` command line, just as it is for a nonshared library (see `cc`(1)). All operations that can be performed on a nonshared library can be performed on a host shared library.

The target shared library contains the executable code for all of the routines in the library and must be fully resolved. This library is brought into memory, if not already present there, during the execution of a program that calls it. The library attaches to a user's process during execution. The text section of target objects is shared by all processes using that target library, but each process gets its own copy of the data.

The user interface to `mkshlib` consists of command line arguments and a shared-library specification file (specs). The specification file provides information necessary to build the host and target shared libraries.

To build both sublibraries, provide both the *target* and *host* arguments. To build only the target library, do not provide a host name. (However, a host library is required to access the target library by means of the link-edit process. Presumably, you either have a usable host library or will build one separately.) If you want `mkshlib` to build only a new host shared library and reuse an existing target shared library, use the `-n` option. You must supply the *target* argument, even if you are only building the host.

To build the host and target files, `mkshlib` invokes other tools, such as the archiver, `ar`(1), the assembler, `as`(1), and the loader, `ld`(1).

The shared-library specification file contains all the information necessary to build both the host and target shared libraries. The file contains directive names and associated specification information. Directive names must be at the beginning of the line. Some directives have specification information on the same line, and some directives introduce multiple specifications on subsequent lines. Lines following such a directive are interpreted as specification lines for that directive, until another directive or the end of the file is encountered.

The following list describes the six possible directive names and explains how to use them. The directives, except for `#init`, can be given in any order in the specification file.

`##` *comment-text*
Specifies that the rest of the line is a comment. All comment text on that line is ignored. Comment lines may occur anywhere. Comments

are recommended, but optional.

`#address` *section address*
Specifies the starting address in the virtual address space at which to bind the specified section of the target shared library. Typically, address directives are provided for the `.text` and `.data` sections of the target library. Addresses must be on a 256 kilobyte (KB) boundary.

The `.bss` section is grouped with the `.data` section and does not require a starting address.

`#branch`
*branch-table-specification...*
Interprets all lines following the `#branch` directive as branch-table specifications until it encounters another directive. A specification file can contain only one `#branch` directive. The branch table built from these specifications consists of jump instructions to the specified functions.

Branch-table specification lines have this format:

*function position*

Give branch-table entries only to external functions. The position value is the relative location of the function name in the branch table. Each function can appear only once. The value of *position* for each *function* given is the position (or position range) of the name in the branch table. The value of *position* is a single integer, or a range of integers of the form *position1-position2*. Position values start with 1. Use each position value only once. You must account for all position values from 1 to the highest value.

When adding functions to an existing library, provide the new functions at higher positions than those they occupy in the existing branch table. Changing positions in an existing branch table renders that shared library unusable by previously linked applications.

A position range can also be used to reserve empty slots in the branch table for later use. Only the highest value of the range is associated with the function name. The remaining positions in the range can be used later for other functions.

`#init` *object*
*initialization...*
Specifies *object*, using the name of an object file that requires initialization code (because it uses an imported variable). Each object file that requires initialization must be specified. (If the shared library being built is completely self-contained, that is, if it uses no imported

variable, no `#init` directive is used because no initialization code is necessary.)

All `#init` directives must be placed after the `#objects` directive and its associated specifications in the specification file.

An `#init` directive is followed by one or more initialization specification lines pertaining to the object file, *object*, named in the directive. Each line following the directive is interpreted as a specification line until another directive is encountered. To specify each line of *initialization*, use the following format:

*import importptr*

Replace *import* with an imported variable. Replace *importptr* with a pointer defined in the object file named in the `#init` directive preceding the initialization line. For each initialization line specified by this method, initialization code is generated in this form:

```
importptr = &import;
```

The system sets the value of `importptr` to the absolute address of `import`.

`#objects`
*file...*

Specifies each entry of *file*, using the names of the object files that constitute the target shared library.

This directive can be specified only once per shared library specification file. The system interprets the lines following the directive as specifications of *file* until another directive is encountered.

`#target` *pathname*

Specifies the absolute pathname for the location of the target shared library on the target system. This pathname, copied into `a.out` files, tells the operating system where to find the target shared library when executing a file that uses it. The maximum length of *pathname* is 64 characters.

**FILES**

`/lib/*_s.a`
Host (archive) library file

`/shlib/*_s`
Target (executable) library file

`/tmp/`*pid-and-time*
Temporary directory

`/usr/bin/mkshlib`
Executable file

**SEE ALSO**

ar(1), as(1), cc(1), ld(1)

a.out(4), ar(4) in *A/UX Programmer's Reference*

"Shared Libraries" in *A/UX Programming Languages and Tools, Volume I*

**NAME**

mkstr — creates an error message file by massaging C source programs

**SYNOPSIS**

mkstr [-] *messagefile prefix file...*

**ARGUMENTS**

- Causes the error messages to be placed at the end of the specified message file for recompiling part of a large mkstred program.

*file* Specifies the file to be processed.

*messagefile*
Specifies the file into which the error messages are placed.

*prefix*
Specifies the prefix for the output file. The name of the output file consists of the prefix (*prefix*) and the original filename (*file*).

**DESCRIPTION**

mkstr is used to create files of error messages. Using it can make programs with large numbers of error diagnostics much smaller, and reduce system overhead in running the program as the error messages do not have to be constantly swapped in and out.

The mkstr program will process each specified *file*, placing a massaged version of the input file in a file whose name consists of the specified *prefix* and the original name.

To process the error messages in the source to the message file mkstr keys on the string error(" in the input stream. Each time it occurs, the C string starting at the " is placed in the message file followed by a newline character and a null character; the null character terminates the message so it can be easily used when retrieved, the newline character makes it possible to sensibly cat the error message file to see its contents. The massaged copy of the input file then contains a lseek pointer into the file, which can be used to retrieve the message, i.e.:

```
char efilname[] = "/usr/lib/pi_strings";
int  efil = -1;
```

You have to write the error-handling function yourself. The following is an example:

```
error(a1, a2, a3, a4)
{
        char buf[256];
        if (efil < 0) {
                efil = open(efilname, 0);
                if (efil < 0) {
```

```
oops:
                        perror(efilname);
                        exit(1);
                }
        }
        if (lseek(efil, (long) a1, 0) < 0L
        || read(efil, buf, 256) <= 0)
                goto oops;
        printf(buf, a2, a3, a4);
}
```

**EXAMPLES**

If the current directory has files `a.c` and `b.c`, then

```
mkstr exs x *.c
```

would create a new file `exs`, which holds all the error messages extracted from the source files `a.c` and `b.c`, as well as two new source files `xa.c` and `xb.c`, which no longer contain the extracted error messages.

**LIMITATIONS**

All the arguments except the name of the file to be processed are unnecessary.

**FILES**

`/bin/mkstr`
: Executable file

**SEE ALSO**

`cc`(1), `xstr`(1)

`lseek`(2) in *A/UX Programmer's Reference*

## NAME

mm — formats documents that contain nroff and mm macro formatting requests

## SYNOPSIS

mm [-12] [-c] [-e] [-E] [-t] [-T*tty-type*] [*file*]...

## ARGUMENTS

-12
: Indicates that the document is to be produced in 12-pitch. You can use this flag option when $TERM is set to one of 300, 300s, 450, and 1620. If you use this flag option, you must manually set the switch to 12 on the DASI 300 and 300s terminals.

-c Causes mm to invoke col; note that col is invoked automatically by mm unless $TERM is one of 300, 300s, 450, 37, 4000a, 382, 4014, tek, 1620, or X.

-e Causes mm to invoke neqn in a way that causes neqn to read the /usr/pub/eqnchar file. See eqnchar(5) for details.

-E Invokes the -e option of nroff.

*file* Specifies the file to be formatted.

-t Causes mm to invoke tbl.

-T*tty-type*
: Specifies the type (*tty-type*) of output terminal.

## DESCRIPTION

mm formats documents using nroff and the mm text-formatting macro package. The mm command has options to specify preprocessing by tbl, eqn, and neqn and postprocessing by various terminal-oriented output filters.

Using the options you select, mm generates the proper pipelines and the required flag options and arguments for nroff and the mm macros. Any other options that appear on the mm command line are passed to nroff as appropriate. You may use such options in any order, but you must put them before the *file* argument. If you do not specify any arguments, mm prints a list of its options.

The list of recognized values for the -T*tty-type* option follow. Replace *tty-type* with one of these values:

37 Prepares output for a TELETYPE+ Model 37, which is the default for nroff.

40/4
: Prepares output for a TELETYPE Model 40/4 using the -c option.

`43` Prepares output for a TELETYPE Model 43 using the `-c` option.

`450`
Prepares output for a DASI 450, which is the default for `mm`. This value for *tty-type* is equivalent to `-T1620`.

`450-12`
Prepares output for a DASI 450 in 12-pitch mode.

`300`
Prepares output for a DASI 300 terminal.

`300-12`
Prepares output for a DASI 300 in 12-pitch mode.

`300s`
Prepares output for a DASI 300S.

`300s-12`
Prepares output for a DASI 300S in 12-pitch mode.

`382`
Prepares output for a DTC-382.

`745`
Prepares output for a Texas Instrument 700 series terminal using the `-c` option. This value for *tty-type* is equivalent to `-T735`.

`832`
Prepares output for an Anderson Jacobson 832 printer using the `-c` option.

`2631`
Prepares output for an HP2631 printer using the `-c` option.

`2631-c`
Acts the same as `-T2631`, but uses compressed mode.

`2631-e`
Acts the same as `-T2631`, but uses expanded mode.

`4000a`
Prepares output for a TRENDATA 4000A.

`4014`
Prepares output for a TEKTRONIX 4014.

`8510`
Prepares output for a C. Itoh printer using the `-c` option.

`hp` Prepares output for a Hewlett-Packard HP262*x* or HP264*x* using the `-c` option. This value for *tty-type* is equivalent to `-T2621`, `-T2640`, and `-T2645`.

`lp` Prepares output for a device with no reverse or partial line motions or other special features using the `-c` option.

`tn300`
Prepares output for a Terminet 300 printer using the `-c` option.

`X` Prepares output for an EBCDIC line printer.

If you do not use the `-T`*tty-type* option, `mm` uses the value of the shell variable `$TERM` from the environment (see `profile`(4) and `environ`(5)) as the value of *tty-type*, if `$TERM` is set; otherwise, `mm` uses `450` as the value of *tty-type*. If you specify several terminal types, the last one takes precedence.

If you lie to `mm` about the kind of terminal its output is to be printed on, you get readily apparent or subtle garbage. If you redirect output to a file, use the `-T37` option and then use the appropriate terminal filter when you actually print the file.

When you specify a hyphen (`-`) on the command line instead of a filename, `mm` reads the standard input. Reading the standard input allows `mm` to be used as a filter, as shown here:

`cat` *file* `| mm -`

Specifying a filename in addition to a hyphen does not work.

**Options for `nroff`**

The `mm` command invokes `nroff` with the `-h` option, which causes `nroff` to assume that the terminal has tabs set every 8 character positions.

You can use the `-o`*list* option of `nroff` to specify ranges of pages to be output. Note, however, that if you invoke `mm` with `-o`*list* and one or more of the `-e`, `-t`, and `-` options, a harmless `broken pipe` diagnostic may be appear if the last page of the document is not specified in *list*.

If you use the `-s` option of `nroff` to stop between pages of output, use linefeed (rather than return or newline) to restart the output. The `-s` option of `nroff` does not work with the `-c` option of `mm` or if `mm` automatically invokes `col`. See the `-c` option described earlier.

**EXAMPLES**

Assuming that the shell variable `$TERM` is set to `450`, the two command lines below are equivalent:

```
mm -t -rC3 -12 file
tbl file | nroff -cm -T450-12 -h -rC3
```

**STATUS MESSAGES AND VALUES**

The mm command displays

```
mm: no input file
```

if none of the arguments is a readable file and mm is not used as a filter.

**FILES**

/bin/mm
: Executable file

/usr/pub/terminals
: File containing a list of terminals

**SEE ALSO**

checkmm(1), col(1), env(1), eqn(1), greek(1), mmt(1), nroff(1), tbl(1), troff(1)

profile(4), mm(5), term(5) in *A/UX Programmer's Reference*

"mm Reference" in *A/UX Text Processing Tools*

## NAME

mmt, mvt — typeset documents that contain troff and mm or mv macro-formatting requests

## SYNOPSIS

mmt [-a] [-D*dest*] [-e] [-g] [-p] [-t] [-T*tty-type*] [-z] [*file*]...

mvt [-a] [-D*dest*] [-e] [-g] [-p] [-t] [-T*tty-type*] [-z] [*file*]...

## ARGUMENTS

-a Sends the output to an ASCII terminal.

-D*dest*
Directs output by means of the device *dest*, using the tc(1) filter. The currently supported values for *dest* are 4014 and i10 for the TEKTRONIX 4014 terminal and an Imagen Imprint-10 device, respectively.

-e Invokes eqn in a way that causes eqn to read the /usr/pub/eqnchar file. See eqnchar(5) for details.

*file* Specifies the file to be typeset. If you specify a hyphen (-) on the command line instead of a filename, mmt and mvt read the standard input. Reading the standard input allows either of these commands to be used as a filter, as shown here:

```
cat file | mmt -
```

-g Invokes grap(1), which in turn calls pic.

-p Invokes pic(1).

-t Invokes tbl(1).

-T*tty-type*
Creates output for a particular destination device specified by *tty-type*. Appropriate output is created through the selection of an appropriate postprocessor. To create output for an APS-5 device or troff device, replace *tty-type* with aps or ptty. See daps(1) and troff(1) for details.

-z Invokes no output filter to process or redirect the output of troff.

## DESCRIPTION

mmt and mvt are front-ends for calling troff, just as mm is a front-end for calling nroff(1). The mmt command uses the mm macro package and has options that let you specify preprocessing by eqn, grap, pic, and tbl. The mvt command uses the mv macro package and has the same options.

The discussion that follows refers only to mmt, but also applies to mvt.

Using the options you specify, mmt generates the proper pipelines and the required arguments and flags for troff and the appropriate macros.

Options can be specified in any order, but they must appear before the *file* arguments. If you do not specify any arguments, mmt prints a list of its options.

You can use the -o*list* option of troff to specify ranges of pages to be output. Note, however, that if you invoke mmt with -o*list* and one or more of the -e, -g, -p, -t, and - options, a harmless broken pipe diagnostic may appear if the last page of the document is not specified in *list*.

**STATUS MESSAGES AND VALUES**

If none of the arguments is a readable file and the command is not being used as a filter, the following error message is displayed:

```
mmt: no input file
```

**LIMITATIONS**

Because the programs gcat and vpr are not supplied with A/UX, you cannot use vp and st as replacement values for *tty-type*.

**FILES**

/bin/mmt
: Executable file

/bin/mvt
: Executable file

**SEE ALSO**

daps(1), env(1), eqn(1), grap(1), mm(1), nroff(1), pic(1), tbl(1), tc(1), troff(1)

profile(4), environ(5), mm(5), mv(5) in *A/UX Programmer's Reference*

*A/UX Text Processing Tools*

## NAME

more, page — show the contents of a file in display-size chunks

## SYNOPSIS

more [-c] [-d] [-f] [-l] [-n] [-s] [-u] [+*linenumber*] [*file*]...

more [-c] [-d] [-f] [-l] [-n] [-s] [-u] [+/*pattern*] [*file*]...

page [-c] [-d] [-f] [-l] [-n] [-s] [-u] [+*linenumber*] [*file*]...

page [-c] [-d] [-f] [-l] [-n] [-s] [-u] [+/*pattern*] [*file*]...

## ARGUMENTS

+*linenumber*
: Causes more to start up at *linenumber*.

+/*pattern*
: Causes more to start up two lines before the line containing the regular expression *pattern*, if the input is from a file. If input is from a pipe, more starts on the line where the pattern was found.

-c
: Causes more to draw each page by beginning at the top of the screen and erasing each line just before it draws on it. This avoids scrolling the screen, making it easier to read while more is writing. This flag option will be ignored if the terminal does not have the ability to clear to the end of a line.

-d
: Causes more to prompt the user with the message

```
Hit space to continue, Rubout to abort
```

at the end of each screenful.

-f
: Causes more to count logical lines, rather than screen lines; that is, long lines are not folded. This flag option is recommended if nroff output is being piped through ul, since the latter may generate escape sequences. These escape sequences contain characters which would ordinarily occupy screen positions, but which do not print when they are sent to the terminal as part of an escape sequence. Thus more may think that lines are longer than they actually are, and, therefore, fold lines erroneously.

*file*
: Specifies the file to be displayed.

-l
: Causes more not to treat CONTROL-L (form feed) as special. If this flag option is not given, more will pause after any line that contains a CONTROL-L, as if the end of a screenful had been reached. Also, if a file begins with a form feed, the screen will be cleared before the file is printed.

-n
: Specifies an integer which is the size (in lines) of the window which more will use instead of the default.

`-s` Squeezes multiple blank lines from the output, producing only one blank line. Especially helpful when viewing `nroff` output, this option maximizes the useful information present on the screen.

`-u` Suppresses normal processing of underlining. `more` will handle underlining such as produced by `nroff` in a manner appropriate to the particular terminal; if the terminal can perform underlining or has a stand-out mode, `more` will output appropriate escape sequences to enable underlining or use stand-out mode for underlined information in the source file.

**DESCRIPTION**

`more` is a filter which allows examination of continuous text one screenful at a time on a CRT terminal. It normally pauses after each screenful, printing `--More--` at the bottom of the screen.

`page` functions similarly, except that the screen is cleared before each screenful is displayed (but only if a full screenful is displayed), and that *k*-1 rather than *k*-2 lines are printed in each screenful, where *k* is the number of lines the terminal can display.

If the user then presses RETURN, one more line is displayed. If the RETURN is preceded by an integer, that number becomes the new window size. If the user hits a space, another screenful is displayed. If a space is preceded by an integer, that number of lines is displayed. If the user presses `d` or CONTROL-D, 11 more lines (usually half a screenful) are displayed (a ''scroll''). If `d` or CONTROL-D is preceded by an integer, that number becomes the new scroll size.

The `more` program looks in the file `/etc/termcap` to determine terminal characteristics and to determine the default window size. On a terminal capable of displaying 24 lines, the default window size is 22 lines.

The `more` program looks in the environment variable `MORE` to preset any flags desired. For example, if you prefer to view files using the `-c` mode of operation, the `sh` command sequence

```
MORE='-c'; export MORE
```

or the `csh` command

```
setenv MORE -c
```

would cause all invocations of `more`, including invocations by programs such as `man` and `msgs`, to use this mode. (Note, however, that the `man` command also looks at the `PAGER` environment variable; see `man`(1).) Normally, the user will place the command sequence that sets up the `MORE` environment variable in the shell startup file `.login`, `.profile`, or `.cshrc`.

If more is reading from a file rather than a pipe, then a percentage is displayed along with the --More-- prompt. This gives the fraction of the file (in characters, not lines) that has been read so far.

Once inside more, other sequences may be typed when more pauses. The sequences and their effects are as follows (*i* is an optional integer argument, defaulting to 1) :

= Displays the current line number.

v Starts up the editor vi at the current line (does not work if the input to the program is from a pipe).

h Invokes help which provides a description of all the more commands.

*i*:n Skips to the *i*th next file given in the command line. (Skips to last file if *i* doesn't make sense.)

*i*:p Skips to the *i*th previous file given in the command line. If this command is given in the middle of printing out a file, then more goes back to the beginning of the file. If *i* doesn't make sense, more skips back to the first file. If more is not reading from a file, the bell rings and nothing more happens.

:f Displays the current filename and line number.

:q
:Q Exits from more (same as q or Q).

. Repeats the previous command.

*i*z Same as typing a space except that *i*, if present, becomes the new window size.

*i*s Skips *i* lines and prints a screenful of lines.

*i*f Skips *i* screenfuls and prints a screenful of lines.

*i*n Searches for the *i*th occurrence of the last regular expression entered.

q
Q Exits from more. The interrupt character may also be used.

*i*/*expr*
Searches for the *i*th occurrence of the regular expression *expr*. Terminated either by pressing RETURN or the ESCAPE key. If the input is a file (rather than a pipe), and there are fewer than *i* occurrences of *expr*, then the position in the file remains unchanged and an error message is printed. If the input is a file (rather than a pipe), and there are at least *i* occurrences of *expr*, a screenful is displayed, starting two lines before the place where the expression was found. If the input is a pipe and there are fewer than *i* occurrences of *expr*, an error message is printed and more exits (because the entire input stream

has been read). If the input is a pipe and there are at least *i* occurrences of *expr*, a screenful is displayed, starting on the line where the expression was found. The user's erase and kill characters may be used to edit the regular expression. Erasing back past the first column cancels the search command.

' Goes to the point from which the last search started. If no search has been performed in the current file, this command goes back to the beginning of the file. (Doesn't work if the input to the program is from a pipe.)

!*command*
Invokes a shell with *command*. Terminated either by pressing RETURN or the ESCAPE key.

Up to the time when the command character itself is given, the user may hit the line kill character to cancel the numerical argument being formed. In addition, the user may hit the erase character to redisplay the `--More--`(*xx*%) message.

:!*command*
Invokes a shell with *command*. (Same as !*command*).

CONTROL-L (^L)
Redraws the screen by pressing CONTROL-L (^L). (This doesn't work if the input to the program is from a pipe.)

Any time output is being sent to the terminal, the user may press the quit key (normally CONTROL-\). `more` will stop sending output, and will display the usual `--More--` prompt. The user may then enter one of the commands in the normal manner. Unfortunately, some output is lost when this is done, due to the fact that any characters waiting in the terminal's output queue are flushed when the quit signal occurs.

The terminal is set to `noecho` mode by this program so that the output can be continuous. What is typed will not show on the terminal, except for the `/` and `!` commands.

If the standard output is not a terminal, then `more` acts just like `cat`, except that a header is printed before each file (if there is more than one).

**EXAMPLES**

The command:

```
nroff -ms +2 doc.n | more
```

would show the `nroff` output on the terminal screen.

**FILES**

`/bin/more`
Executable file

`/bin/page`
Executable file

`/etc/termcap`
Terminal capabilities file

`/usr/lib/more.help`
Help information file

**SEE ALSO**

`cat`(1), `pg`(1)

`termcap`(4), `terminfo`(4) in *A/UX Programmer's Reference*

## NAME

mt — manipulates magnetic tape media

## SYNOPSIS

mt [-f*device-file*] *command* [*count*]

## ARGUMENTS

*command*

Specifies the command that you want to have performed. Supported commands are bsf, bsr, eof, format, fsf, fsr, offline, rewind, rewoffl, status, and weof. (For more information, see the "Description" section.)

*count*

Specifies the number of times that *command* should be performed. When no count value is supplied, the default is 1. (This value is ignored for the rewind and format commands).

-f*device-file*

Specifies the raw device file that addresses the desired device. Your choices include the device files in the directory /dev/rmt that reference a port or SCSI ID through which the tape drive is connected. Note that *device-file* must reference a raw (not block) tape device driver.

## DESCRIPTION

mt sends commands that you specify to manipulate a magnetic tape drive as moderated by the tape device interface mtio (see mtio(7)). If *device-file* is not specified with the -f option, the environment variable TAPE is used; if TAPE does not exist, mt attempts to use the device referenced by /dev/rmtc.

### Device Commands

The device commands that you may specify are listed following. You only need to specify as many characters as needed to uniquely identify a command within the set.

bsf

Back-spaces *count* files.

bsr

Back-spaces *count* records.

eof
weof

Writes *count* end-of-file markers at the current position on the tape.

format

Formats a tape cartridge. (*count* is ignored.) This value applies only to /dev/rmt/tc*x*[n] device files that represent the Apple Tape

Backup 40SC.

`fsf`
Forward-spaces *count* files.

`fsr`
Forward-spaces *count* records.

`offline`
`rewoffl`
Rewinds the tape and place the tape unit off-line (*count* is ignored).

`rewind`
Rewinds the tape. (*Count* is ignored.)

`status`
Prints status information about the tape unit.

*Note:* The number reported as the available space does not give any consideration to media defects, which could reduce the usable space. You should subtract about 5 percent of the total tape capacity to determine the ''usable capacity'' that allows for skippage over any defects.

**`fsr` and `bsr` ''Records''**

For the `fsr` and `bsr` commands, records are equivalent to 8192 KB for all tape drives. This is not true for 9-track tape drives, however. Other than those drives, all other tape drives use a logical block size of 8192 KB. Do not confuse the record size with the physical block size of the tape unit, which could be anywhere in the range of .5 KB (512 bytes) to 8192 KB.

For example, the number reported by the `status` command as the `maxblk` amount is equivalent to the maximum number of 8192 KB blocks on the tape media, unless a 9-track drive was queried.

**STATUS MESSAGES AND VALUES**

The `mt` program returns a 0 exit status when *count* invocations of *command* are successful, 1 if *command* was unrecognized, and 2 if any invocation of *command* failed.

**FILES**

`/dev/rmt/*`
Raw magnetic tape device files

**SEE ALSO**

`ioctl`(2), `environ`(5), `mtio`(7) in *A/UX Programmer's Reference*

## NAME

mv — moves or renames files

## SYNOPSIS

mv [-i] [-f] [-] *file1 file2*

mv [-i] [-f] [-] *file... directory*

## ARGUMENTS

- Interprets all the following arguments to mv as filenames. This allows filenames starting with minus.

*directory*
Specifies the directory into which the files will be placed.

-f Causes mv to use force. This flag option overrides any mode restrictions or the -i option.

*file* Specifies the file that will be moved into the directory (*directory*).

*file1*
Specifies the file that is to be renamed.

*file2*
Specifies the file that *file1* was renamed to.

-i Specifies interactive mode. Whenever a move is to supercede an existing file, the user is prompted by the name of the file followed by a question mark. If he answers with a line starting with y, the move continues. Any other reply prevents the move from occurring. The -f option overrides this option.

## DESCRIPTION

mv moves (changes the name of) one file (*file1*) to another file RI ( file2 ). If *file2* already exists, it is removed before *file1* is moved. If *file2* has a mode which forbids writing, mv prints the mode (see chmod(2)) and reads the standard input to obtain a line; if the line begins with y, the move takes place; if not, mv exits.

The mv command can also move one or more files (*file*), which can be plain files or directories, into a directory (*directory*) with their original filenames.

The mv command refuses to move a file onto itself.

## LIMITATIONS

If *file1* and *file2* lie on different file systems, mv must copy the file and delete the original. In this case the owner name becomes that of the copying process and any linking relationship with other files is lost.

**FILES**

`/bin/mv`
Executable file

**SEE ALSO**

cp(1), ln(1)

*See* mmt(1)

## NAME

ndx — creates a subject-page index for a document

## SYNOPSIS

ndx *subjfile formatter-command-line*

## ARGUMENTS

*formatter-command-line*

Creates the final form of the document. The syntax for *formatter-command-line* is:

*formatter* [*option*]... *file*...

*subjfile*

Specifies the list of subjects to be included in the index. Each subject must begin on a new line and have the following format:

*word1* [*word2*...][, *wordn*...]

## DESCRIPTION

ndx, given a list of subjects (*subjfile*), searches a specified document and writes a subject-page index to the standard output.

Some examples of a subject file are:

```
printed circuit boards
arrays
arrays, dynamic storage
Smith, W. P.
printed circuit boards, channel-oriented
Aranoff
University of Illinois
PL/I
```

The subject must start in column 1.

The following are examples of valid formatter command lines:

```
mm -Tlp files
nroff -mm -Tlp -rW60 file
troff -rB2 -Taps -r01.5i files
```

For more information about the formatter command line, see mm(1), mmt(1), nroff(1), and troff(1).

The document must include formatting commands for mm, nroff, or troff. The formatter command line tells ndx whether troff, nroff, mm, or mmt would be used to produce the final version of the document.

troff or mmt

Specifies troff as the formatting program.

`nroff` or `mm`
Specifies `nroff` as the formatting program.

The options are those that would be given to the `troff`, `nroff`, `mm`, or `mmt` command in printing the final form of the document and are necessary to determine the correct page numbers for subjects as they are located in the document. `ndx` does not actually cause the final version of the document to be printed. The author must create the document separately. The indexer, of course, should not be used until the document is complete and no further changes are expected.

**EXAMPLES**

The command:

```
ndx subjfile "nroff -mm -rW70 files" > indexfile
```

would produce a subject-page index for the document *files* and take its subjects from the list, *subjfile*. The page numbers would correspond to the document produced by:

```
nroff -mm -rW70 files
```

The command:

```
ndx subjfile "mm -rW60 -rN2 -rO0 ch1 ch2 ch3" > indexfile
```

would produce a subject-page index for the documents `ch1`, `ch2`, and `ch3`. The page numbers would correspond to the documents produced by:

```
mm -rW60 -rN2 -rO0 ch1 ch2 ch3
```

The command:

```
ndx subjfile "troff -rB2 -rW5i -rO1.5i -mm files" > indexfile
```

would produce a subject-page index for the document *file*. The page numbers would correspond to the document produced by:

```
troff -rB2 -rW5i -rO1.5i -mm files
```

**FILES**

`/usr/bin/ndx`
Executable file

**SEE ALSO**

`mm`(1), `mmt`(1), `nroff`(1), `subj`(1), `troff`(1)

**NAME**

neqn — formats mathematical text for nroff

**SYNOPSIS**

neqn [-d*xy*] [-f*n*] [-p*n*] [-s*n*] [-] [*file*]...

**ARGUMENTS**

- Causes neqn to read the standard input, if this option is specified as the last argument.

-d*xy*
Sets delimiters to *x* and *y* between .EQ and .EN. The left and right delimiters may be the same character. The dollar sign ($) is often used as such a delimiter. All text that is neither between delimiters nor between .EQ and .EN is passed through untouched.

*file* Specifies the file to be formatted. If this option is not given, neqn reads the standard input.

-f*n*
Specifies the font to be used. Replace *n* with the desired font.

-p*n*
Specifies the point size of the equation. Replace *n* with a point size. The legal point size numbers are:

```
6  7  8  9  10  11  12  14
16  18  20  22  24  28  36
```

-s*n*
Specifies the amount to reduce or enlarge the point size of the equation. The default size is 10. Replace *n* with the number in which you want the default size reduced or enlarged.

**DESCRIPTION**

neqn is a preprocessor for typesetting mathematical text on typewriter-like terminals. Normal usage is:

neqn [*option*]... [*file*]... | nroff [*option*]... | [*printer*]

Full details of use are given in eqn(1).

**FILES**

/bin/neqn
Executable file

**SEE ALSO**

eqn(1), mm(1), nroff(1), tbl(1)

eqnchar(5), mm(5) in *A/UX Programmer's Reference*

"eqn Reference" in *A/UX Text Processing Tools*

**NAME**

netstat — displays network status information

**SYNOPSIS**

netstat [-a] [-A] [-n] [-f *address-family*] [*kernel*] [*memory-interface*]

netstat [-h] [-i] [-m] [-n] [-r] [-s] [-f *address-family*] [*kernel*] [*memory-interface*]

netstat [-I *interface*] *interval* [*kernel*] [*memory-interface*]

netstat -I *interface* [-n] [*kernel*] [*memory-interface*]

**ARGUMENTS**

-a Displays the state of all sockets, including sockets that are used by server processes, using the default display.

-A Displays the address of any protocol control blocks associated with sockets as well as the default display. This option is used for debugging.

-f *address-family*
Limits the display of statistics or address control blocks to those specified by the value of *address-family*. You can use these address families: inet (for AF_INET) and unix (for AF_UNIX).

-h Displays the state of the ICMP host table, using the default display.

-i Displays the state of interfaces that have been configured into the kernel by autoconfig. Interfaces statically configured into the system but not assigned at the time the system starts up are not shown.

-I *interface*
Displays information about the interface specified by *interface* only. Possible values for *interface* include ae0, lo0, sl0, and sl1.

*interval*
Specifies, in seconds, an interval of time during which netstat accumulates and displays data continuously for the default interface, which is the first interface that autoconfig configured when it made the kernel.

*kernel*
Specifies a kernel other than the default, /unix.

-m Displays statistics recorded by the memory-management routines that manage a private pool of memory buffers.

*memory-interface*
Specifies a memory interface other than the default, /dev/kmem.

`-n` Causes `netstat` to display Internet addresses as numbers. If you do not specify this option, `netstat` interprets addresses and displays them symbolically. You can use this option with any option that causes the display of an Internet address.

`-r` Displays the routing tables. If you also specify the `-s` option, the `-r` option displays routing statistics instead.

`-s` Displays statistics on a per-protocol basis. The available protocols include UDP, TCP, ICMP, and IP.

## DESCRIPTION

`netstat` displays the contents of various network-related data structures in the kernel, including the active sockets for each protocol, interface information, and packet traffic.

### Default Display

When the `netstat` command is given with no options or with the `-a`, `-A`, or `-h` option, `netstat` uses its default display to present the resulting information for all sockets except those sockets used by server processes. The information includes the name of the protocol, the size of the send queue and the size of the receive queue in bytes, the local and remote addresses, and the internal state of the protocol. This format is the default display. If a socket's address specifies a network but no specific host address, address formats are of the form *host.port* or *network.port.* When the host and network addresses are known, `netstat` displays them symbolically according to `/etc/hosts` and `/etc/networks`, respectively. If a symbolic name for an address is unknown or if you specify the `-n` option, `netstat` displays the address numerically according to the address family. For more information on the format of Internet addresses, see `inet`(3N). An unspecified or wildcard address or port appears as an asterick (`*`).

### Routing Table Display

Use the `-r` option to show the available routes and their status. Each route consists of a destination host or network and a gateway to use in forwarding packets. The `Flags` field shows the state of the route ( `U` if ''up''), whether the route is to a gateway (`G`), and whether the route was created dynamically by a redirect (`D`). Direct routes are created for each interface attached to the local host; the `Gateway` field for such entries shows the address of the outgoing interface. The `Ref` field gives the current number of active uses of the route. Connection-oriented protocols usually preserve a single route for the duration of a connection, whereas connectionless protocols obtain a route while sending to the same destination. The `Use` field provides a count of the number of packets sent over that route. The `Interface` field indicates the network interface used for the route.

**Memory Statistics Display**

Use the `-m` option to show the number of buffers allocated to packet headers, socket structures, protocol control blocks, routing table entries, socket names and addresses, and interface addresses. The output also includes a summary of mapped pages in use, allocated interface pages, requests for memory delayed or denied, and the number of calls to the protocol drain routines.

**Interface Display**

Use the `-i` or `-I` option to show interface information. The `Mtu` field gives the maximum transmission unit of the interface. The `Ipkts` and `Opkts` fields give, respectively, the number of incoming and outgoing packets transmitted since the system was started. The `Ierrs` and `Oerrs` fields give, respectively, the number of incoming and outgoing errors that have occurred since the system was started. The `Coll` field gives the number of collisions that have occurred since the system was started.

**Protocol Display**

Use the `-s` option to produce a summary of protocol-specific information. For example, the output for the UDP protocol includes the number of incomplete headers, the number of bad data-length fields, the number of bad checksums, the number of received packets, the number of received big packets, and the number of socket overflows.

**Interval Display**

If you specify the *interval* argument, `netstat` displays five columns of summary information about the default interface since the system was started and five columns of summary information about all interfaces. Subsequent lines of output show values accumulated over the preceding interval. To stop the output, send an interrupt signal (`SIGINT`), usually CONTROL-C, to `netstat`. You can change the default interface by using the `-I` option.

**LIMITATIONS**

The notion of errors is ill-defined.

Collisions have a different meaning in the ICMP protocol than in the other protocols.

**FILES**

`/dev/kmem`
: Default memory interface file

`/unix`
: Default kernel directory

`/usr/bin/netstat`
: Executable file

**SEE ALSO**

hosts(4), networks(4N), protocols(4N), services(4N) in *A/UX Programmer's Reference*

trpt(1M) in *A/UX System Administrator's Reference*

**NAME**

newform — changes the format of a text file

**SYNOPSIS**

newform [-a*n*] [-b*n*] [-c*char*] [-e*n*] [-f] [-i*tabspec*] [-l*n*] [-o*tabspec*] [-p*n*] [-s] [*file*]...

**ARGUMENTS**

-a*n*
: Works the same as -p*n* except characters are appended to the end of a line.

-b*n*
: Truncates *n* characters from the beginning of the line when the line length is greater than the effective line length (see -l*n*). Default is to truncate the number of characters necessary to obtain the effective line length. The default value is used when -b with no *n* is used. This option can be used to delete the sequence numbers from a COBOL program as follows:

```
newform -l1 -b7 filename
```

: The -l1 must be used to set the effective line length shorter than any existing line in the file so that the -b option is activated.

-c*char*
: Changes the prefix/append character to *char*. Default character for *char* is a space.

-e*n*
: Works the same as -b*n* except that characters are truncated from the end of the line.

-f
: Writes the tab specification format line on the standard output before any other lines are output. The tab specification format line which is printed will correspond to the format specified in the last -o option. If no -o option is specified, the line which is printed will contain the default specification of -8.

*file*
: Specifies the file to be reformatted.

-i*tabspec*
: Inputs tab specification: expands tabs to spaces, according to the tab specifications given. *tabspec* recognizes all tab specification forms described in tabs(1). In addition, *tabspec* may be --, in which newform assumes that the tab specification is to be found in the first line read from the standard input (see fspec(4)). If no *tabspec* is given, *tabspec* defaults to -8. A *tabspec* of -0 expects no tabs; if any are found, they are treated as -1.

`-l`*n*
Sets the effective line length to *n* characters. If *n* is not entered, `-l` defaults to 72. The default line length is 80 characters. Note that tabs and backspaces are considered to be one character (use `-i` to expand tabs to spaces).

`-o`*tabspec*
Outputs tab specification: replaces spaces by tabs, according to the tab specifications given. The tab specifications are the same as for `-i`*tabspec*. If no *tabspec* is given, *tabspec* defaults to `-8`. A *tabspec* of `-0` means that no spaces will be converted to tabs on output.

`-p`*n*
Prefixes *n* characters (see `-c`*char*) to the beginning of a line when the line length is less than the effective line length. Default is to prefix the number of characters necessary to obtain the effective line length.

`-s` Shears off leading characters on each line up to the first tab and places up to eight of the sheared characters at the end of the line. If more than eight characters (not counting the first tab) are sheared, the eighth character is replaced by an * and any characters to the right of it are discarded. The first tab is always discarded.

An error message and program exit will occur if this option is used on a file without a tab on each line. The characters sheared off are saved internally until all other options specified are applied to that line. The characters are then added at the end of the processed line.

For example, to convert a file with leading digits, one or more tabs, and text on each line, to a file beginning with the text, all tabs after the first expanded to spaces, padded with spaces out to column 72 (or truncated to column 72), and the leading digits placed starting at column 73, the command would be:

```
newform -s -i -l -a -e filename
```

## DESCRIPTION

`newform` reads lines from the named *files*, or the standard input if no input file is named, and reproduces the lines on the standard output. Lines are reformatted in accordance with command line options in effect.

Except for the `-s` option, options may appear in any order, may be repeated, and may be intermingled with the optional *files*. Command options are processed in the order specified. This means that option sequences like `-e15 -l60` will yield results different from `-l60 -e15`.

## STATUS MESSAGES AND VALUES

All error messages cause newform to stop.

usage: ...
: newform was called with a bad option.

not -s format
: There was no tab on one line.

can't open file
: Self explanatory.

internal line too long
: A line exceeds 512 characters after being expanded in the internal work buffer.

tabspec in error
: A tab specification is incorrectly formatted, or specified tab stops are not ascending.

tabspec indirection illegal
: A *tabspec* read from a file (or standard input) may not contain a *tabspec* referencing another file (or standard input). The following exit values can be set:

  0 Specifies normal execution.

  1 Indicates any error.

## LIMITATIONS

The newform command normally only keeps track of printable characters; however, for the -i and -o options, newform will keep track of backspaces in order to line up tabs in the appropriate logical columns.

The newform command will not prompt the user if a *tabspec* is to be read from the standard input (by use of -i- or -o-).

If the -f option is used, and the last -o option specified was -o-, and was preceded by either a -o- or a -i-, the tab specification format line will be incorrect.

## FILES

/bin/newform
: Executable file

## SEE ALSO

csplit(1), tabs(1)

fspec(4) in *A/UX Programmer's Reference*

**NAME**

newgrp — logs you into a new group

**SYNOPSIS**

newgrp [-] [*group*]

**ARGUMENTS**

- Changes the environment to what would be expected if the user actually logged in again.

*group*

Specifies the group you wish to be a part of.

**DESCRIPTION**

newgrp changes a user's group identification. The user remains logged in, and the current directory is unchanged, but calculations of access permissions to files are performed with respect to the new real and effective group IDs. The user is always given a new shell, replacing the current shell, by newgrp, regardless of whether it terminated successfully or terminated due to an error condition (that is, unknown group).

Exported variables retain their values after invoking newgrp; however, all unexported variables are either reset to their default value or set to null. System variables (such as PS1, PS2, PATH, MAIL, and HOME), unless exported by the system or explicitly exported by the user, are reset to default values. For example, a user has a primary prompt string (PS1) other than $ (default) and has not exported PS1. After an invocation of newgrp, successful or not, their PS1 will now be set to the default prompt string $. Note that the shell command export (see sh(1)) is the method used to export variables so that their assigned value is retained when invoking new shells.

With no arguments, newgrp changes the group identification back to the group specified in the user's password file entry.

A password is demanded if the group has a password and the user does not, or if the group has a password and the user is not listed in /etc/group as being a member of that group.

**EXAMPLES**

The command:

```
newgrp grpnam
```

would set the user's group ID to that of the group named grpnam.

**LIMITATIONS**

There is no convenient way to enter a password into /etc/group. Use of group passwords is not encouraged, because, by their very nature, they encourage poor security practices. Group passwords may disappear in the

future.

**FILES**

`/bin/newgrp`
Executable file

`/etc/group`
File containing a list of groups

`/etc/passwd`
File containing a list of passwords

**SEE ALSO**

`login`(1), `sh`(1)

`group`(4), `passwd`(4), `environ`(5) in *A/UX Programmer's Reference*

**NAME**

news — displays local news items

**SYNOPSIS**

news [-a] [-n] [-s] [*items*]

**ARGUMENTS**

-a Displays all items, regardless of currency. In this case, the stored time is not changed.

-n Reports the names of the current items without displaying their contents, and without changing the stored time.

-s Reports how many current items exist, without displaying their names or contents, and without changing the stored time. It is useful to include such an invocation of news in one's .profile file, or in the system's /etc/profile.

*items*

Specifies the news items that are to be displayed.

**DESCRIPTION**

news is used to keep the user informed of current events. By convention, these events are described by files in the directory /usr/news.

When invoked without arguments, news displays the contents of all current files in /usr/news, most recent first, with each preceded by an appropriate header. The news program stores the ''currency'' time as the modification date of a file named .news_time in your home directory (the identity of this directory is determined by the environment variable $HOME); only files more recent than this currency time are considered ''current.''

If the interrupt character (usually CONTROL-c) is pressed during the display of a news item, the display stops and the next item is started. Another interrupt within one second of the first causes the program to terminate.

**EXAMPLES**

The command:

```
news
```

will display all files in /usr/news that have not been read previously by the account owner.

**FILES**

/bin/news
Executable file

/etc/profile
Executable file

`/usr/news/*`
Files containing news items
`$HOME/.news_time`
Executable file

**SEE ALSO**
`profile`(4), `environ`(5) in *A/UX Programmer's Reference*

**NAME**

nice — executes a command at low priority

**SYNOPSIS**

nice [*-increment*] *command* [*arguments*]

**ARGUMENTS**

*-increment*
Specifies the increment of the command. Replace *increment* with a number between 1 and 19. If this argument is not specified, an increment of 10 is assumed. A negative increment (such as -10) enables the superuser to run commands with priority higher than normal.

*arguments*
Specifies the arguments for the specified command.

*command*
Specifies the command to be executed.

**DESCRIPTION**

nice executes *command* with a lower CPU scheduling priority. The nice command built into the C shell is different from /bin/nice, which can be used by any shell.

**EXAMPLES**

For the Bourne shell (sh) or Korn shell (ksh):

```
nice -10 date
```

would cause the program date to be processed at a priority lower than normal (0), i.e., at +10. In the C shell (csh), the same is achieved by typing in

```
nice +10 date
```

**LIMITATIONS**

An *increment* larger than 19 is equivalent to 19.

**STATUS MESSAGES AND VALUES**

The nice program returns the exit status of the subject command.

**FILES**

/bin/nice
Executable file

**SEE ALSO**

csh(1), ksh(1), nohup(1), sh(1)

nice(2) in *A/UX Programmer's Reference*

**NAME**

nl — processes a file through a line numbering filter

**SYNOPSIS**

nl [-b*type*] [-d*delim*] [-f*type*] [-h*type*] [-i*incr*] [-l*num*] [-n*format*] [-p] [-s*sep*] [-v*start#*] [-w*width*] *file*

**ARGUMENTS**

-b*type*

Specifies which logical page body lines are to be numbered. Recognized *type*s and their meanings are:

a number all lines;

t numbers the lines with printable text only;

n no line numbering;

p*string*

numbers only the lines that contain the regular expression specified in *string*. Default *type* for logical page body is t (text lines numbered).

-d*delim*

Specifies the delimiter characters indicating the start of a logical page section may be changed from the default characters (\:) to two user-specified characters. If only one character is entered, the second character remains the default character (:). No space should appear between the -d and the delimiter characters. To enter a backslash, use two backslashes.

-f*type*

Specifies which logical footer lines are to be numbered. The default for logical page footer is n (no lines numbered).

*file* Specifies the file to be read. If this argument is not given, the standard input is read.

-h*type*

Specifies which logical header lines are to be numbered. The default *type* for logical page header is n (no lines numbered).

-i*incr*

Numbers logical page lines with the increment value of *incr*. The default is 1.

-l*num*

Specifies the number of blank lines, *num*, to be considered as one. For example, -l2 results in only the second adjacent blank being numbered (if the appropriate -ha, -ba, and/or -fa option is set). The default is 1.

-n*format*
Specifies the line numbering format. The recognized values are:

`ln` left justified, leading zeroes suppressed;

`rn` right justified, leading zeroes suppressed;

`rz` right justified, leading zeroes kept.

The default line numbering format is `rn` (right justified).

`-p` Does not restart numbering at logical page delimiters.

-s*sep*
Separates the line number and the corresponding text line with *sep*. The default character is a tab.

-v*start#*
Numbers logical page lines with the initial value *start#*. The default is `1`.

-w*width*
Specifies *width* as the number of characters to be used for the line number. The default width is 6.

## DESCRIPTION

`nl` reads lines from the named *file* and reproduces the lines on the standard output. Lines are numbered on the left in accordance with the command options in effect.

The `nl` program views the text it reads in terms of logical pages. Line numbering is reset at the start of each logical page. A logical page consists of a header, a body, and a footer section. Empty sections are valid. Different line numbering options are independently available for header, body, and footer (for example, no numbering of header and footer lines while numbering blank lines only in the body).

The start of logical page sections are signaled by input lines containing nothing but the following delimiter character(s):

| *Line contents* | *Start of* |
|---|---|
| `\:\:\:` | header |
| `\:\:` | body |
| `\:` | footer |

Unless otherwise specified, `nl` assumes the text being read is in a single logical page body.

## EXAMPLES

The command:

```
nl -v10 -i10 -d!+ file1
```

will number `file1` starting at line number ten with an increment of ten. The logical page delimiters are !+.

**FILES**

`/bin/nl`
Executable file

**SEE ALSO**

`awk`(1), `cat`(1), `pr`(1), `sed`(1)

**NAME**

nm — displays the symbol table of a common object file

**SYNOPSIS**

nm [-d] [-e] [-f] [-h] [-n] [-o] [-T] [-u] [-v] [-V] [-x] *file...*

**ARGUMENTS**

-d Prints the value and size of a symbol in decimal (the default).

-e Prints only static and external symbols.

-f Produces full output. Redundant symbols (.test, .data, .bss), normally suppressed, are printed.

*file* Specifies the common object file.

-h Does not display the output header data.

-n Sorts external symbols by name before they are printed.

-o Prints the value and size of a symbol in octal instead of decimal.

-T Truncates long names. By default, nm prints the entire name of the symbols listed. Since object files can have symbol names with an arbitrary number of characters, a name that is longer than the width of the column set aside for names will overflow, forcing every column after the name to be misaligned. The -T option causes nm to truncate every name which would otherwise overflow its column and place an asterisk as the last character in the displayed name to mark it as truncated.

-u Prints undefined symbols only.

-v Sorts external symbols by value before they are printed.

-V Prints the version of the nm command executing on the standard error output.

-x Prints the value and size of a symbol in hexadecimal instead of decimal.

**DESCRIPTION**

nm displays the symbol table of each common object file *filename*. The argument, *filename*, may be a relocatable or absolute common object file, or it may be an archive of relocatable or absolute common object files. The nm program prints the following information for each symbol. Note that the object file must have been compiled with the -g option of the cc(1) command for there to be *type*, *size*, or *line* information.

*name*

The name of the symbol.

*value*
Its value expressed as an offset or an address depending on its storage class.

*class*
Its storage class.

*tv* If the symbol is accessed through a transfer vector, this field contains *tv*.

*type* Its type and derived type. If the symbol is an instance of a structure or a union, the structure or union tag is given following the type (e.g., *struct-tag*). If the symbol is an array, the array dimensions are given following the type (e.g., `char[n] [m]`).

*size* Its size in bytes, if available.

*line* The source line number at which it is defined, if available.

*section*
For storage classes static and external, the object file section containing the symbol (e.g., text, data, or bss).

Options may be used in any order, either singly or in combination, and may appear anywhere in the command line. Therefore, both `nm` *name* `-e -v` and `nm -ve` *name* print the static and external symbols in *name*, with external symbols sorted by value.

**WARNINGS**

When all the symbols are printed, they must be printed in the order they appear in the symbol table in order to preserve scoping information. Therefore, the `-v` and `-n` options should be used only in conjunction with the `-e` option.

**STATUS MESSAGES AND VALUES**

`nm: `*name*`: cannot open`
*name* cannot be read.

`nm: `*name*`: bad magic`
*name* is not an appropriate common object file.

`nm: `*name*`: no symbols`
The symbols have been stripped from *name*.

**FILES**

`/bin/nm`
Executable file

**SEE ALSO**

as(1), cc(1), ld(1)

a.out(4), ar(4) in *A/UX Programmer's Reference*

## NAME

nohup — runs a command so that it can continue to run even after your session has ended

## SYNOPSIS

nohup *command-line* &

## ARGUMENTS

*command-line*
Specifies a command line.

&
Specifies background mode.

## DESCRIPTION

nohup executes *command-line* in such a way that it does not terminate when an end-of-transmission (EOT, CONTROL-D) signal is received from the controlling terminal. The end-of-transmission signal is also known as a hangup signal.

With nohup, the priority is automatically incremented by 5. The nohup program should be used with processes running in the background (with &) in order to prevent them from responding to interrupts or stealing the input from the next person who logs in on the same terminal. In csh, processes run in the background are automatically immune to hangup signals.

If output is not redirected by the user, both the standard output and standard error output are sent to a file named nohup.out. If nohup.out is not writable in the current directory, output is redirected to $HOME/nohup.out.

## EXAMPLES

The command:

```
nohup  nroff -mm  docsfile | lp &
```

runs the nroff command shown, immune to hangups, quits, and interrupts.

To apply nohup to pipelines or lists of commands, you need to create a shell script so that multiple commands can be run through a single filename reference. For example, when you enter

```
nohup sh batchfile &
```

the processing affected by nohup includes all of the commands inside of batchfile. To permit the running of this script more simply as

```
nohup file &
```

you need to establish execute permission for batchfile by using chmod (see chmod(1)). For more information about background processing and scripts, see csh(1), ksh(1), and sh(1).

**WARNINGS**

Do not expect the reach of `nohup` to extend to commands after the first in a series, such as

`nohup` *command1*`;` *command2*

Each of the following command lines is one possible remedy for this kind of problem:

`nohup (`*command1*`;` *command2*`)`
`nohup` *command1*`;` `nohup` *command2*

If you are not careful when you redirect the standard error output, you can create unexpected problems. Any error messages produced by the following command are sent to the same disk used for the archive, possibly corrupting the archive:

```
nohup cpio -o < list > /dev/dsk/c8d0s0 2>&1 &
```

To avoid possible corruption of the archive, redirect the error output to some other place, such as a file named `errors`:

```
nohup cpio -o < list > /dev/dsk/c8d0s0 2>errors &
```

**FILES**

`./nohup.out`
Default file where standard output and standard error output are sent

`/bin/nohup`
Executable file

**SEE ALSO**

`chmod`(1), `csh`(1), `ksh`(1), `nice`(1), `sh`(1)

`nice`(2), `signal`(3) in *A/UX Programmer's Reference*

## NAME

nroff — text formatter

## SYNOPSIS

nroff [-e] [-h] [-i] [-m*name*] [-n*start-no*] [-o*page-range*] [-q] [-r*letter*[*value*]] [-s[*pages-per-pause*]] [-T*tty-type*] [-u[*boldening-amt*]] [-z] [*file*]...

## ARGUMENTS

-e Produces equally spaced words in adjusted lines, using the full resolution of the particular terminal.

*file* Specifies the file to be formatted.

-h Uses output tabs during horizontal spacing to speed output and reduce output character count. Tab settings are assumed to be every 8 nominal character widths.

-i Reads standard input after *file*s are exhausted.

-m*name*
Prepends to the input *file*s the macro file /usr/lib/tmac/tmac.*name*.

-n*start-no*
Numbers the first generated page *start-no*.

-o*page-range*
Prints only pages whose page numbers appear in the *page-range* of numbers and ranges, separated by commas. A range *x*-*y* means pages *x* through *y*; a range given by -*y* means from the beginning to page *y*; a range given by *x*- means from page *x* to the end. (See LIMITATIONS below.)

-q Invokes the simultaneous input-output mode of the .rd request.

-r*letter*[*value*]
Sets the number register referenced by *letter* to *integer*.

-s[*pages-per-pause*]
Specifies the number of pages to print between pauses, causing nroff to halt to allow paper loading or changing. Printing resumes upon receipt of a linefeed or newline (newlines do not work in pipelines, e.g., with mm(1)). The default is 1. This option does not work if the output of nroff is piped through col(1). When nroff (otroff) halts between pages, an ASCII BEL is sent to the terminal.

-T*tty-type*
Prepares the output for the specified terminal. The known *tty-type*s are:

| | |
|---|---|
| 2631 | Hewlett-Packard 2631 printer in regular mode |
| 2631-c | Hewlett-Packard 2631 printer in compressed mode |
| 2631-e | Hewlett-Packard 2631 printer in expanded mode |
| 300 | DASI-300 printer |
| 300-12 | DASI-300 terminal set to 12-pitch (12 characters per inch) |
| 300s | DASI-300s printer (300s is a synonym) |
| 300s-12 | DASI-300s terminal set to 12-pitch (12 characters per inch) (300s-12 is a synonym) |
| 37 | TELETYPE Model 37 terminal (default) |
| 382 | DTC-382 |
| 4000a | Trendata 4000a terminal (4000a is a synonym) |
| 450 | DASI-450 (Diablo Hyterm) printer |
| 450-12 | DASI-450 (Diablo Hyterm) printer set to 12-pitch (12 characters per inch) |
| 832 | Anderson Jacobson 832 terminal |
| 8510 | C.ITOH printer |
| lp | generic name for printers that can underline and tab (All text using reverse linefeeds, such as those having tables, that is sent to lp must be processed with col(1) ) |
| tn300 | GE Terminet 300 terminal |
| X | Printers equipped with TX print train |

-u[*boldening-amt*]
Sets the emboldening factor (number of character overstrikes) for the third font position (bold) to *boldening-amt*, or to zero if *boldening-amt* is missing.

-z Prints only messages generated by .tm (terminal message) requests.

**DESCRIPTION**

nroff formats text contained in *files* (standard input by default) for printing on typewriter-like devices and line printers.

An argument consisting of a minus (-) is taken to be a filename corresponding to the standard input.

**LIMITATIONS**

The nroff program believes in Eastern Standard Time; as a result, depending on the time of the year and on your local time zone, the date that nroff generates may be off by one day from your idea of what the date is.

When nroff is used with the -o option inside a pipeline (e.g., with one or more of neqn(1), and tbl(1)), it may cause a harmless ''broken pipe'' diagnostic if the last page of the document is not specified in *page-range*.

**FILES**

/bin/nroff
: Executable file

/usr/lib/tmac/tmac.*
: Standard macro files

/usr/lib/macros/*
: Standard macro files

/usr/lib/nterm/*
: Files containing terminal driving tables for nroff

/usr/pub/terminals
: File containing a list of supported terminals

**SEE ALSO**

checknr(1), col(1), deroff(1), greek(1), mm(1), neqn(1), tbl(1)

mm(5) in *A/UX Programmer's Reference*

''nroff/troff Reference'' in *A/UX Text Processing Tools*

## NAME

nslookup — interactively queries name servers

## SYNOPSIS

nslookup

nslookup *-server*

nslookup *host-to-find* [*server*]

## ARGUMENTS

*-server*
Specifies interactive mode. Replace *server* with the host name of a name server.

*host-to-find*
Specifies the name of the host to be queried.

*server*
Specifies either the host name or the address for a name server.

## DESCRIPTION

nslookup is a program which queries DARPA Internet domain name servers.

The nslookup program has two modes: interactive and non-interactive. Interactive mode allows the user to query the name server for information about various hosts and domains or print a list of hosts in the domain. Non-interactive mode is used to print just the name and Internet address of a host or domain.

Interactive mode is entered in the following cases:

a) when no arguments are given (the default name server will be used), and

b) when the first argument is a hyphen (-) and the second argument is the host name of a name server.

Non-interactive mode is used when the name of the host to be looked up is given as the first argument. The optional second argument specifies a *server*.

### Interactive commands

Commands may be interrupted at any time by typing a CONTROL-C. To exit, enter the end-of-file signal, CONTROL-D. The command line length must be less than 80 characters.

*Note:* an unrecognized command will be interpreted as a host name.

*host* [*server*]
Looks up information for *host* using the current default server, or using *server* if it is specified.

`server` *domain*
`lserver` *domain*
Changes the default server to *domain*. `lserver` uses the initial server to look up information about *domain* while `server` uses the current default server. If an authoritative answer can't be found, the names of servers that might have the answer are returned.

`root`
Changes the default server to the server for the root of the domain name space. Currently, the host `sri-nic.arpa` is used. (This command is a synonym for the `lserver`sri-nic.arpa.) The name of the root server can be changed with the `set root` command.

`finger` [*name*] [>*filename*]
`finger` [*name*] [>>*filename*]
Connects with the finger server on the current host. The current host is defined when a previous look-up for a host was successful and returned address information (see the `set querytype=A` command). *name* is optional. > and >> can be used to redirect output in the usual manner.

`ls` *domain* [>*filename*]
`ls` *domain* [>>*filename*]
`ls -a` *domain* [>*filename*]
`ls -a` *domain* [>>*filename*]
`ls -h` *domain* [>*filename*]
`ls -h` *domain* [>>*filename*]
Lists the information available for *domain*. The default output contains host names and their Internet addresses. The `-a` option lists aliases of hosts in the domain. The `-h` option lists CPU and operating system information for the domain. When output is directed to a file, hash marks are printed for every 50 records received from the server.

`view` *filename*
Sorts and lists the output of the `ls` command with `more`(1).

`help`
`?` Prints a brief summary of commands.

`set` *keyword*[=*value*]
Changes state information that affects the look-ups. Valid keywords are:

`all`
Prints the current values of the various options to `set`. Information about the current default server and host is also printed.

[no]debug
Turns debugging mode on. A lot more information is printed about the packet sent to the server and the resulting answer.
(Default = nodebug, abbreviation = [no]deb)

[no]defname
Appends the default domain name to every look-up.
(Default = nodefname, abbreviation = [no]def)

domain=*name*
Changes the default domain name to *name*. The default domain name is appended to all look-up requests if the *defname* option has been set.
(Default = value in /etc/resolv.conf, abbreviation = do)

querytype=*value*
Changes the type of information returned from a query to one of:

A Specifies the host's Internet address (the default).

CNAME
Specifies the canonical name for an alias.

HINFO
Specifies the host CPU and operating system type.

MD Specifies the mail destination.

MX Specifies the mail exchanger.

MG Specifies the mail group member.

MINFO
Specifies the mailbox or mail list information.

MR Specifies the mail rename domain name.

Other types specified in the RFC883 document are valid but aren't very useful.
(Abbreviation = q)

[no]recurse
Tells the name server to query other servers if it does not have the information.
(Default = recurse, abbreviation = [no]rec)

retry=*number*
Sets the number of retries to *number*. When a reply to a request is not received within a certain amount of time (changed with set timeout), the request is resent. The

retry value controls how many times a request is resent before giving up.
(Default = 2, abbreviation = ret)

root=*host*
Changes the name of the root server to *host*. This affects the root command.
(Default = sri-nic.arpa, abbreviation = ro)

timeout=*number*
Changes the time-out interval for waiting for a reply to *number* seconds.
(Default = 10 seconds, abbreviation = t)

[no]vc
Always uses a virtual circuit when sending requests to the server.
(Default = novc,abbreviation= [no]v)

**Tutorial**

The domain name space is tree-structured and currently has five top-level domains:

com
for commercial establishments

edu
for educational institutions

gov
for government agencies

org
for not for profit organizations

mil
for MILNET hosts

If you are looking for a specific host, you need to know something about the host's organization in order to determine the top-level domain it belongs to. For instance, if you want to find the Internet address of a machine at UCLA, do the following:

a) Connect with the root server using the root command. The root server of the name space has knowledge of the top-level domains.

b) Because UCLA is a university, its domain name is ucla.edu. Connect with a server for the ucla.edu domain with the command server ucla.edu. The response will print the names of hosts that act as servers for the domain ucla.edu. Note that the root server does not have information about ucla.edu but knows the names and

addresses of hosts that do. All future queries will be sent to the UCLA name server.

c) To request information about a particular host in the domain, type the host name. To request a listing of hosts in the UCLA domain, use the `ls` command. The `ls` command requires a domain name (in this case, `ucla.edu`) as an argument.

Note that if you are connected with a name server that handles more than one domain, all look-ups for host name must be fully specified with its domain. For instance, the domain `harvard.edu` is served by `seismo.css.gov`, which also services the `css.gov` and `cornell.edu` domains. A look-up request for the host `aiken` in the `harvard.edu` domain must be specified as `aiken.harvard.edu`. However, the `set domain=`*name* and `set defname` commands can be used to automatically append a domain name to each request.

After a successful look-up of a host, use the `finger` command to see who is on the system or to finger a specific person. To get other information about the host, use the `set querytype=`*value* command to change the type of information desired and request another look-up. (The `finger` requires *value* to be `A`.)

**STATUS MESSAGES AND VALUES**

If the look-up request was not successful, an error message is printed. Possible errors are:

`Time-out`
: The server did not respond to a request after a certain amount of time (changed with `set timeout=`*value* ) and a certain number of retries (changed with `set retry=`*value*).

`No information`
: Depending on the query type set with the `set querytype` command, no information about the host was available, though the host name is valid.

`Non-existent domain`
: The host or domain name does not exist.

`Connection refused`
`Network is unreachable`
: The connection to the name or finger server could not be made at the current time. This error commonly occurs with `finger` requests.

`Server failure`
: The name server found an internal inconsistency in its database and could not return a valid answer.

`Refused`
The name server refused to service the request.

`Format error`
The name server found that the request packet was not in the proper format. This error should not occur. It would indicate a bug in the program.

**FILES**

`/etc/bind/tools/nslookup`
Executable file

`/etc/resolv.conf`
File containing initial domain name and name server addresses

**SEE ALSO**

`named`(1M) in *A/UX System Administrator's Reference*

`resolver`(4) in *A/UX Programmer's Reference*

RFC-882, RFC-883 (DNN Network Information Center, SRI International)

## NAME

od — converts binary data to a displayable form in octal, decimal, hexadecimal, or ASCII

## SYNOPSIS

od [-b] [-c] [-d] [-o] [-s] [-x] [*file*] [[+]*offset* [.] [b]]

## ARGUMENTS

[[+]*offset* [.] [b]]
: Specifies the offset in the file where dumping is to commence. This argument is normally interpreted as octal bytes. If . is appended, the offset is interpreted in decimal. If b is appended, the offset is interpreted in blocks of 512 bytes. If the file argument is omitted, the offset argument must be preceded by +.

-b
: Interprets bytes in octal.

-c
: Interprets bytes in ASCII. Certain nongraphic characters appear as C escapes: null=\0, backspace=\b, form-feed=\f, newline=\n, return=\r, tab=\t; others appear as 3-digit octal numbers.

-d
: Interprets words in unsigned decimal.

*file*
: Specifies the file that is to be dumped. If no file argument is specified, the standard input is used.

-o
: Interprets words in octal.

-s
: Interprets words in signed decimal.

-x
: Interprets words in hex.

## DESCRIPTION

od dumps *file* in one or more formats as selected by the first argument. If the first argument is missing or an illegal flag option is specified, the -o option is default.

Dumping continues until end-of-file. If a file contains many lines of repeating characters, od represents the repeating lines with an asterisk.

## EXAMPLES

The command:

```
od -d file +2
```

produces an octal dump of file divided up into 32-bit words expressed in decimal equivalents with the dump starting point offset by 2 octal bytes.

## FILES

/bin/od
: Executable file

**SEE ALSO**

adb(1), dump(1), nm(1), strings(1)

**NAME**

otroff — formats text for a specific phototypesetter

**SYNOPSIS**

otroff [-c*name*] [-b] [-f] [-k*name*] [-m*name*] [-p*point-size*] [-t] [-w] [*file*]...

**ARGUMENTS**

-c*name*
Inserts before the input *file*s the compacted macro files:

/usr/lib/macros/cmp.[nt].[dt].*name*
/usr/lib/macros/ucmp.[nt].*name*

-b Reports whether the phototypesetter is busy or available. No text processing is done.

-f Refrains from feeding out paper and stopping phototypesetter at the end of the run.

*file* Specifies the file to be formatted. If this argument is not given, the standard input is read. An argument consisting of a single minus (-) is taken to be a filename corresponding to the standard input.

-k*name*
Compacts the macros used in this invocation of otroff, placing the output in files [dt].*name* in the current directory.

-m*name*
Prepends the named macro definition file to the input stream. The location of the macro file is the pathname constructed as follows:

/usr/lib/tmac/tmac.*name*

-p*point-size*
Specifies the point size to use for type. It still preserves all prescribed spacings and motions, reducing phototypesetter elapsed time.

-t Directs output to the standard output, instead of the phototypesetter (this is the default).

-w Waits until the phototypesetter is available, if it is currently busy.

**DESCRIPTION**

otroff is the old version of troff(1). It formats text contained in *file*s for standard output. The output is formatted for a Wang C/A/T phototypesetter.

**EXAMPLES**

The command:

otroff -mm *file*

formats the text contained in *file*, and invokes the macro package mm.

**FILES**

/bin/otroff
: Executable file

/usr/lib/suftab
: File containing suffix hyphenation tables

/tmp/ta$#
: Temporary file

/tmp/trtmp*
: Temporary file

/usr/lib/tmac/tmac.*
: Standard macro files

/usr/lib/macros/*
: Standard macro files

/usr/lib/font/dev*/*
: Files containing font width tables for troff

**SEE ALSO**

cw(1), eqn(1), mmt(1), nroff(1), pic(1), tbl(1), tc(1), troff(1)

mm(5), ms(5), mv(5) in *A/UX Programmer's Reference*

*A/UX Text Processing Tools*

**NAME**

pack, pcat, unpack — compress and expand files

**SYNOPSIS**

pack [-] [-f] *file...*

pcat *file...*

unpack *file...*

**ARGUMENTS**

- Sets an internal flag that causes the number of times each byte is used, its relative frequency, and the code for the byte to be printed on the standard output. Additional occurrences of - in place of *file* will cause the internal flag to be set and reset.

-f Forces the packing of *file*. This is useful for causing an entire directory to be packed, even if some of the files will not benefit. If pack is successful, *file* will be removed. Packed files can be restored to their original form using unpack or pcat.

*file* Specifies the file to be compressed or expanded.

**DESCRIPTION**

pack attempts to store the specified files in a compressed form. Wherever possible (and useful), each input file is replaced by a packed file *file*.z with the same access modes, access and modified dates, and owner as those of *file*.

The pack program uses Huffman (minimum redundancy) codes on a byte-by-byte basis.

The amount of compression obtained depends on the size of the input file and the character frequency distribution. Because a decoding tree forms the first part of each .z file, it is usually not worthwhile to pack files smaller than three blocks, unless the character frequency distribution is very skewed, which may occur with printer plots or pictures.

Typically, text files are reduced to 60 to 75 percent of their original size. Load modules, which use a larger character set and have a more uniform distribution of characters, show little compression, the packed versions being about 90 percent of the original size.

The pack program returns a value that is the number of files that it failed to compress.

No packing will occur if:

the file appears to be already packed; the filename has more than 12 characters; the file has links; the file is a directory; the file cannot be opened; no disk storage blocks will be saved by packing; the file is of zero length; a file called *file*.z already exists; the .z file cannot be

created; an I/O error occurred during processing.

The last segment of the filename must contain no more than 12 characters to allow space for the appended .z extension. Directories cannot be compressed.

The pcat command does for packed files what cat(1) does for ordinary files, except that pcat cannot be used as a filter. The specified files are unpacked and written to the standard output. Thus to view a packed file named *file*.z use:

```
pcat file.z
```

or just:

```
pcat file
```

To make an unpacked copy, say *nnn*, of a packed file named *file*.z (without destroying *file*.z) use the command:

```
pcat file > nnn
```

The pcat program returns the number of files it was unable to unpack. Failure may occur if:

the filename (exclusive of the .z) has more than 12 characters; the file cannot be opened; the file does not appear to be the output of pack.

The unpack program expands files created by pack. For each file specified in the command, a search is made for a file called *file*.z (or just *file*, if *file* ends in .z). If this file appears to be a packed file, it is replaced by its expanded version. The new file has the .z suffix stripped from its name, and has the same access modes, access and modification dates, and owner as those of the packed file.

The unpack program returns a value that is the number of files it was unable to unpack. Failure may occur for the same reasons that it may in pcat, as well as for the following:

```
a file with the ''unpacked'' name already exists;
```

if the unpacked file cannot be created.

**EXAMPLES**

The command:

```
pack file1
```

will pack the file, file1, into the file, file1.z, then removes file1 if packing is successful.

**FILES**

/usr/bin/pack
Executable file

/usr/bin/pcat
Executable file

/usr/bin/unpack
Executable file

**SEE ALSO**

cat(1), compact(1)

*See* more(1)

**NAME**

pagesize — displays the system page size

**SYNOPSIS**

pagesize

**DESCRIPTION**

pagesize prints the size of a page of memory in bytes. This program is useful in constructing portable shell scripts.

**FILES**

/bin/pagesize
Executable file

**SEE ALSO**

uvar(2) in *A/UX Programmer's Reference*

**NAME**

passwd — changes the login password

**SYNOPSIS**

passwd [*name*]

**ARGUMENTS**

*name*

Specifies the login name of the user.

**DESCRIPTION**

This command changes (or installs) a password associated with the login name.

Ordinary users may change only the password that corresponds to their login *name*.

The passwd program prompts ordinary users for their old password, if any. It then prompts for the new password twice. The first time the new password is entered, passwd checks to see if the old password has aged sufficiently. If aging is insufficient, the new password is rejected and passwd terminates; see passwd(4).

Assuming aging is sufficient, a check is made to ensure that the new password meets construction requirements. When the new password is entered a second time, the two copies of the new password are compared. If the two copies are not identical, the cycle of prompting for the new password is repeated for at most two more times.

Passwords must meet the following requirements:

Each password must have at least six characters. Only the first eight characters are significant.

Each password must contain at least two alphabetic characters (uppercase or lowercase) and at least one numeric or special character.

Each password must differ from the user's login *name* and any reverse or circular shift of that login *name*. For comparison purposes, an uppercase letter and its corresponding lowercase letter are equivalent.

New passwords must differ from the old by at least three characters. For comparison purposes, an uppercase letter and its corresponding lowercase letter are equivalent.

One whose effective user ID is zero is called a superuser; see id(1), and su(1). Superusers may change any password; hence, passwd does not prompt superusers for the old password. Superusers are not forced to comply with password aging and password construction requirements. A superuser can create a null password by entering a carriage return in response to the prompt for a new password.

**EXAMPLES**

Entering:

```
passwd
```

will give the response

```
Changing password for <username>
```

and will then prompt for your present password and for the new password (twice).

**FILES**

`/bin/passwd`
Executable file

`/etc/passwd`
Executable file

**SEE ALSO**

`chsh`(1), `login`(1), `id`(1), `su`(1)

`crypt`(3C), `passwd`(4) in *A/UX Programmer's Reference*

## NAME

paste — merges lines of several files or subsequent lines of one file

## SYNOPSIS

paste *file1 file2 ...*
paste -d*list file1 file2 ...*
paste -s [-d*list*] *file1 file2 ...*

## ARGUMENTS

*file1*
*file2*

Specifies the first (*file1*) and second (file2*)* input files. If - is used in place of any filename, a line is read from the standard input. (There is no prompting.)

-d*list*

Replaces the newline characters of each but the last file (or last line in case of the -s option) with a tab character, without this option. This option allows replacing the tab character by one or more alternate characters (see below). Replace *list* with one or more characters immediately following -d. Replace the default tab as the line concatenation character. The list is used circularly, that is, when exhausted, it is reused. In parallel merging (i.e., no -s option), the lines from the last file are always terminated with a newline character, not from the *list*. The list may contain the special escape sequences: \n (newline) \t (tab), \\ (backslash), and \0 (empty string, not a null character). Quoting may be necessary, if characters have special meaning to the shell (e.g., to get one backslash, use *-d* \\\\"" ).

-s Merges subsequent lines rather than one from each input file. Use tab, for concatenation, unless a *list* is specified with -d option. Regardless of the *list*, the very last character of the file is forced to be a newline.

## DESCRIPTION

In the first two forms, paste concatenates corresponding lines of the given input files *file1*, *file2*, etc. It treats each file as a column or columns of a table and pastes them together horizontally (parallel merging). If you will, it is the counterpart of cat(1), which concatenates vertically, i.e., one file after the other. In the last form above, paste replaces the function of an older command with the same name by combining subsequent lines of the input file (serial merging). In all cases, lines are glued together with the tab character, or with characters from an optionally specified *list*. Output is to the standard output, so it can be used as the start of a pipe, or as a filter, if - is used in place of a filename.

**EXAMPLES**

The command:

```
ls | paste -d"" -
```

lists directories in one column.

The command:

```
ls | paste - - - -
```

lists directories in four columns.

The command:

```
paste -s -d"\ t\ n" file
```

combines pairs of lines into lines.

**STATUS MESSAGES AND VALUES**

`line too long`
: Output lines are restricted to 511 characters.

`too many files`
: Except for `-s` option, no more than 12 input files may be specified.

**FILES**

`/usr/bin/paste`
: Executable file

**SEE ALSO**

cut(1), grep(1), pr(1)

## NAME

pax — copies files to or from an archive in an IEEE format

## SYNOPSIS

pax [-cimopuvy] [-f *archive*] [-s *replstr*] [-t *device*] [*pattern*]...

pax -r [-cimnopuvy] [-f *archive*] [-s *replstr*] [-t *device*] [*pattern*]...

pax -w [-adimuvy] [-b *blocking*] [-f *archive*] [-s *replstr*] [-t *device*] [-x *format*] [*path*]...

pax -rw [-ilmopuvy] [-s *replstr*] [*path*]... *directory*

## ARGUMENTS

-a Appends the files specified by *path* to the specified archive.

-b *blocking*
Blocks the output to the archive file at the number of bytes specified by *blocking*. A k suffix multiplies the value of *blocking* by 1024, a b suffix multiplies *blocking* by 512, and an m suffix multiplies by 1048576 (1 megabyte). For computers with 16-bit integers, the maximum buffer size is one byte less than 32K. If the value of *blocking* is not specified, it is automatically determined on input and is ignored for -rw.

-c Excludes the files and directories that match the *pattern* argument.

-d Instructs pax not to create intermediate directories not explicitly listed in the archive. This option is ignored unless the -r option is specified.

*directory*
Specifies the destination directory for files copied with pax. To invoke this mode, use both the -r and the -w options. The directory must exist and be writable before the copy is made; otherwise, an error results.

-f *archive*
Specifies *archive* as the path of the input or output archive and overrides the default of standard input for -r or standard output for -w.

-i Interactively renames files. The pax command performs substitutions specified by the -s option before requesting the new filename from the user. The pax command skips a file if an empty line is entered and exits with an exit status of 0 if it encounters the end-of-file signal.

-l Links files instead of copying them, when possible.

-m Instructs pax not to retain file modification times.

`-n` Instructs `pax` to treat the *pattern* arguments as ordinary filenames, when `-r` is specified, but `-w` is not. Only the first occurrence of each of these files in the input archive is read. The `pax` utility exits with an exit status of 0 after reading all files in the list. If one or more files in the list are not found, `pax` writes a message to standard error for each of the files and exits with a nonzero exit status. The filenames are compared before any of the `-i`, `-s`, or `-y` options that are present are applied.

`-o` Restores file ownership as specified in the archive. The invoking process must have appropriate privileges restore file ownership.

`-p` Preserves the access time of the input files after they have been copied.

*path*
Specifies a file to be copied into the archive instead of the files named on the standard input. When a directory is specified, `pax` recursively copies all the files and subdirectories of *path* as well.

*pattern*
Specifies, in the standard notation for shell filename generation, particular files to be read from an archive. If you do not specify a value for *pattern*, `pax` selects all files by default.

`-r` Causes `pax` to read an archive file from the standard input. Only files with names that match any of the *pattern* arguments are selected for extraction. The selected files are conditionally created and copied relative to the current directory tree, subject to the other options specified. By default, the owner and group of selected files are those of the invoking process, and the permissions and modification times are the same as those in the archive.

The supported archive formats are automatically detected on input. The default output format is `ustar`, but you can override this default by using the `-x` option.

`-rw`
Causes `pax` to read the files and directories referred to in *path* and copy them to the destination specified by *directory*. The placeholder *path* refers to the files and (recursively) subdirectories of that directory. If the value of *path* is not given, the standard input is read to get a list of paths to copy, one path per line. In this case, only those paths appearing on the standard input are copied. The directory *directory* must exist and have the proper permissions before copying can occur.

`-s` *replstr*
Modifies filenames according to the substitution string, using the

syntax of ed(1), as in this example:

```
pax -rs /old/new/[gp]
```

Any non-null character can be used as a delimiter. In the example shown, / is the delimiter. You can specify multiple -s expressions; the expressions are applied in the order specified, terminating with the first successful substitution. The optional trailing p causes successful mappings to be listed to standard error. The optional trailing g causes the old expression to be replaced each time it occurs in the source string. Files that substitute to an empty string are ignored both on input and output.

-t *device*
Names the input or output archive device by using the *device* argument as an implementation-defined identifier. This option overrides the default of standard input for -r and standard output for -w.

-u Copies each file only if it is newer than a pre-existing file with the same name. This output is used in conjunction with the -a option.

-v Lists filenames as they are encountered. This option produces a verbose list of the table of contents on the standard output when both the -r and -w options are omitted; otherwise, the filenames are printed to standard error as they are encountered in the archive.

-x *format*
Specifies the format of the output archive. The input format, which must be one of those listed here, is automatically determined when the -r option is used. These formats are supported:

cpio
The extended CPIO interchange format specified in ''Extended CPIO Format'' in *IEEE Standard 1003.1-1988.*

ustar
The extended TAR interchange format specified in ''Extended TAR Format'' in *IEEE Standard 1003.1-1988.* This is the default archive format.

-w Writes the files and directories specified by *path* arguments to the standard output, together with the path and status information prescribed by the archive format used. If the argument *path* refers to a directory, pax recursively traverses all the files and subdirectories of *path* as well. If *path* is not given, then the standard input is read to get a list of paths to copy, one path per line. In this case, only those paths appearing on the standard input are copied.

`-y` Interactively prompts for the disposition of each file. Substitutions specified by `-s` options (described earlier in this list) are performed before the user is prompted for disposition. The end-of-file signal or an input line starting with the character `q` after the prompt causes `pax` to exit. Otherwise, an input line starting with anything other than the `-y` option causes the file to be ignored. This option cannot be used in conjunction with the `-i` option.

**DESCRIPTION**

`pax` reads and writes archive files that conform to the description in "Archive/Interchange File Format" specified in *IEEE Standard 1003.1-1988*. The `pax` utility can also read, but not write, a number of other file formats in addition to those specified in "Archive/Interchange File Format." Support for these traditional file formats, such as V7 `tar` and System V `cpio`, is provided for backward compatibility and maximum portability.

The `pax` command also supports traditional `cpio` and `tar` interfaces if invoked with the name `cpio` or `tar`, respectively. See `cpio`(1) or `tar`(1) for more details.

Combinations of the `-r` and `-w` options specify whether `pax` reads, writes, or lists the contents of the specified archive, or moves the specified files to another directory.

If neither the `-r` nor the `-w` option is given, then `pax` lists the contents of the specified archive. In this mode, `pax` lists normal files one per line, lists nonsymbolic link paths as

*path* == *link*

and lists symbolic link paths, if supported by the implementation, as

*path* -> *link*

where *path* is the name of the file being extracted and *link* is the name of a file that appeared earlier in the archive.

If the `-v` option is specified, then `pax` lists normal paths in the same format as that used by the `ls` utility when `ls` is invoked with the `-l` option. Nonsymbolic links are shown as follows:

<*listing*> == *link*

Symbolic links, if supported, are shown as follows:

<*listing*> -> *link*

You can use the `pax` command to read and write archives that span multiple physical volumes. Upon detecting an end-of-medium signal on an archive that is not yet complete, `pax` prompts you for the next volume of the archive and allows you to specify the location of the next volume.

Only the last instance of multiple -f or -t options takes effect.

When pax writes to an archive, the standard input is used as a list of paths if *path* is not specified. The format is one path per line. Otherwise, the standard input is the archive file, which is formatted according to one of the specifications in "Archive/Interchange File Format" in *IEEE Standard 1003.1-1988* or according to some other implementation-defined format.

The user ID and group ID of the process, together with the appropriate privileges, affect the ability of pax to restore ownership and permissions attributes of the archived files. (See "Format-reading Utility" in "Archive/Interchange File Format," in *IEEE Standard 1003.1-1988.*)

The options -a, -c, -d, -i, -l, -p, -t, -u, and -y are provided for functional compatibility with the historical cpio and tar utilities. The option defaults reflect the most common usage of these options; therefore, some of the options have meanings different from those of the historical commands.

**EXAMPLES**

To copy the contents of the current directory to tape drive 0, use the following command:

```
pax -w -f /dev/rmt0
```

To copy the contents of olddir to newdir, use the following command:

```
mkdir newdir
cd olddir
pax -rw . newdir
```

The following command reads the archive pax.out. All archive files below /usr are extracted relative to the current directory.

```
pax -r -s ',/usr/*,,' -f pax.out
```

In this example, the substitution string delimiter is the comma.

**STATUS MESSAGES AND VALUES**

The pax program exits with one of two types of values. If all of the files in the archive were processed successfully, pax exits with a value of 0. If pax terminated because of errors encountered during operation, pax exits with a nonzero value.

**LIMITATIONS**

Special permissions may be required to copy or extract special files.

Device, user ID, and group ID numbers larger than 65535 cause additional header records to be output. Some historical versions of cpio(1) and tar(1) ignore these records.

Certain historical restrictions apply to the archive formats described in ''Archive/Interchange File Format.'' For example, the length of paths stored in the archive is restricted.

In `ls -l` style listings of `tar` format archives, link counts are listed as 0 because the `ustar` archive format does not keep link-count information.

In the event of errors, `pax` terminates immediately, without processing any additional files on the command line or in the archive.

**NOTES**



**FILES**

`/usr/bin/pax`
Executable file

**SEE ALSO**

`cpio`(1), `find`(1), `tar`(1)

`cpio`(4), `tar`(4) in *A/UX Programmer's Reference*

*See* pack(1)

*See* machid(1)

## NAME

pg — shows the contents of a file in display-size chunks

## SYNOPSIS

pg [-*number*] [+*linenumber*] [+/*pattern*] [-c] [-e] [-f] [-n]
[-p *string*] [-s] [*file*]...

## ARGUMENTS

-*number*
: Specifies an integer the size (in lines) of the window that pg is to use instead of the default. On a terminal containing 24 lines, the default window size is 23.

+*linenumber*
: Starts examining the file at *linenumber*.

+/*pattern*
: Starts examining the file at the first line containing the regular expression pattern. The terminal / may be omitted from this command.

-c
: Homes the cursor and clears the screen before displaying each page. This option is ignored if clear_screen is not defined for this terminal type in the terminfo data base.

-e
: Causes pg not to pause at the end of each file.

-f
: Inhibits pg from splitting lines. Normally, pg splits lines longer than the screen width, but some sequences of characters in the text being displayed (e.g., escape sequences for underlining) generate undesirable results.

*file*
: The filename - or null arguments indicate that pg should read from the standard input.

-n
: Causes an automatic end of command as soon as a command letter is entered. Normally, commands must be terminated by a newline.

-p *string*
: Causes pg to use *string* as the prompt. If the prompt string contains a %d, the first occurrence of %d in the prompt will be replaced by the current page number when the prompt is issued. The default prompt string is :.

-s
: Causes pg to print all messages and prompts in standout mode (usually inverse video).

## DESCRIPTION

pg is a filter that examines files, one screenful at a time, on a soft-copy terminal. Each screenful is followed by a prompt. If the user types a carriage return, another page is displayed; other possibilities are

enumerated later in this manual page.

This command is different from previous paginators in that it allows you to back up and review something that has already passed. The method for doing this is explained later in this manual page.

In order to determine terminal attributes, pg scans the terminfo database for the terminal type specified by the environment variable TERM. If TERM is not defined, the terminal type dumb is assumed.

The responses that may be typed when pg pauses can be divided into three categories: those causing further perusal, those that search, and those that modify the perusal environment.

Commands which cause further perusal normally take a preceding *address*, an optionally signed number indicating the point from which further text should be displayed. This *address* is interpreted in either pages or lines depending on the command. A signed *address* specifies a point relative to the current page or line, and an unsigned *address* specifies an address relative to the beginning of the file. Each command has a default address that is used if none is provided.

The perusal commands and their defaults are as follows:

+1 newline
: (or blank) Causes one page to be displayed. The address is specified in pages.

+1 l
: Causes pg to simulate scrolling the screen forward or backward, the number of lines specified, with a relative address. With an absolute address this command prints a screenful beginning at the specified line.

+1 d or CONTROL-d
: Simulates scrolling half a screen forward or backward.

The following perusal commands take no address:

. or CONTROL-l
: Causes the current page of text to be redisplayed by typing a single period or CONTROL-l.

$ Displays the last windowful in the file. Use with caution when the input is a pipe.

The following commands are available for searching for text patterns in the text. The regular expressions described in ed(1) are available. They must always be terminated by a newline, even if the *-n* option is specified:

*i*/*pattern*/
: Searches forward for the *i*th (default *i*=1) occurrence of *pattern*.

Searching begins immediately after the current page and continues to the end of the current file, without wrap-around. The final / may be omitted unless m, b, or t modifiers are appended.

*i*^*pattern*^

*i*?*pattern*?
Searches backward for the *i*th (default *i*=1) occurrence of *pattern*. Begins searching immediately before the current page and continues to the beginning of the current file, without wrap-around. The final ^ and ? may be omitted from these commands unless the m, b, or t modifiers are appended. The ^ notation is useful for Adds 100 terminals that will not properly handle the ?.

After searching, pg will normally display the line found at the top of the screen. This can be modified by appending m or b to the search command to leave the line found in the middle or at the bottom of the window from now on. The suffix t can be used to restore the original situation.

The user of pg can modify the environment of perusal with the following commands:

*i*f Skips *i* screenfuls and prints a screenful of lines *i*n. Begins perusing the *i*th next file in the command line. The *i* is an unsigned number, default value is 1.

*i*p Begins perusing the *i*th previous file in the command line. *i* is an unsigned number, default is 1.

*i*w Displays another window of text. If *i* is present, set the window size to *i*.

*i*z Specifies *i* as the new window size, if present. Otherwise, this option has the same effect as typing a space.

s *filename*
Saves the input in the named file. Only the current file being perused is saved. The white space between the s and *filename* is optional. This command must always be terminated by a newline, even if the *-n* option is specified.

h Helps by displaying an abbreviated summary of available commands.

q
Q Quits the pg program.

!*command*
Passes *command* to the shell, whose name is taken from the SHELL environment variable. If this is not available, the default shell is used. This command must always be terminated by a newline, even if the *-n* option is specified.

At any time when output is being sent to the terminal, the user can press the quit key (normally CONTROL-\) or the interrupt key. This causes pg to stop sending output, and display the prompt. The user may then enter one of the above commands in the normal manner. Unfortunately, some output is lost when this is done, due to the fact that any characters waiting in the terminal's output queue are flushed when the quit signal occurs.

If the standard output is not a terminal, then pg acts just like cat(1), except that a header is printed before each file (if there is more than one).

**EXAMPLES**

A sample usage of pg in reading system news is:

```
news | pg -p "(Page %d):"
```

**LIMITATIONS**

If terminal tabs are not set every eight positions, undesirable results may occur.

When using pg as a filter with another command that changes the terminal I/O options (e.g., crypt(1)), terminal settings may not be restored correctly.

**NOTES**

While waiting for terminal input, pg responds to the interrupt character (CONTROL-C by default) by terminating execution. Between prompts, however, the interrupt signal interrupts pg's current task and place the user in prompt mode. These should be used with caution when input is being read from a pipe, since an interrupt is likely to terminate the other commands in the pipeline.

**FILES**

/usr/bin/pg
: Executable file

/usr/lib/terminfo/*
: Terminal information files

/tmp/pg*
: Temporary file

**SEE ALSO**

crypt(1), ed(1), grep(1), more(1)

terminfo(4) in *A/UX Programmer's Reference*

**NAME**

pic — preprocesses troff files that contain drawings

**SYNOPSIS**

pic [-T*tty-type*] [-] [*file*]...

**ARGUMENTS**

- Indicates that pic should read the file from the standard input.

*file* Specifies the file to be preprocessed.

-T*tty-type*
Specifies the device type, *tty-type*. The currently supported device types are: psc (POSTSCRIPT® device such as the Apple LaserWriter®), iw (the Apple ImageWriter® II printer), and aps (Autologic APS-5). The default is -Tpsc.

**DESCRIPTION**

pic is a troff(1) preprocessor for drawing simple figures on a typesetter. The basic objects are boxes, lines, arrows, circles, ellipses, arcs, and text.

**FILES**

/usr/bin/pic
Executable file

**SEE ALSO**

grap(1), troff(1)

postscript(4) in *A/UX Programmer's Reference*

"pic Reference" in *A/UX Text Processing Tools*

## NAME

pr — formats text for a print device

## SYNOPSIS

pr [+*pageno*] [-*columns*] [-a] [-d] [-e*ck*] [-f] [-h *head*] [-i*ck*] [-l*k*] [-m] [-n*ck*] [-o*k*] [-p] [-r] [-s*c*] [-t] [-w*k*] [*file*]...

## ARGUMENTS

+*pageno*
Specifies the page number (*pageno*) to begin formatting. The default is 1.

-*columns*
Specifies the number of columns (*columns*) to produce in the output. The default is 1. The options -e and -i are assumed for multicolumn output.

-a Prints multicolumn output across the page.

-d Produces double-spaced output.

-e*ck*
Expands *input* tabs to character positions *k*+1, 2* *k*+1, 3* *k*+1, etc. If *k* is 0 or is omitted, default tab settings at every eighth position are assumed. Tab characters in the input are expanded into the appropriate number of spaces. If *c* (any nondigit character) is given, it is treated as the input tab character (default for *c* is the tab character).

-f Uses the form-feed character for new pages (default is to use a sequence of line-feeds). Pause before beginning the first page if the standard output is associated with a terminal.

*file* Specifies the file to be formatted. If *file* is -, or if no files are specified, the standard input is assumed.

-h *head*
Uses the next argument as the header instead of the filename.

-i*ck*
Replaces white space wherever possible by inserting tabs to character positions *k*+1, 2* *k*+1, 3* *k*+1, etc, in the output. If *k* is 0 or is omitted, default tab settings at every eighth position are assumed. If *c* (any nondigit character) is given, it is treated as the output tab character (default for *c* is the tab character).

-l*k* Sets the length of a page to *k* lines (default is 66).

-m Merges and formats all files simultaneously, one per column (overrides the -*k*, and -a options).

-n*ck*
Provides *k*-digit line numbering (default for *k* is 5). The number

occupies the first *k*+1 character positions of each column of normal output or each line of `-m` output. If *c* (any nondigit character) is given, it is appended to the line number to separate it from whatever follows (default for *c* is a tab).

`-o`*k* Offsets each line by *k* character positions (default is 0). The number of character positions per line is the sum of the width and offset.

`-p` Pauses before beginning each page if the output is directed to a terminal (`pr` will ring the bell at the terminal and wait for a carriage return).

`-r` Prints no diagnostic reports on failure to open files.

`-s`*c* Separates columns by the single character *c* instead of by the appropriate number of spaces (default for *c* is a tab).

`-t` Prints neither the five-line identifying header nor the five-line trailer normally supplied for each page. Quit formatting after the last line of each file without spacing to the end of the page.

`-w`*k* Sets the width of a line to *k* character positions instead of the default 72 characters for multicolumn output. This option *must* be used with the `-`*k* (number of columns) option.

## DESCRIPTION

`pr` formats the named files on the standard output. By default, the listing is separated into pages, each headed by the page number, a date and time, and the name of the file.

By default, columns are of equal width, separated by at least one space; lines which do not fit are truncated. If the `-s` option is used, lines are not truncated and columns are separated by the separation character.

If the standard output is associated with a terminal, error messages are withheld until `pr` has completed formatting.

## EXAMPLES

The command:

```
pr -3dh "file list" file1 file2
```

formats `file1` and `file2` as a double-spaced, three-column listing headed by "`file list`".

The command:

```
pr -e9 -t < file1 > file2
```

writes `file1` on `file2`, expanding tabs to columns 10, 19, 28, 37, and so on.

**FILES**

`/bin/pr`
: Executable file

`/dev/tty*`
: Device files

**SEE ALSO**

`cat`(1), `fmt`(1), `lp`(1), `lpr`(1)

**NAME**

printenv — displays the value of variables set in the current environment

**SYNOPSIS**

printenv [*argument*]

**ARGUMENTS**

*argument*

Specifies the environment variable name that will have its values displayed. Replace *argument* with environment variables names such as: HOME, SHELL, PATH, TERM, LOGNAME, TERMCAP, and EXINIT. If no argument is given, it displays the values for the entire environment.

**DESCRIPTION**

printenv takes an environment variable name as an *argument* and displays only the value of that variable.

Some environment variable names and their descriptions are:

HOME
Specifies the pathname of user's home directory.

SHELL
Specifies the shell present at login.

PATH
Searchs the path for binary programs.

TERM
Specifies the type of terminal being used.

LOGNAME
Specifies the login name of the user.

TERMCAP
Specifies the terminal capabilities string.

EXINIT
Specifies a startup list of commands read by ex, edit, and vi.

The man page on the shell you are using (csh (1), ksh (1), or sh (1)) gives a complete list of the environment variables that apply to you.

**EXAMPLES**

The command:

```
printenv HOME
```

displays the pathname of your home directory.

**FILES**

`/bin/printenv`
Executable file

**SEE ALSO**

`csh`(1), `env`(1), `ksh`(1), `sh`(1), `stty`(1), `tset`(1)

`environ`(5) in *A/UX Programmer's Reference*

## NAME

prof — displays profile data

## SYNOPSIS

prof [-a] [-c] [-g] [-h] [-m *mdata*] [-n] [-o] [-s] [-t] [-x] [-z] [*objfile*]

## ARGUMENTS

-a Sorts by increasing symbol address.

-c Sorts by decreasing number of calls.

-g Includes nonglobal symbols (static functions).

-h Suppresses the heading normally printed on the report. (This is useful if the report is to be processed further.)

-m *mdata*
Uses the file *mdata* instead of mon.out for profiling data.

-n Sorts lexically by symbol name.

-o Prints each symbol address (in octal) along with the symbol name.

*objfile*
Specifies the object file from which the symbol table is used.

-s Prints a summary of several of the monitoring parameters and statistics on the standard error output.

-t Sorts by decreasing percentage of total time (default).

-x Prints each symbol address (in hexadecimal) along with the symbol name.

-z Includes all symbols in the profile range (see monitor(3C)), even if associated with zero number of calls and zero time.

## DESCRIPTION

prof interprets the profile file produced by the monitor(3C) function. The symbol table in the object file *objfile* (a.out by default) is read and correlated with the profile file (mon.out by default). For each external text symbol the percentage of time spent executing between the address of that symbol and the address of the next is printed, together with the number of times that function was called and the average number of milliseconds per call.

For the number of calls to a function to be tallied, the -p option of cc(1) must have been given when the file containing the function was compiled. This option to the cc command also arranges for the object file to include a special profiling start-up function that calls monitor(3C) at the beginning and end of execution. It is the call to monitor at the end of execution that causes the mon.out file to be written. Thus, only programs that call

exit(2) or return from *main* cause the mon.out file to be produced.

**LIMITATIONS**

There is a limit of 600 functions that may have call counters established during program execution. If this limit is exceeded, other data is overwritten and the mon.out file is corrupted. The number of call counters used is reported automatically by the prof command whenever the number exceeds 250.

**FILES**

/bin/prof
: Executable file

mon.out
: File used for profile

a.out
: File used for namelist

**SEE ALSO**

cc(1), nm(1)

exit(2), profil(2), monitor(3C) in *A/UX Programmer's Reference*

**NAME**

prs — displays information about an SCCS file

**SYNOPSIS**

prs [-a] [-c[*date-time*]] [-d[*dataspec*]] [-e] [-l] [-r[*SID*]] *file...*

**ARGUMENTS**

-a Requests information for both removed, i.e., delta type = *R*, (see rmdel(1)) and existing, i.e., delta type = *D*, deltas. If this option is not specified, information for existing deltas only is provided.

-c[*date-time*]
Specifies the cutoff date and time. Replace *date-time* with a value in the following format:
*mm/dd/yy* [*hh:mm:ss*]

Units omitted from the the date-time default to their maximum possible values; that is, -c7502 is equivalent to -c750228235959. Any number of nonnumeric characters may separate the various 2-digit pieces of the *cutoff* date in the form:

```
-c77/2/2 9:22:25
```

-d[*dataspec*]
Specifies the output data specification. Replace *dataspec* with a string consisting of SCCS file *data keywords* (see ''Data Keywords'' later in this manual page) interspersed with optional user-supplied text.

-e Requests information for all deltas created earlier than and including the delta designated via the -r option or the date given by the -c option.

*file* Specifies the SCCS file to be affected. If a name of - is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file or directory to be processed; nonSCCS files and unreadable files are silently ignored.

-l Requests information for all deltas created later than and including the delta designated via the -r option or the date given by the -c option.

-r[*SID*]
Specifies the SCCS Identification (SID) string of a delta for which information is desired. If *SID* is not specified, the SCCS Identification string of the most recently created delta is assumed.

**DESCRIPTION**

prs displays, on the standard output, parts or all of an SCCS file (see sccsfile(4)) in a user-supplied format. If a directory is named, prs behaves as though each file in the directory were specified as a named file, except that nonSCCS files (last component of the pathname does not begin

with `s.`) and unreadable files are silently ignored.

Arguments to `prs`, may appear in any order.

**Data keywords**

Data keywords specify which parts of an SCCS file are to be retrieved and output. All parts of an SCCS file (see `sccsfile`(4)) have an associated data keyword. There is no limit on the number of times a data keyword may appear in a *dataspec*.

The information displayed by `prs` consists of: (1) the user-supplied text; and (2) appropriate values (extracted from the SCCS file) substituted for the recognized data keywords in the order of appearance in the *dataspec*. The format of a data keyword value is either *Simple* (S), in which keyword substitution is direct, or *Multi-line* (M), in which keyword substitution is followed by a RETURN.

"User-supplied text" is any text other than recognized data keywords. A tab is specified by `\t` and RETURN/newline is specified by `\n`. The default data keywords are:

```
:Dt:\t:DL:\nMRs:\n:MR:COMMENTS:\n:C:
```

TABLE 1. SCCS Files Data Keywords

| *Keyword* | *Data Item* | *File Section* | *Value* | *Format* |
|---|---|---|---|---|
| :Dt: | Delta information | Delta Table | See below* | S |
| :DL: | Delta line statistics | " | :Li:/:Ld:/:Lu: | S |
| :Li: | Lines inserted by Delta | " | nnnnn | S |
| :Ld: | Lines deleted by Delta | " | nnnnn | S |
| :Lu: | Lines unchanged by Delta | " | nnnnn | S |
| :DT: | Delta type | " | D or R | S |
| :I: | SCCS ID string (SID) | " | :R:.:L:.:B:.:S: | S |
| :R: | Release number | " | nnnn | S |
| :L: | Level number | " | nnnn | S |
| :B: | Branch number | " | nnnn | S |
| :S: | Sequence number | " | nnnn | S |
| :D: | Date Delta created | " | :Dy:/:Dm:/:Dd: | S |
| :Dy: | Year Delta created | " | nn | S |
| :Dm: | Month Delta created | " | nn | S |
| :Dd: | Day Delta created | " | nn | S |
| :T: | Time Delta created | " | :Th:::Tm:::Ts: | S |
| :Th: | Hour Delta created | " | nn | S |
| :Tm: | Minutes Delta created | " | nn | S |
| :Ts: | Seconds Delta created | " | nn | S |
| :P: | Programmer who created Delta | " | logname | S |
| :DS: | Delta sequence number | " | nnnn | S |

| | | | | |
|---|---|---|---|---|
| :DP: | Predecessor Delta seq-no. | " | nnnn | S |
| :DI: | Seq-no. of deltas incl., excl., ignored | " | :Dn:/:Dx:/:Dg: | S |
| :Dn: | Deltas included (seq #) | " | :DS: :DS:... | S |
| :Dx: | Deltas excluded (seq #) | " | :DS: :DS:... | S |
| :Dg: | Deltas ignored (seq #) | " | :DS: :DS:... | S |
| :MR: | MR numbers for delta | " | text | M |
| :C: | Comments for delta | " | text | M |
| :UN: | User names | User Names | text | M |
| :FL: | Flag list | Flags | text | M |
| :Y: | Module type flag | " | text | S |
| :MF: | MR validation flag | " | *yes or no* | *S* |
| :MP: | MR validation pgm name | " | *text* | *S* |
| :KF: | Keyword error/warning flag | " | *yes or no* | *S* |
| :KV: | Keyword validation string | " | *text* | *S* |
| :BF: | *Branch flag* | " | *yes or no* | *S* |
| :J: | *Joint edit flag* | " | *yes or no* | *S* |
| :LK: | *Locked releases* | " | *:R:...* | *S* |
| :Q: | *User defined keyword* | " | *text* | *S* |
| :M: | *Module name* | " | *text* | *S* |
| :FB: | *Floor boundary* | " | *:R:* | *S* |
| :CB: | *Ceiling boundary* | " | *:R:* | *S* |
| :Ds: | *Default SID* | " | *:I:* | *S* |
| :ND: | *Null delta flag* | " | *yes or no* | *S* |
| :FD: | File descriptive text | *Comments* | *text* | *M* |
| :BD: | *Body* | *Body* | *text* | *M* |
| :GB: | *Gotten body* | " | *text* | *M* |
| :W: | A form of what(1) string | *N/A* | *:Z::M:\t:I:* | *S* |
| :A: | A form of what(1) string | *N/A* | *:Z::Y: :M: :I::Z:* | *S* |
| :Z: | what(1) string delimiter | *N/A* | *@(#)* | *S* |
| :F: | *SCCS filename* | *N/A* | *text* | *S* |
| :PN: | *SCCS file pathname* | *N/A* | *text* | *S* |

** :Dt: = :DT: :I: :D: :T: :P: :DS: :DP:*

## EXAMPLES

The command

```
prs -d"User IDs for :F: are:\n:UN:" s.file
```

may produce on the standard output:

```
User IDs for s.file are:
xyz
131
abc
```

The command

```
prs -d"Newest delta for pgm :PM:: \
       :I: Created :D: By :P:" -r s.file
```

may produce on the standard output:

```
Newest delta for pgm main.c: C.7 Created 77/12/1 By cas
```

As a ''special'' case,

```
prs s.file
```

may produce on the standard output:

```
D 1.1 77/12/1 00:00:00 cas 1 000000/00000/00000
MRs:
bl78-12345
bl79-54321
COMMENTS:
this is the comment line for s.file initial delta
```

for each delta table entry of the ''D'' type. The only argument allowed to be used with the ''special'' case is the `-a` option.

**STATUS MESSAGES AND VALUES**

Use `help` for explanations.

**FILES**

`/usr/bin/prs`
: Executable file

`/tmp/pr?????`
: Temporary file

**SEE ALSO**

`admin`(1), `cdc`(1), `comb`(1), `delta`(1), `get`(1), `help`(1), `rmdel`(1), `sact`(1), `sccs`(1), `sccsdiff`(1), `unget`(1), `val`(1), `what`(1)

`sccsfile`(4) in *A/UX Programmer's Reference*

''SCCS Reference'' in *A/UX Programming Languages and Tools, Volume 2*

**NAME**

ps — reports process status

**SYNOPSIS**

ps [-a] [-c*corefile*] [-d] [-e] [-f] [-g*grplist*] [-l] [-n*namelist*] [-p*proclist*] [-s*swapdev*] [-t*termlist*] [-u*uidlist*]

**ARGUMENTS**

-a Prints information about all processes, except process group leaders and processes not associated with a terminal.

-c*corefile*
Use the file *corefile* in place of /dev/kmem.

-d Prints information about all processes, except process group leaders.

-e Prints information about all processes.

-f Generates a *full* listing.

-g*grplist*
Restricts listing to data about processes whose process group leaders are given in *grplist*.

-l Generates a *long* listing.

-n*namelist*
Specifies *namelist* as the alternate file to be used in place of /unix.

-p*proclist*
Restricts listing to data about processes whose process ID numbers are given in *proclist*.

-s*swapdev*
Uses the file *swapdev* in place of /dev/swap. This is useful when examining a *corefile*. A *swapdev* of /dev/null will cause the user block to be zeroed out.

-t*termlist*
Restricts listing to data about the processes associated with the terminals given in *termlist*. The *termlist* may be in one of two forms: a list of terminal identifiers separated from one another by a comma, or a list of terminal identifiers enclosed in double quotes and separated from one another by a command and/or one or more spaces. Terminal identifiers may be specified in one of two forms: the device's filename (e.g., tty04), or, if the device's filename starts with tty, just the digit identifier (e.g., 04).

-u*uidlist*
Restricts listing to data about processes whose user ID numbers or login names are given in *uidlist*. In the listing, the numerical user ID will be printed unless the -f option is used, in which case the login name will be printed.

**DESCRIPTION**

ps prints certain information about active processes. Without options, information is printed about processes associated with the current terminal. The output consists of a short listing containing only the process ID, terminal identifier, cumulative execution time, and the command name. Otherwise, the information that is displayed is controlled by the selection of options.

Options using lists as arguments may have the list specified in one of two forms: a list of identifiers separated from one another by a comma, or a list of identifiers enclosed in double quotes and separated from one another by a comma and/or one or more spaces.

The column headings and the meaning of the columns in a ps listing are given below; the letters f and l indicate the option (*full* or *long*), respectively, that causes the corresponding heading to appear; *all* means that the heading always appears. Note that these two options determine only what information is provided for a process; they do not determine which processes will be listed.

F (l)
Flags (hex and additive) associated with the process:

0 swapped;

1 system process;

2 being traced by another process;

4 another tracing flag;

8 process cannot be woken by a signal;

10 in core;

20 locked in memory;

100 process group leader;

200 faulting in page

400 COFF binary

1000
process is using select system call

2000
timing out during sleep

4000
4.2-style job control

8000
restore old mask after signal

`S (l)`
The state of the process:

- `-` nonexistent;
- `S` sleeping;
- `R` running;
- `I` intermediate (between states);
- `Z` terminated;
- `T` stopped.
- `O` as running on CPU
- `X` waiting for virtual memory

`UID (f,l)`
The user ID number of the process owner; the login name is printed under the `-f` option.

`PID (all)`
The process ID of the process; it is possible to kill a process if you know this datum.

`PPID (f,l)`
The process ID of the parent process.

`C (f,l)`
Processor utilization for scheduling.

`PRI (l)`
The priority of the process; higher numbers mean lower priority.

`NI (l)`
Nice value; used in priority computation.

`ADDR (l)`
The memory address of the `u-area` (a pointer to the page tables) of the process, if resident; otherwise, the disk address.

`SZ (l)`
The size in logical pages of the core image of the process.

`WCHAN (l)`
The event for which the process is waiting or sleeping; if blank, the process is running.

STIME (f)
Starting time of the process.

TTY (all)
The controlling terminal for the process.

TIME (all)
The cumulative execution time for the process.

COMMAND (all)
The command name; the full command name and its arguments are printed under the -f option.

A process that has exited and has a parent, but has not yet been waited for by the parent, is marked defunct.

Under the -f option, ps tries to determine the command name and arguments given when the process was created by examining memory or the swap area. Failing this, the command name, as it would appear without the -f option, is printed in square brackets.

**EXAMPLES**

The command:

```
ps -ef
```

displays information about all processes, with or without terminals.

**LIMITATIONS**

Things can change while ps is running; the picture it gives is only a close approximation to reality. Some data printed for defunct processes are irrelevant.

Processes which are swapped onto other than the default swap device (see swap(1M)) will have some invalid information printed out.

**FILES**

/bin/ps
Executable file

/unix
A/UX kernel file

/dev/kmem
Memory file

/dev/swap
Default swap device file

/etc/passwd
File which supplies UID information

/etc/ps_data
Internal data structure file

`/dev`
File containing terminal (`tty`) names

**SEE ALSO**

`kill`(1), `nice`(1), `w`(1)

`acctcom`(1M), `pstat`(1M) in *A/UX System Administrator's Reference*

## NAME

psdit — converts troff intermediate format to POSTSCRIPT format

## SYNOPSIS

psdit [-F *fontdir*] [-o *list*] [-p *prologue*] [*file*]

## ARGUMENTS

-F *fontdir*
: Takes font information from *fontdir* instead of the default.

*file*
: Specifies the file to be translated. If this argument is not specified, the standard input is used.

-o *list*
: Prints pages whose numbers are given in the comma-separated *list*. The list contains single numbers *n* and ranges *n1-n2*. A missing *n1* means the lowest-numbered page; a missing *n2* means the highest.

-p *prologue*
: Uses the contents of *prologue* instead of the default POSTSCRIPT prologue.

## DESCRIPTION

psdit translates a *file* created by device-independent troff(1) to POSTSCRIPT format for printing on a POSTSCRIPT printer. The POSTSCRIPT file is sent to the standard output.

> *Note:* The input for psdit should be prepared with the corresponding -Tpsc option of troff, pic, grap, and so forth. The eqn program should be run with the flags -r576 and -m2 to produce suitable output. The pic program should be run with the -D option and the -T576 option to set the correct resolution.

psdit enables troff to include arbitrary POSTSCRIPT code in the generated POSTSCRIPT file. psdit recognizes the heretofore undefined % command in the troff intermediate file format to signal the start of raw POSTSCRIPT to be placed ''as is'' in the output file. Everthing between (but not including) the percent sign and a line containing a single period (.) will be placed in the generated POSTSCRIPT output. This POSTSCRIPT is not insulated from the troff coordinate system or from the state of the generated POSTSCRIPT. However, two functions are defined in the prologue so that users may insulate themselves, if they so desire. The PB function (for ''picture begin'') will perform a POSTSCRIPT save operation, translate the POSTSCRIPT coordinate system to troff's idea of the current position on the page, and change the scale and orientation of the coordinate system axes to the standard POSTSCRIPT 72 units per inch. The PE macro (for ''picture end'') will end this protected environment.

Several methods may be employed to incorporate included POSTSCRIPT into the troff intermediate file. The .cf .sy and \! troff commands may be useful. For example, the following sequence may appear anywhere in troff input

```
\!%PB
.cf mypic.ps
\!PE
\!.
```

to include mypic.ps as an illustration. This facility is both powerful and useful, but indiscriminate inclusion of poorly behaved POSTSCRIPT code may be dangerous to your document's health.

**Environment variables**

PSLIBDIR
: Pathname of a directory to use instead of /usr/lib/ps for psdit prologue.

**EXAMPLES**

The following command line will format the file ch.1 using the troff text formatting program, translate troff's output into POSTSCRIPT, and then send the POSTSCRIPT output to the appropriate printer.

```
troff -Tpsc -mm ch.1 | psdit | lp -dPigs
```

**LIMITATIONS**

The B-splines generated by troff are drawn with an approximation. The functions D~ and D~~ in the prologue need a little work.

**FILES**

/usr/bin/psdit
: Executable file

/usr/lib/font/devpsc/*
: troff default description files for POSTSCRIPT virtual device

/usr/lib/ps/psdit.pro
: default POSTSCRIPT prologue file

**SEE ALSO**

lp(1), lpr(1), psroff(1), troff(1)

## NAME

psroff — formats a file through troff so it can be printed on a POSTSCRIPT printer

## SYNOPSIS

psroff [-t] [[-a] [-i] [-m*name*] [-n*N*] [-o*list*] [-q] [-r*aN*] [-s*N*] [-T*dest*]] [[-d*dest*] [-C *class*] [-J *name*] [-h] [-n*x*] [-P *printer*] [-r] [-s] [-m] [-w]] [*file*]...

## ARGUMENTS

-a Sends a printable ASCII approximation of the results to the standard output. This is a troff option.

-C *class*
Sets the job classification for use on the burst page.

-d*dest*
Causes the output to be sent to the named destination.

*file* Specifies the file to be processed through troff. If this argument is not specified, the standard input is used.

-h Suppresses the printing of the job burst page.

-i Reads standard input after the input files are exhausted. This is a troff option.

-J *name*
Sets the job name for use on the burst page.

-m Sends mail after files have been printed.

-m*name*
Inserts the /usr/lib/tmac/tmac.*name* macro file at the beginning of the input *files*. This is a troff option.

-n*N*
Numbers the first generated page *N*. This is a troff option.

-n*x* Causes *x* copies of the output to be produced. The default is one.

-o*list*
Prints only pages whose page numbers appear in the comma-separated *list* of numbers and ranges. A range *N*-*M* means pages *N* through *M*; an initial -*N* means from the beginning to page *N*; and a final *N*- means from *N* to the end. (See LIMITATIONS, later in this section.) This is a troff option.

-P*printer*
Sends the output to the named printer.

-q Invokes the simultaneous input-output mode of the .rd request. This is a troff option.

`-r` Does not page-reverse the output when used with the `lp` spooler. Removes the file upon completion of spooling or upon completion of printing, when used with the *lpr* spooler.

`-r`*aN*
Sets register *a* (one character name) to *N*. This is a `troff` option.

`-s` Suppresses messages from `lp`, when used with the *lp* spooler. Uses symbolic links instead of copying files to the spool directory, when used with the `lpr` spooler.

`-s`*N*
Generates output to encourage typesetter to stop every *N* pages, produce a trailer to allow changing cassettes, and resume when the typesetter's start button is pressed. This is a `troff` option.

`-t` Sends the POSTSCRIPT output to the standard output rather than spooling it to a printer. Note that this overrides the meaning of the `troff -t` option; if that option is needed, then run `troff` directly.

`-T`*dest*
Prepares output for device *dest*, which may be a laser printer or a typesetter. For POSTSCRIPT output destined for an Apple LaserWriter, use `-Tpsc`, and pipe the output to the POSTSCRIPT filter `psdit`.

The supported typesetter is the Autologic APS-5 (`-Taps`). For output destined for an Apple ImageWriter II printer, use the `-Tiw` option and pipe the output to `daiw`(1). Other output devices may be available. This is a `troff` option.

`-w` Writes to the user's terminal after files have been printed.

**DESCRIPTION**

`psroff` is a shell script that runs `troff`(1) in an environment to produce output on a POSTSCRIPT printer. It uses `psdit` to convert `troff` intermediate output to POSTSCRIPT format and spools this output for printing.

By default, the print spooler used by `psroff` is the Berkeley spooler, `lpr`(1), which uses these options: `-C` *class*, `-J` *name*, `-P` *printer*, `-r`, and `-s`.

The environment variable `SPOOLER` may be set to specify the System V spooler, `lp`(1), which uses these options: `-d`*dest*, `-n`*x*, `-h`, `-r`, `-s`, `-m`, and `-w`.

All other options are passed to `troff`.

Using `psroff` is equivalent to using the pipeline of commands:

```
troff -Tpsc options | psdit | $SPOOLER options
```

Using psroff instead of this pipeline involves less typing, but the entire sequence may take slightly more time since a shell script will be executed.

**Environment variables**

The following environmental variables may be used in conjunction with psroff:

SPOOLER
: The name of the print spooler, lpr or lp, for psroff to use. If SPOOLER is not set, psroff spools to lpr.

PRINTER
: The name of a printer (as in the -P option) for lpr to use. If no -P option is specified, lpr uses this printer. If neither -P nor PRINTER is set, psroff spools to a printer named PostScript. This environment variable has no effect on the spooler lp.

LPDEST
: The name of a printer (as in the -d option) for lp to use. If no -d option is specified, lp uses this printer. If neither -d nor LPDEST is set, psroff spools to a printer class named PostScript. This environment variable has no effect on the spooler lpr.

**LIMITATIONS**

The eqn supplied with troff is different from the original. Use the options -r576 -m2 for best results. Other programs (for example, pic) distributed with troff have the device names compiled in (so much for device independence!). Use -T576 with the pic distributed with the Documenter's Workbench®. If your output is destined for an Apple® ImageWriter® II printer, use the -Tiw option for both pic and eqn.

**FILES**

/usr/bin/psroff
: Executable file

/usr/lib/tmac/tmac.*
: Standard macro files

/usr/lib/font/devpsc/*
: troff description files for POSTSCRIPT virtual device

**SEE ALSO**

daiw(1), eqn(1), lpr(1), lp(1), pic(1), psdit(1), refer(1), tbl(1), troff(1).

"nroff/troff Reference" in *A/UX Text Processing Tools*

## NAME

ptx — generates a permuted index

## SYNOPSIS

ptx [-b *break*] [-f] [-g *gap*] [-i *ignore*] [-r] [-t] [-w *n*] [*input* [*output*]]

ptx [-b *break*] [-f] [-g *gap*] [-o *only*] [-r] [-t] [-w *n*] [*input* [*output*]]

## ARGUMENTS

-b *break*
: Uses the characters in the file *break* to separate words. Tab, newline, and space characters are always used as break characters.

-f
: Folds uppercase and lowercase letters for sorting.

-g *gap*
: Uses the next argument *n* as the number of characters that ptx reserves in its calculations for each gap among the four parts of the line as finally printed. The default gap is 3.

-i *ignore*
: Does not use as keywords any words given in the file *ignore*. If the -i and -o options are missing, use /usr/lib/eign as the file *ignore*. This option cannot be used with the -o option.

*input*
: Specifies the file to be processed with a text formatter.

-o *only*
: Uses as keywords any words given in the file *only*. This option cannot be used with the -i option.

*output*
: Specifies the file from which a permuted index is generated.

-r
: Takes any leading nonblank characters of each input line to be a reference identifier (as to a page or chapter), separate from the text of the line, then attaches that identifier as a fifth field on each output line.

-t
: Prepares the output for the phototypesetter.

-w *n*
: Uses the next argument *n* as the length of the output line. The default line length is 72 characters for nroff and 100 for troff.

## DESCRIPTION

ptx generates the file *output* that can be processed with a text formatter (nroff or troff) to produce a permuted index of file *input*. Standard input (-) and standard output are the default. The ptx program has three phases: first, the permutation is done, generating one line for each keyword

in an input line. The keyword is rotated to the front. Second, the permuted file is then sorted. Finally, the sorted lines are rotated so the keyword comes at the middle of each line. ptx output is in the following form:

where .xx is assumed to be an nroff(1) or troff(1) macro provided by the user or provided by ptx(1). The mptx(5) macro package provides the .xx macro definition. The *before keyword* and *keyword and after* fields incorporate as much of the line as fits around the keyword when it is printed. The *tail* and *head* fields, at least one of which is always the empty string, are wrapped-around pieces small enough to fit in the unused space at the opposite end of the line.

**LIMITATIONS**

Line-length counts do not account for overstriking or proportional spacing. Lines that contain tildes (~) are botched because ptx uses that character internally. The ptx program does not discard nonalphanumeric characters.

**FILES**

/usr/bin/ptx
: Executable file

/bin/sort
: Executable file

/usr/lib/eign
: Executable file

/usr/lib/tmac/tmac.ptx
: Executable macro file

**SEE ALSO**

troff(1)

mm(5), mptx(5) in *A/UX Programmer's Reference*

"Other Text Processing Tools," in *A/UX Text Processing Tools*

**NAME**

pwd — prints the name of the working directory

**SYNOPSIS**

pwd

**DESCRIPTION**

pwd prints the pathname of the working (current) directory.

**EXAMPLES**

The command:

```
pwd
```

produces a pathname, such as /usr/games, indicating the directory you are currently in.

**STATUS MESSAGES AND VALUES**

The messages: "Cannot open .." and "Read error in ..." indicate possible file system trouble and should be referred to a system administrator.

**FILES**

/bin/pwd
: Executable file

**SEE ALSO**

csh(1), ksh(1), sh(1)

*A/UX Command Reference* was written, edited, and composed on a desktop publishing system using Apple Macintosh computers, and `troff` running on A/UX. Page proofs were created on Apple LaserWriter printers. Final pages were output directly to 70 millimeter film on an Electrocomp 2000 Electron Beam Recorder. PostScript, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type and display type are Times, Garamond, and Helvetica. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier, a fixed-width font.

Writers: Erik Akin, Mike Elola, Kristi Fredrickson, Michael Hinkson, Linda Kinnier, Paul Pannish, Cheryl Salgado, Kathy Wallace, and Laura Wirth

Writing Group Lead: Mike Elola

Developmental Editor: Silvio Orsino

Art Director: Tamara Whiteside

Production Editor: Jeannette Allen

Production Supervisor: Robin Kerns

Special thanks to Anne Szabla and Chris Wozniak