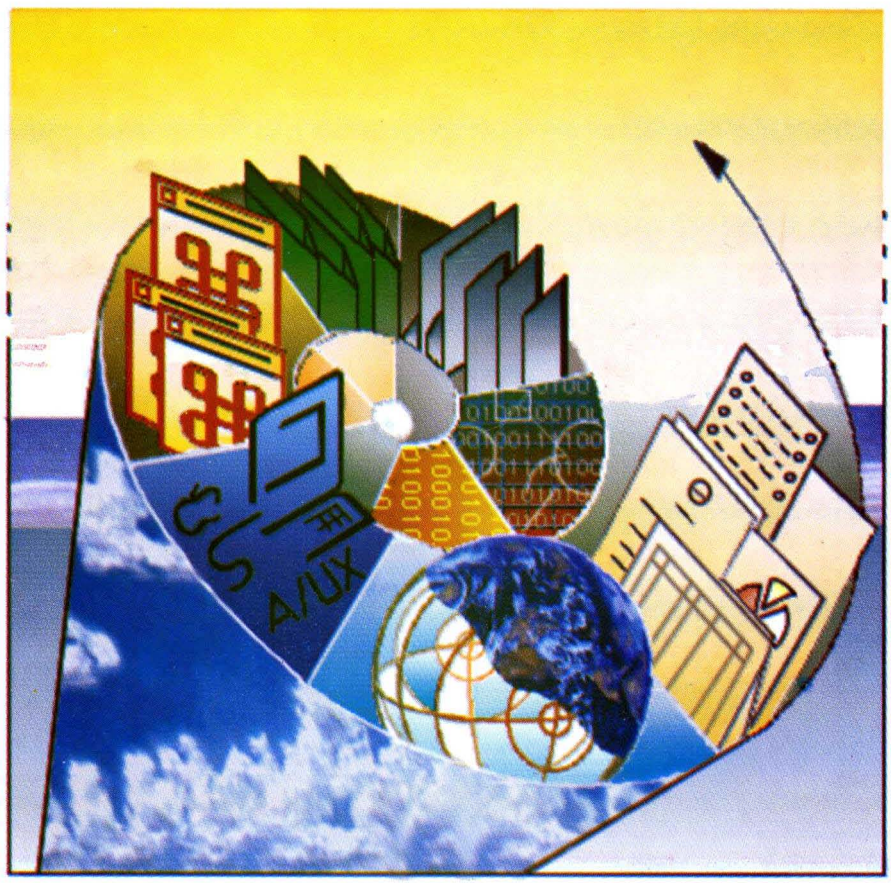
A/UX® Command Reference
Section 1(A-L)



A/UX® Command Reference

Section 1(A-L)

030-0781

APPLE COMPUTER, INC.

Apple Computer, Inc.
20525 Mariani Ave.
Cupertino, California 95014
(408) 996-1010

Simultaneously published in the United States and Canada.

030-0781

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

If you discover physical defects in the manual or in the media on which a software product is distributed, Apple will replace the media or manual 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 THIS MANUAL, 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 reviewed this manual, **APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

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.

A/UX Command Reference

Contents

Preface

Conventions Used in This Manual

A/UX® manuals follow certain conventions regarding presentation of information. Words or terms that require special emphasis appear in specific fonts within the text of the manual. The following sections explain the conventions used in this manual.

Significant fonts

Words that you see on the screen or that you must type exactly as shown appear in `Courier` font. For example, when you begin an A/UX work session, you see the following on the screen:

```
login:
```

The text shows `login:` in `Courier` typeface to indicate that it appears on the screen. If the next step in the manual is

Enter `start`

`start` appears in `Courier` to indicate that you must type in the word. Words that you must replace with a value appropriate to a particular set of circumstances appear in *italics*. Using the example just described, if the next step in the manual is

`login:` *username*

you type in your name—`Laura`, for example— so the screen shows:

```
login: Laura
```

Key presses

Certain keys are identified with names on the keyboard. These modifier and character keys perform functions, often in combination with other keys. In the manuals, the names of these keys appear in the format of an Initial Capital letter followed by SMALL CAPITAL letters.

The list that follows provides the most common keynames.

RETURN	DELETE	SHIFT	ESCAPE
OPTION	CAPS LOCK	CONTROL	

For example, if you enter

```
Applee
```

instead of

```
Apple
```

you would position the cursor to the right of the word and press the DELETE key once to erase the additional *e*.

For cases in which you use two or more keys together to perform a specific function, the keynames are shown connected with hyphens. For example, if you see

Press CONTROL-C

you must press CONTROL and C simultaneously (CONTROL-C normally cancels the execution of the current command).

Terminology

In A/UX manuals, a certain term can represent a specific set of actions. For example, the word *Enter* indicates that you type in an entry and press the RETURN key. If you were to see

Enter the following command: `whoami`

you would type `whoami` and press the RETURN key. The system would then respond by identifying your login name.

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

Term	Action
Enter	Type in the entry and press the RETURN key
Press	Press a *single* letter or key *without* pressing the RETURN key
Type	Type in the letter or letters *without* pressing the RETURN key
Click	Press and then immediately release the mouse button

Term	Action
Select	Position the pointer on an item and click the mouse button
Drag	Position the pointer on an icon, press and hold down the mouse button while moving the mouse. Release the mouse button when you reach the desired position.
Choose	Activate a command title in the menu bar. While holding down the mouse button, drag the pointer to a command name in the menu and then release the mouse button. An example is to drag the File menu down until the command name Open appears highlighted and then release the mouse button.

Syntax notation

A/UX commands follow a specific order of entry. A typical A/UX command has this form:

`command` [*flag-option*] [*argument*] . . .

The elements of a command have the following meanings.

Element	Description
`command`	Is the command name.
flag-option	Is one or more optional arguments that modify the command. Most flag-options have the form [-opt...] where opt is a letter representing an option. Commands can take one or more options.
argument	Is a modification or specification of the command; usually a filename or symbols representing one or more filenames.

Element	Description
brackets ([ ])	Surround an optional item—that is, an item that you do not need to include for the command to execute.
ellipses (...)	Follow an argument that may be repeated any number of times.

For example, the command to list the contents of a directory (`ls`) is followed below by its possible flag options and the optional argument *names*.

```
ls [-R] [-a] [-d] [-C] [-x] [-m] [-l] [-L]
[-n] [-o] [-g] [-r] [-t] [-u] [-c] [-p] [-F]
[-b] [-q] [-i] [-s] [names]
```

You can enter

```
ls -a /users
```

to list all entries of the directory `/users`, where

`ls`	Represents the command name
`-a`	Indicates that *all* entries of the directory be listed
`/users`	Names which directory is to be listed

Command reference notation

Reference material is organized by section numbers. The standard A/UX cross-reference notation is

cmd(sect)

where *cmd* is the name of the command, file, or other facility; *sect* is the section number where the entry resides.

- ☐ Commands followed by section numbers (1M), (7), or (8) are listed in *A/UX System Adminstrator's Reference.*
- ☐ Commands followed by section numbers (1), (1C), (1G), (1N), and (6) are listed in *A/UX Command Reference.*
- ☐ Commands followed by section numbers (2), (3), (4), and (5) are listed in *A/UX Programmer's Reference.*

For example,

```
cat(1)
```

refers to the command `cat`, which is described in Section 1 of *A/UX Command Reference*. References can also be called up on the screen. The `man` command or the `apropos` command displays pages from the reference manuals directly on the screen. For example, enter the command

```
man cat
```

In this example, the manual page for the `cat` command including its description, syntax, options, and other pertinent information appears on the screen. To exit, continue pressing the space bar until you see a command prompt, or press Q at any time to return immediately to your command prompt. The manuals often refer to information discussed in another guide in the suite. The format for this type of cross reference is "Chapter Title," *Name of Guide*. For a complete description of A/UX guides, see *Road Map to A/UX Documentation*. This guide contains descriptions of each A/UX guide, the part numbers, and the ordering information for all the guides in the A/UX documentation suite.

Introduction

to the A/UX Reference Manuals

1. How to use the reference manuals

A/UX Command Reference, *A/UX Programmer's Reference*, and *A/UX System Administrator's Reference* are reference manuals for all the programs, utilities, and standard file formats included with your A/UX® system.

The reference manuals constitute a compact encyclopedia of A/UX information. They are not intended to be tutorials or learning guides. If you are new to A/UX or are unfamiliar with a specific functional area (such as the shells or the text formatting programs), you should first read *A/UX Essentials* and the other A/UX user guides. After you have worked with A/UX, the reference manuals help you understand new features or refresh your memory about command features you already know.

2. Information contained in the reference manuals

A/UX reference manuals are divided into three volumes:

- The two-part *A/UX Command Reference* contains information for the general user. It describes commands you type at the A/UX prompt that list your files, compile programs, format text, change your shell, and so on. It also includes programs used in scripts and command language procedures. The commands in this manual generally reside in the directories `/bin`, `/usr/bin` and `/usr/ucb`.
- The two-part *A/UX Programmer's Reference* contains information for the programmer. It describes utilities for programming, such as system calls, file formats of subroutines, and miscellaneous programming facilities.
- *A/UX System Administrator's Reference* contains information for the system administrator. It describes commands you type at the A/UX prompt to control your machine, such as accounting

commands, backing up your system, and charting your system's activity. These commands generally reside in the directories `/etc`, `/usr/etc`, and `/usr/lib`.

These areas can overlap. For example, if you are the only person using your machine, then you are both the general user and the system administrator.

To help direct you to the correct manual, you may refer to *A/UX Reference Summary and Index*, which is a separate volume. This manual summarizes information contained in the other A/UX reference manuals. The three parts of this manual are a classification of commands by function, a listing of command synopses, and an index.

3. How the reference manuals are organized

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

Manual pages are collated alphabetically by the primary name associated with each. For the individual sections, a table of contents is provided to show the sequence of manual pages. A notable exception to the alphabetical sequence of manual pages is the first entry at the start of each section. As a representative example, `intro.1` appears at the start of Section 1. These `intro.`*section-number* manual pages are brought to the front of each section because they introduce the

other man pages in the same section, rather than describe a command or similar provision of A/UX.

Each of the reference manuals includes at least one complete section of man pages. For example, the *A/UX Command Reference* contains sections 1 and 6. However, since Section 1 (User Commands) is so large, this manual is divided into two volumes, the first containing Section 1 commands that begin with letters A through L, and the second containing Section 6 commands and Section 1 commands that begin with letters M through Z. The sections included in each volume are as follows.

A/UX Command Reference contains sections 1 and 6. Note that both of these sections describe commands and programs available to the general user.

- Section 1—User Commands
  The commands in Section 1 may also belong to a special category. Where applicable, these categories are indicated by the letter designation that follows the section number. For example, the N in `ypcat`(1N) indicates networking as described following.

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

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

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

- Section 6—User Commands
  This section contains all the games, such as `cribbage` and `worms`.

A/UX Programmer's Reference contains sections 2 through 5.

- 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 networking as described following.

  2N Networking system calls

  2P POSIX system calls

- Section 3—Subroutines
  This section describes the available subroutines. The binary versions are in the system libraries in the `/lib` and `/usr/lib` directories. The section includes six 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 networking as described following.

  3C C and assembler 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. Where applicable, these categories are indicated by the letter designation that follows the section number. For example, the N in

`protocols`(4N) indicates networking as described following.

4N Networking formats

- Section 5—Miscellaneous facilities
  This section contains various character sets, macro packages, and other miscellaneous formats. 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 as described following. by the letter designation in parenthesis at the top of the page:

  5F Protocol families

  5P Protocol descriptions

A/UX System Administrator's Reference contains sections 1M, 7 and 8.

- Section 1M—System Maintenance Commands
  This section contains system maintenance programs such as `fsck` and `mkfs`.
- Section 7—Drivers and Interfaces for Devices
  This section discusses the drivers and interfaces through which devices are normally accessed. While access to one or more disk devices is fairly transparent when you are working with files, the provision of *device files* permits you more explicit modes with which to access particular disks or disk partitions, as well as other types of devices such as tape drives and modems. For example, a tape device may be accessed in automatic-rewind mode through one or more of the device file names in the `/dev/rmt` directory (see `tc`(7)). The **FILES** sections of these manual pages identify all the device files supplied with the system as well as those that are automatically generated by certain A/UX configuration utilties. The names of the man pages generally refer to device names or device driver names, rather than the names of the device files themselves.
- Section 8—A/UX Startup Shell Commands
  This section describes the commands that are available from within the A/UX Startup Shell, including detailed descriptions of

those that contribute to the boot process and those that help with the maintenance of file systems.

4. How a manual entry is organized

The name for a manual page entry normally appears twice, once in each upper corner of a page. Like dictionary guide words, these names appear at the top of every physical page. After each name is the section number and, if applicable, a category letter enclosed in parenthesis, such as (1) or (2N).

Some entries describe several routines or commands. For example, `chown` and `chgrp` share a page with the 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 only included in *A/UX Command Reference* and *A/UX System Administrator's Reference*.)

All of the entries have a common format, and may include any of the following parts:

NAME
is the name or names and a brief description.

SYNOPSIS
describes the syntax for using the command or routine.

DESCRIPTION
discusses what the program does.

FLAG OPTIONS
discusses the flag options.

EXAMPLES
gives an example or examples of usage.

RETURN VALUE
describes the value returned by a function.

ERRORS
describes the possible error conditions.

FILES
lists the filenames that are used by the program.

SEE ALSO
provides pointers to related information.

DIAGNOSTICS
discusses the diagnostic messages that may be produced. Self-explanatory messages are not listed.

WARNINGS
points out potential pitfalls.

BUGS
gives known bugs and sometimes deficiencies. Occasionally, it describes the suggested fix.

5. Locating information in the reference manuals

The directory for the reference manuals, *A/UX Reference Summary and Index*, can help you locate information through its index and summaries. The tables of contents within each of the reference manuals can be used also.

5.1 Table of contents

Each reference manual contains an overall table of contents and individual section contents. The general table of contents lists the overall contents of each volume. The more detailed section contents lists the manual pages contained in each section and a brief description of their function. For the most part, entries appear in alphabetic order within each section.

5.2 Commands by function

This summary classifies the A/UX user and administration commands by the general, or most important function they perform. The complete descriptions of these commands are found in *A/UX Command Reference* and *A/UX System Administrator's Reference*. Each is mentioned just once in this listing.

The summary gives you a broader view of the commands that are available and the context in which they are most often used.

5.3 Command synopses

This section is a compact collection of syntax descriptions for all the commands in *A/UX Command Reference* and *A/UX System Administrator's Reference*. It may help you find the syntax of commands more quickly when the syntax is all you need.

5.4 Index

The index lists key terms associated with A/UX subroutines and commands. These key terms allow you to locate an entry when you don't know the command or subroutine name.

The key terms were constructed by examining the meaning and usage of the A/UX manual pages. It is designed to be more discriminating and easier to use than the traditional permuted index, which lists nearly all words found in the manual page NAME sections.

Most manual pages are indexed under more than one entry; for example, `lorder`(1) is included under "archive files," "sorting," and "cross-references." This way you are more likely to find the reference you are looking for on the first try.

5.5 Online documentation

Besides the paper documentation in the reference manuals, A/UX provides several ways to search and read the contents of each reference from your A/UX system.

To see a manual page displayed on your screen, enter the `man`(1) command followed by the name of the entry you want to see. For example,

```
man passwd
```

To see the description phrase from the NAME section of any manual page, enter the `whatis` command followed by the name of the entry you want to see. For example,

```
whatis apropos
```

To see a list of all manual pages whose descriptions contain a given keyword or string, enter the `apropos` command followed by the word or string. For example,

```
apropos remove
```

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

Table of Contents

Section 1: User Commands (A-L)

NAME

intro — introduction to commands and applications programs

DESCRIPTION

This section describes, in alphabetical order, generally available commands. Certain distinctions of purpose are made in the headings:

(1C) Commands for communication with other systems.
(1G) Commands used primarily for graphics and computer-aided design.
(1N) Network commands.

DIAGNOSTICS

Upon termination, each command returns two bytes of status, one supplied by the system and giving the cause for termination, and (in the case of "normal" termination) one supplied by the program (see wait(2) and exit(2)). The former byte is 0 for normal termination; the latter is customarily 0 for successful execution and nonzero to indicate troubles such as erroneous parameters, bad or inaccessible data, or other inability to cope with the task at hand. It is called variously "exit code," "exit status," or "return code," and is described only where special conventions are involved.

WARNINGS

Some commands produce unexpected results when processing files containing null characters. These commands often treat text input lines as strings and therefore become confused upon encountering a null character (the string terminator) within a line.

NAME

300, 300s — filter text containing printer control sequences for a DASI terminal

SYNOPSIS

300 [+12] [-*n*] [-d*t*, *l*, *c*]

300s [+12] [-*n*] [-d*t*, *l*, *c*]

DESCRIPTION

300 supports special functions and optimizes the use of the DASI 300 (GSI 300 or DTC 300) terminal; 300s performs the same functions for the DASI 300s (GSI 300s or DTC 300s) terminal. It converts half-line forward, half-line reverse, and full-line reverse motions to the correct vertical motions. It also attempts to draw Greek letters and other special symbols. It permits convenient use of 12-pitch text. It also reduces printing time 5 to 70%. 300 can be used to print equations neatly, in the sequence:

neqn *file*... | nroff | 300

WARNING: if your terminal has a PLOT switch, make sure it is turned ON before 300 is used.

The behavior of 300 can be modified by the optional flag arguments to handle 12-pitch text, fractional line spacings, messages, and delays.

+12 permits use of 12-pitch, 6 lines/inch text. DASI 300 terminals normally allow only two combinations: 10-pitch, 6 lines/inch, or 12-pitch, 8 lines/inch. To obtain the 12-pitch, 6 lines per inch combination, the user should turn the PITCH switch to 12, and use the +12 option.

-*n* controls the size of half-line spacing. A half-line is, by default, equal to 4 vertical plot increments. Because each increment equals 1/48 of an inch, a 10-pitch line-feed requires 8 increments, while a 12-pitch line-feed needs only 6. The first digit of *n* overrides the default value, thus allowing for individual taste in the appearance of subscripts and superscripts. For example, nroff half-lines could be made to act as quarter-lines by using -2. The user could also obtain appropriate half-lines for 12-pitch, 8 lines/inch mode by using the option -3 alone, having set the PITCH switch to 12-pitch.

`-d`*t*`,`*l*`,`*c* controls delay factors. The default setting is `-d3,90,30`. DASI 300 terminals sometimes produce peculiar output when faced with very long lines, too many tab characters, or long strings of blankless, nonidentical characters. One null (delay) character is inserted in a line for every set of *t* tabs and for every contiguous string of *c* nonblank, nontab characters. If a line is longer than *l* bytes, 1+(total length)/20 nulls are inserted at the end of that line. Items can be omitted from the end of the list, implying use of the default values. Also, a value of zero for *t* (*c*) results in two null bytes per tab (character). The former may be needed for C programs, the latter for files like `/etc/passwd`. Because terminal behavior varies according to the specific characters printed and the load on a system, the user may have to experiment with these values to get correct output. The `-d` option exists only as a last resort for those few cases that do not otherwise print properly. For example, the file `/etc/passwd` may be printed using `-d3,30,5`. The value `-d0,1` is a good one to use for C programs that have many levels of indentation.

Note that the delay control interacts heavily with the prevailing carriage return and line-feed delays. The `stty`(1) modes `nl0 cr2` or `nl0 cr3` are recommended for most uses.

`300` can be used with the `nroff -s` flag or `.rd` requests, when it is necessary to insert paper manually or change fonts in the middle of a document. Instead of hitting the RETURN key in these cases, you must use the line-feed key to get any response.

In many (but not all) cases, the following two command lines are equivalent:

```
nroff -T300 files
nroff files | 300
```

Similarly, in many (but not all) cases, the following two command lines are equivalent:

```
nroff -T300 -12 files
nroff files | 300 +12
```

The use of `300` can thus often be avoided unless special delays or

options are required; in a few cases, however, the additional movement optimization of 300 may produce better-aligned output.

The neqn names of and resulting output for the Greek and special characters supported by 300 are shown in greek(5).

FILES

/usr/bin/300
/usr/bin/300s

SEE ALSO

450(1), eqn(1), mesg(1), nroff(1), stty(1), tabs(1), tbl(1), tplot(1G), greek(5).

BUGS

Some special characters cannot be correctly printed in column 1 because the print head cannot be moved to the left from there.
If your output contains Greek or reverse line-feeds, use a friction-feed platen instead of a forms tractor; although good enough for drafts, the latter has a tendency to slip when reversing direction, distorting Greek characters and misaligning the first line of text after one or more reverse line-feeds.

See 300(1)

NAME

4014 — filter text containing printer control sequences a page at a time

SYNOPSIS

4014 [-c*n*] [-n] [-p*l*] [-t] [*file*]

DESCRIPTION

The output of 4014 is intended for a Tektronix 4014 terminal; 4014 arranges for 66 lines to fit on the screen, divides the screen into *n* columns, and contributes an eight-space page offset in the (default) single-column case. Tabs, spaces, and backspaces are collected and plotted when necessary. TELETYPE Model 37 half- and reverse-line sequences are interpreted and plotted. At the end of each page, 4014 waits for a newline (empty line) from the keyboard before continuing on to the next page. In this wait state, the command ! *cmd* will send the *cmd* to the shell.

The command line options are:

-t Don't wait between pages (useful for directing output into a file).

-n Start printing at the current cursor position and never erase the screen.

-c*n* Divide the screen into *n* columns and wait after the last column.

-p*l* Set page length to *l*. *l* accepts the scale factors i (inches) and l (lines); the default is lines.

FILES

/usr/bin/4014

SEE ALSO

pr(1), tc(1), troff(1).

NAME

450 — filter text containing printer control sequences for the DASI terminal

SYNOPSIS

450

DESCRIPTION

450 supports special functions of, and optimizes the use of, the DASI 450 terminal, or any terminal that is functionally identical, such as the DIABLO 1620 or XEROX 1700. It converts half-line forward, half-line reverse, and full-line reverse motions to the correct vertical motions. It also attempts to draw Greek letters and other special symbols in the same manner as 300(1). 450 can be used to print equations neatly, in the sequence:

neqn *file* ... | nroff | 450

WARNINGS

Make sure that the PLOT switch on your terminal is ON before 450 is used. The SPACING switch should be put in the desired position (either 10- or 12-pitch). In either case, vertical spacing is 6 lines/inch, unless dynamically changed to 8 lines per inch by an appropriate escape sequence.

450 can be used with the nroff -s flag or .rd requests, when it is necessary to insert paper manually or change fonts in the middle of a document. Instead of hitting the RETURN key in these cases, you must use the line-feed key to get any response.

In many (but not all) cases, the use of 450 can be eliminated in favor of one of the following:

nroff -T450 *files* ...

or

nroff -T450 -12 *files* ...

The use of 450 can thus often be avoided unless special delays or options are required; in a few cases, however, the additional movement optimization of 450 may produce better-aligned output.

The neqn names of, and resulting output for, the Greek and special characters supported by 450 are shown in greek(5).

FILES

/usr/bin/450

SEE ALSO

300(1), eqn(1), mesg(1), nroff(1), stty(1), tabs(1), tbl(1), tplot(1G), greek(5).

BUGS

Some special characters cannot be correctly printed in column 1 because the print head cannot be moved to the left from there. If your output contains Greek and/or reverse line-feeds, use a friction-feed platen instead of a forms tractor; although good enough for drafts, the latter has a tendency to slip when reversing direction, distorting Greek characters and misaligning the first line of text after one or more reverse line-feeds.

NAME

adb — debugger

SYNOPSIS

adb [-k] [-w] [*objfil* [*corfil*]]

DESCRIPTION

adb is a general purpose debugging program. It may be used to examine files and to provide a controlled environment for the execution of A/UX programs.

objfil is normally an executable program file, preferably containing a symbol table; if not, then the symbolic features of adb cannot be used, although the file can still be examined. The default for *objfil* is a.out. *corfil* is assumed to be a core image file produced after executing *objfil*; the default for *corfil* is core.

Requests to adb are read from the standard input and responses are to the standard output. If the -w flag option is present, then both *objfil* and *corfil* are created (if necessary) and opened for reading and writing so that files can be modified using adb. adb ignores quit signals; an interrupt causes return to the next adb command.

To exit adb: use $q or $Q or CONTROL-d.

Normally, for portability, adb does a system call to gather information regarding core file relocation addresses. If using adb on a standalone program (such as the kernel, /unix), using the -k flag option skips that part of the adb code.

In general, requests to adb are of the form

[*address*] [, *count*] [*command*] [;]

If *address* is present, then "dot" is set to *address*. Initially, dot is set to 0. For most commands, *count* specifies how many times the command will be executed. The default *count* is 1. *address* and *count* are expressions.

The interpretation of an address depends on the context in which it is used. If a subprocess is being debugged, then addresses are interpreted in the usual way in the address space of the subprocess. If the operating system is being debugged either post-mortem or using the special file /dev/kmem to examine interactively and/or modify memory, the maps are set to map the kernel virtual addresses. For further details of address mapping, see ADDRESSES.

EXPRESSIONS

. The value of dot.

+ The value of dot, incremented by the current increment.

^ The value of dot, decremented by the current increment.

" The last *address* typed.

integer A number. The prefix `0` (zero) forces interpretation in octal radix; the prefixes `0d` and `0D` force interpretation in decimal radix; the prefixes `0x` and `0X` force interpretation in hexadecimal radix. Thus `020 = 0d16 = 0x10` = sixteen. If no prefix appears, then the *default radix* is used; see the `$d` command. The default radix is initially hexadecimal. The hexadecimal digits are 0123456789abcdefABCDEF, with the obvious values. Note that a hexadecimal number the most significant digit of which would otherwise be an alphabetic character must have a `0x` (or `0X`) prefix (or a leading zero, if the default radix is hexadecimal).

integer.fraction
A 32-bit floating point number.

'cccc' The ASCII value of up to 4 characters. \ may be used to escape a '.

<name The value of *name*, which is either a variable name or a register name. `adb` maintains a number of variables (see VARIABLES) named by single letters or digits. If *name* is a register name, then the value of the register is obtained from the system header in *corfil*. The register names are those printed by the `$r` command.

symbol A *symbol* is a sequence of upper or lowercase letters, underscores or digits, not starting with a digit. \ may be used to escape other characters. The value of the *symbol* is taken from the symbol table in *objfil*. An initial _ or ~ will be prefixed to *symbol*, if needed.

_ symbol In C, the "true name" of an external symbol begins with _ . It may be necessary to utter this name to distinguish it from internal or hidden variables of a program.

(exp) The value of the expression *exp*.

Monadic Operators

**exp* The contents of the location addressed by *exp* in *corfil*.

@exp The contents of the location addressed by *exp* in *objfil*.

–exp Integer negation.

~exp Bitwise complement.

#exp Logical negation.

Dyadic operators are left associative and are less binding than monadic operators.

e1+e2 Integer addition.

e1–e2 Integer subtraction.

*e1*e2* Integer multiplication.

e1%e2 Integer division.

e1&e2 Bitwise conjunction.

e1|e2 Bitwise disjunction.

e1#e2 *e1* rounded up to the next multiple of *e2*.

COMMANDS

Most commands consist of a verb followed by a modifier or list of modifiers. The following verbs are available. (The commands "?" and "/" may be followed by "*"; see ADDRESSES for further details.)

?f Locations starting at *address* in *objfil* are printed according to the format *f*. Dot is incremented by the sum of the increments for each format letter (q.v.).

/f Locations starting at *address* in *corfil* are printed according to the format *f*, and dot is incremented as for "?".

=f The value of *address* itself is printed in the styles indicated by the format *f*. (For `i` format, "?" is printed for the parts of the instruction that reference subsequent words.)

A *format* consists of one or more characters that specify a style of printing. Each format character may be preceded by a decimal integer that is a repeat count for the format character. While stepping through a format, dot is incremented by

the amount given for each format letter. If no format is given, then the last format is used. The format letters available are as follows.

i	*n*	Disassemble the addressed instruction.
o	2	Print 2 bytes in hex. All octal numbers output by adb are preceded by 0.
O	4	Print 4 bytes in octal.
q	2	Print in signed octal.
Q	4	Print long signed octal.
d	2	Print in decimal.
D	4	Print long decimal.
x	2	Print 2 bytes in hexadecimal.
X	4	Print 4 bytes in hexadecimal.
u	2	Print as an unsigned decimal number.
U	4	Print long unsigned decimal.
f	4	Print the 32-bit value as a floating point number.
F	8	Print double floating point.
b	1	Print the addressed byte in octal.
c	1	Print the addressed character.
C	1	Print the addressed character using the standard escape convention where control characters are printed as ^*X* and the delete character is printed as ^?.
s	*n*	Print the addressed characters until a zero character is reached.
S	*n*	Print a string using the ^*X* escape convention (see C above). The *n* is the length of the string including its zero terminator.
Y	4	Print 4 bytes in date format (see ctime(3)).
a	0	Print the value of dot in symbolic form. Symbols are checked to ensure that they have an appropriate type as indicated below. / global data symbol ? global text symbol = global absolute symbol
p	4	Print the addressed value in symbolic form using the same rules for symbol lookup as a.
t	0	When preceded by an integer tabs to the next appropriate tab stop. For example, 8t moves to the next 8-space tab stop.

`r` 0
Print a space.

`n` 0
Print a newline.

`"..."` 0
Print the enclosed string.

`^` Dot is decremented by the current increment. Nothing is printed.

`+` Dot is incremented by 1. Nothing is printed.

`-` Dot is decremented by 1. Nothing is printed.

newline
Repeat the previous command with a *count* of 1.

`[?/]l` *value mask*
Words starting at dot are masked with *mask* and compared with *value* until a match is found. If `L` is used, then the match is for 4 bytes at a time instead of 2. If no match is found, then dot is unchanged; otherwise, dot is set to the matched location. If *mask* is omitted, then −1 is used.

`[?/]w` *value ...*
Write the 2-byte *value* into the addressed location. If the command is `W`, write 4 bytes. Odd addresses are not allowed when writing to the subprocess address space.

`[?/]m` *b1 e1 f1* `[?/]`
New values for (*b1*, *e1*, *f1*) are recorded. If less than three expressions are given, then the remaining map parameters are left unchanged. If more than 3 expressions are given, the values of (*b2*, *e2*, *f2*) (*b3*, *e3*, *f3*) and so on, are changed. If the "?" or "/" is followed by "*", then the first segment (*b1*, *e1*, *f1*) of the mapping is skipped, and the second and subsequent segments are changed instead. (There are as many (*bn*, *en*, *fn*) triples as you have sections in your program.) If the list is terminated by "?" or "/", then the file (*objfil* or *corfil*, respectively) is used for subsequent requests. (So that, for example, "`/m?`" will cause "/" to refer to *objfil*.)

`>` *name*
Dot is assigned to the variable or register named. This command is often used in the form *constant* `>` *name*. This form of the command can be used to enter 96-bit "IEEE Extended Precision" numbers into the floating-point data registers

`fp0-fp7`. For example:

```
0x3FFF00008000000000000000 > fp0
```

puts the value 1.0 into `fp0`. When this form of the command is used, only the first 32 vits of the constant are stored in dot. See *MC68881 Floating Point Coprocessor User's Manual* (available from Motorola Literature Distribution Center, part number MC68881UM/AD), section 2.4, "Extended Real," p. 211, for a description of IEEE Extended Precision format.

! A shell is called to read the rest of the line following "!".

$*modifier*

Miscellaneous commands. The available *modifiers* are:

<*f* Read commands from the file *f*. If this command is executed in a file, further commands in the file are not seen. If *f* is omitted, the current input stream is terminated. If a *count* is given, and is zero, the command will be ignored. The value of the count will be placed in variable *9* before the first command in *f* is executed.

<<*f* Similar to < except it can be used in a file of commands without causing the file to be closed. Variable *9* is saved during the execution of this command, and restored when it completes. There is a (small) finite limit to the number of << files that can be open at once.

>*f* Append output to the file *f*, which is created if it does not exist. If *f* is omitted, output is returned to the terminal.

? Print process ID, the signal which caused stoppage or termination, as well as the registers as `$r`. This is the default if *modifier* is omitted.

`r` Print the general registers and the instruction addressed by `pc`. Dot is set to `pc`.

`f` Print the floating point data registers `fp0-fp7` in IEEE Extended Precision (see >*name*, above, for definition), and exponential notation, along with the floating-point control registers `fpcr`, `fpsr`, and `fpiar`.

`b` Print all breakpoints and their associated counts and commands.

`c` C stack backtrace. If *address* is given, then it is taken as the address of the current frame (instead of `a7`). If `C` is used, then the names and (16 bit) values of all au-

tomatic and static variables are printed for each active function. If *count* is given, then only the first *count* frames are printed.

- `d` Set the default radix to *address* and report the new value. Note that *address* is interpreted in the (old) current radix. Thus `10$d` never changes the default radix. To make decimal the default radix, use `0t10$d`.
- `e` The names and values of external variables are printed.
- `w` Set the page width for output to *address* (default 80).
- `s` Set the limit for symbol matches to *address* (default 255).
- `o` Regard all integers subsequently input as octal.
- `d` Reset integer input as described in EXPRESSIONS.
- `q` Exit from `adb`.
- `v` Print all nonzero variables in hexadecimal.
- `m` Print the address map.

:*modifier*

Manage a subprocess. Available modifiers are:

- `b`*c* Set breakpoint at *address*. The breakpoint is executed *count*–1 times before causing a stop. Each time the breakpoint is encountered the command *c* is executed. If this command is omitted or sets *dot* to zero then the breakpoint causes a stop.
- `d` Delete breakpoint at *address*.
- `r` Run *objfil* as a subprocess. If *address* is given explicitly then the program is entered at this point; otherwise, the program is entered at its standard entry point. *count* specifies how many breakpoints are to be ignored before stopping. Arguments to the subprocess may be supplied on the same line as the command. An argument starting with < or > causes the standard input or output to be established for the command. All signals are turned on on entry to the subprocess.
- `c`*s* The subprocess is continued with signal *s* (see `signal(3)`). If *address* is given, then the subprocess is continued at this address. If no signal is specified, then the signal that caused the subprocess to stop is sent. Breakpoint skipping is the same as for `r`.
- `s`*s* As for `c` except that the subprocess is single stepped *count* times. If there is no current subprocess then *objfil*

is run as a subprocess as for `r`. In this case no signal can be sent; the remainder of the line is treated as arguments to the subprocess.

`k` The current subprocess, if any, is terminated.

VARIABLES

`adb` provides a number of variables. Named variables are set initially by `adb` but are not used subsequently. Numbered variables are reserved for communication as follows.

0 The last value printed.
1 The last offset part of an instruction source.
2 The previous value of variable 1.
9 The count on the last `$<` or `$<<` command.

On entry, the following are set from the system header in the *corfil*. If *corfil* does not appear to be a `core` file, then these values are set from *objfil*.

b The base address of the data segment.
d The data segment size.
e The entry point.
m The "magic" number (0407, 0410 , 0413).
s The stack segment size.
t The text segment size.

ADDRESSES

The address in a file associated with a written address is determined by a mapping associated with that file. Each mapping is represented by *n* triples (*b1*, *e1*, *f1*), (*b2*, *e2*, *f2*), ... (*bn*, *en*, *fn*), corresponding to the number of sections in your object file, and the *file address* corresponding to a written *address* is calculated as follows.

$b1 \le address < e1 \Rightarrow$ *file address*$= address + f1 - b1$, otherwise,

$b2 \le address < e2 \Rightarrow$ *file address*$= address + f2 - b2$, and so on,

otherwise, the requested *address* is not legal. In some cases (e.g., for programs with separated I and D space), the two segments for a file may overlap. If a `?` or `/` is followed by an `*`, then the first triple is not used.

The initial setting of both mappings is suitable for normal `a.out` and `core` files. If either file is not of the kind expected, then for that file *b1* is set to 0, *e1* is set to the maximum file size and *f1* is set to 0; in this way the whole file can be examined with no address translation.

So that adb may be used on large files, all appropriate values are kept as signed 32-bit integers.

EXAMPLES

```
adb obj1
```

will invoke adb with the executable object obj1. When adb responds with either

```
a.out file
ready
```

or

```
a.out (COFF format)
a.out a.out
ready
```

the request:

```
main,10?ia
```

will cause 16 (10 hex) instructions to be printed in assembly code, starting from location main.

FILES

```
/bin/adb
a.out
core
```

SEE ALSO

sdb(1), a.out(4), core(4).

DIAGNOSTICS

Echoes adb when there is no current command or format. Produces comments about inaccessible files, syntax errors, abnormal termination of commands, etc. Exit status is 0, unless last command failed or returned nonzero status.

BUGS

Use of # for the unary logical negation operator is peculiar.

There doesn't seem to be any way to clear all breakpoints.

In certain cases, disassembled code cannot be used directly as input to as. This is because adb gives more useful information than as accepts. For example, explicit register names are given in the disassembly of movm and fmovm instructions.

NAME

addbib — create or extend bibliographic database

SYNOPSIS

addbib [-p *promptfile*] [-a] *database*

DESCRIPTION

addbib initiates or furthers a bibliography, entered as a database. The database structure allows formatting to be imposed as a step separate from data entry; data entry must be performed only once. Database entries consist of keyletters and relevant fields. For example, %A is the keyletter for *author-name*, and Bill Tuthill (the author of refer) could fill in this field. (Further examples are given below.) Once entered, entries may be culled from the database easily and in the proper format. The refer program handles this compilation; you do not have to look up entries by hand.

When this program starts up, answering y to the initial Instructions? prompt yields directions; typing n or RETURN skips them. addbib then prompts for various bibliographic fields, reads responses from the terminal, and sends output records to a *database*. A null response (just RETURN) means to leave out that field. A minus sign (-) means to go back to the previous field. A trailing backslash allows a field to be continued on the next line. The repeating Continue? prompt allows the user either to resume by typing y or RETURN, to quit the current session by typing n or q, or to edit the *database* with any system editor (vi, ex, edit, ed), by typing its name.

The -a flag option suppresses prompting for an abstract; asking for an abstract is the default. Abstracts are ended with a CONTROL-D. The -p flag option causes addbib to use a new prompting skeleton, defined in *promptfile*. This file should contain prompt strings, a tab, and the keyletters to be written to the *database*.

The most common keyletters and their meanings are given below. addbib insulates you from these keyletters, since it gives you prompts in English, but if you edit the bibliography file later on, you will need to know this information.

- %A Author's name
- %B Book containing article referenced
- %C City (place of publication)

%D Date of publication
%E Editor of book containing article referenced
%F Footnote number or label (supplied by `refer`)
%G Government order number
%H Header commentary, printed before reference
%I Issuer (publisher)
%J Journal containing article
%K Keywords to use in locating reference
%L Label field used by `-k` option of `refer`
%N Number within volume
%O Other commentary, printed at end of reference
%P Page number(s)
%Q Corporate or Foreign Author (unreversed)
%R Report, paper, or thesis (unpublished)
%S Series title
%T Title of article or book
%V Volume number
%X Abstract; used by `roffbib`, not by `refer`
%Y ignored by `refer`
%Z ignored by `refer`

Except for `A`, each field should be given just once. Only relevant fields should be supplied. An example is:

```
%A    John Smith
%T    Using A/UX
%I    Apple Computer
%C    New York
%D    1987
```

FILES

`/usr/ucb/addbib`

promptfile
optional file to define prompting

CAVEATS

The length of the prompt strings in a user-defined prompt file should be less than or equal to 20 characters. That is, `addbib` will only display the first 20 characters. If the prompt string is longer than 20 characters, `addbib` will append the keyletter from the prompt file to the end of the truncated prompt string.

SEE ALSO

indxbib(1), lookbib(1), refer(1), roffbib(1), sortbib(1).

NAME

admin — create and administer SCCS files

SYNOPSIS

admin [-a*login*] [-d*flag*[*flag-val*]] [-e*login*] [-f*flag*[*flag-val*]] [-h] [-i[*name*]] [-m[*mrlist*]] [-n] [-r*rel*[.*lev*]] [-t[*name*]] [-y[*comment*]] [-z] *file...*

DESCRIPTION

admin is used to create new SCCS files and change parameters of existing ones. Arguments to admin, which may appear in any order, consist of keyletter arguments, which begin with –, and named files (note that SCCS file names must begin with the characters s.). If a named file does not exist, it is created, and its parameters are initialized according to the specified keyletter arguments. Parameters not initialized by a keyletter argument are assigned a default value. If a named file does exist, parameters corresponding to specified keyletter arguments are changed, and other parameters are left as is.

If a directory is named, admin behaves as though each file in the directory were 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 keyletter arguments are as follows. Each is explained as though only one named file is to be processed since the effects of the arguments apply independently to each named file.

-n This keyletter indicates that a new SCCS file is to be created.

-i[*name*] The *name* of a file from which the text for a new SCCS file is to be taken. The text constitutes the first delta of the file (see -r keyletter for delta numbering scheme). If the i keyletter is used, but the file name is omitted, the text is obtained by reading the standard input until an end-of-file is encountered. If this keyletter is omitted, then the SCCS file is created empty (in this case, you need to explicitly use the -n flag). Only one SCCS file may be created by an admin command on which the i keyletter is supplied. Using a single admin to create two or

more SCCS files requires that they be created empty (no -i keyletter). Note that the -i keyletter implies the -n keyletter.

-r*rel*[.*lev*] *rel* is the release and *lev* is the level into which the initial delta is inserted. This keyletter may be used only if the -i keyletter is also used. If the -r keyletter is not used, the initial delta is inserted into release 1. The default level of the initial delta is 1 (by default initial deltas are named 1.1).

-t[*name*] The *name* of a file from which descriptive text for the SCCS file is to be taken. If the -t keyletter is used and admin is creating a new SCCS file (the -n and/or -i keyletters also used), the descriptive text filename must also be supplied. In the case of existing SCCS files: (1) a -t keyletter without a filename causes removal of descriptive text (if any) currently in the SCCS file, and (2) a -t keyletter with a filename causes text (if any) in the named file to replace the descriptive text (if any) currently in the SCCS file.

-f*flag* [*flag-val*]
This keyletter specifies a *flag*, and, possibly, a value for the *flag*, to be placed in the SCCS file. Several f keyletters may be supplied on a single admin command line. The allowable *flags* and their values are:

b Allows use of the -b keyletter on a get(1) command to create branch deltas.

c*ceil* *ceil* is the highest release (i.e., "ceiling"), a number less than or equal to 9999, which may be retrieved by a get(1) command for editing. The default value for an unspecified c flag is 9999.

f*floor* *floor* is the lowest release (i.e., "floor"), a number greater than 0 but less than 9999, which may be retrieved by a get(1) command for editing. The default value for an unspecified f flag is 1.

d*SID* The default delta number (SID) to be used by a get(1) command.

i[*str*] Causes the No id keywords (ge6) message issued by get(1) or delta(1) to be treated as a fatal error. In the absence of this flag, the message is only a warning. The message is issued if no SCCS identification keywords (see get(1)) are found in the text retrieved or stored in the SCCS file. If *str* is supplied, the keywords must exactly match the given string. *str* must contain a keyword and no embedded newlines.

j Allows concurrent get(1) commands for editing on the same SID of an SCCS file. This allows multiple concurrent updates to the same version of the SCCS file.

l*list* A *list* of releases to which deltas may no longer be made (get -e against one of these "locked" releases fails). The *list* has the following syntax:

<list> ::= *<range>* | *<list>* , *<range>*
<range>::= *RELEASE NUMBER* | a

The character a in the *list* is equivalent to specifying *all releases* for the named SCCS file.

n Causes delta(1) to create a "null" delta in each of those releases (if any) being skipped when a delta is made in a *new* release (e.g., in making delta 5.1 after delta 2.7, releases 3 and 4 are skipped). These null deltas serve as "anchor points" so that branch deltas may later be created from them. The absence of this flag causes skipped releases to be nonexistent in the SCCS file, preventing branch deltas from be-

ing created from them in the future.

q*text* User-definable *text* substituted for all occurrences of the %Q% keyword in SCCS file text retrieved by get(1).

m*mod* *mod* is the module name of the SCCS file which is substituted for all occurrences of the %M% keyword in SCCS file text retrieved by get(1). If the m flag is not specified, the value assigned is the name of the SCCS file with the leading s. removed.

t*type* *type* of module in the SCCS file substituted for all occurrences of %Y% keyword in SCCS file text retrieved by get(1).

v[*pgm*] Causes delta(1) to prompt for Modification Request (MR) numbers as the reason for creating a delta. *pgm* specifies the name of an MR number validity checking program (see delta(1)). (If this flag is set when creating an SCCS file, the m keyletter must also be used even if its value is null).

−d*flag* Causes removal (deletion) of the specified *flag* from an SCCS file. The −d keyletter may be specified only when processing existing SCCS files. Several −d keyletters may be supplied on a single admin command. See the −f keyletter for allowable *flag* names.

l*list* A *list* of releases to be "unlocked". See the −f keyletter for a description of the l flag and the syntax of a *list*.

−a*login* A *login* name, or numeric A/UX system group ID, to be added to the list of users which may make deltas (changes) to the SCCS file. A group ID is equivalent to specifying all login names common to that group ID. Several a keyletters may be used on a single admin command line. As many *login*s, or

numeric group IDs, as desired may be on the list simultaneously. If the list of users is empty, then anyone may add deltas. If *login* or group ID is preceded by a ! they are to be denied permission to make deltas.

–e*login* A *login* name, or numeric group ID, to be erased from the list of users allowed to make deltas (changes) to the SCCS file. Specifying a group ID is equivalent to specifying all *login* names common to that group ID. Several `e` keyletters may be used on a single `admin` command line.

–y[*comment*] The *comment* text is inserted into the SCCS file as a comment for the initial delta in a manner identical to that of `delta`(1). Omission of the `-y` keyletter results in a default comment line being inserted in the form:

```
date and time created YY/MM/DD HH:MM:SS
```

by `login`. If the comment contains spaces, you must enclose the entire comment in double quotes.

The –y keyletter is valid only if the `-i` and/or `-n` keyletters are specified (i.e., a new SCCS file is being created).

–m[*mrlist*] *mrlist*, the list of Modification Requests (MR) numbers, is inserted into the SCCS file as the reason for creating the initial delta in a manner identical to `delta`(1). The `v` flag must be set and the MR numbers are validated if the `v` flag has a value (the name of an MR number validation program). Error diagnostics will appear if the `v` flag is not set or if MR validation fails.

–h Causes `admin` to check the structure of the SCCS file (see `sccsfile`(4)), and to compare a newly computed **checksum** (the sum of all the characters in the SCCS file except those in the first line) with the checksum that is stored in the first line of the SCCS file. Appropriate error diagnostics are produced.

This keyletter inhibits writing on the file, so that it nullifies the effect of any other keyletters supplied,

and is, therefore, only meaningful when processing existing files.

`-z` The SCCS file checksum is recomputed and stored in the first line of the SCCS file (see `-h`, above).

Note that use of this keyletter on a truly corrupted file may prevent future detection of the corruption.

The last component of all SCCS file names must be of the form `s.`*file-name*. New SCCS files are given mode 444 (see `chmod`(1)). Write permission in the pertinent directory is, of course, required to create a file. All writing done by `admin` is to a temporary x-file, called `x.`*file-name*, (see `get`(1)), created with mode 444 if the `admin` command is creating a new SCCS file, or with the same mode as the SCCS file if it exists. After successful execution of `admin`, the SCCS file is removed (if it exists), and the x-file is renamed with the name of the SCCS file. This ensures that changes are made to the SCCS file only if no errors occurred.

It is recommended that directories containing SCCS files be mode 755 and that SCCS files themselves be mode 444. The mode of the directories allows only the owner to modify SCCS files contained in the directories. The mode of the SCCS files prevents any modification at all except by SCCS commands.

If it should be necessary to patch an SCCS file for any reason, the mode may be changed to 644 by the owner allowing use of `ed`(1). Care must be taken! The edited file should *always* be processed by an `admin -h` to check for corruption followed by an `admin -z` to generate a proper check-sum. Another `admin -h` is recommended to ensure the SCCS file is valid.

`Admin` also makes use of a transient lock file (called `z.`*file-name*), which is used to prevent simultaneous updates to the SCCS file by different users. See `get`(1) for further information.

EXAMPLES

`admin -i`*file1* `s.`*file1*

creates a new file in SCCS format named `s.`*file1*, from *file1*.

FILES

`/usr/bin/admin`

SEE ALSO

`delta`(1), `ed`(1), `get`(1), `help`(1), `prs`(1), `what`(1), `sccsfile`(4).
"SCCS Reference" in *A/UX Programming Languages and Tools, Volume 2.*

DIAGNOSTICS

Use `help`(1) for explanations.

NAME

apply — apply a command to a set of arguments

SYNOPSIS

apply [-ac] [-*n*] *command args...*

DESCRIPTION

apply runs the named *command* on each argument *arg* in turn. Normally arguments are chosen singly; the optional number *n* specifies the number of arguments to be passed to *command*. If *n* is zero, *command* is run without arguments once for each *arg*. Character sequences of the form %*d* in *command*, where *d* is a digit from 1 to 9, are replaced by the d*th* following unused *arg*. If any such sequences occur, *n* is ignored, and the number of arguments passed to *command* is the maximum value of *d* in *command*. The character % may be changed by the -a flag option.

EXAMPLES

The command

```
apply echo *
```

is similar to ls(1);

```
apply -2 cmp a1 b1 a2 b2
```

compares the a files to the b files;

```
apply -0 who 1 2 3 4 5
```

runs who(1) 5 times; and

```
apply 'ln %1 /usr/mcfong' *
```

links all files in the current directory to the directory /usr/mcfong.

FILES

/usr/ucb/apply

SEE ALSO

csh(1), ksh(1), sh(1), xargs(1).

BUGS

Shell metacharacters in *command* may have bizarre effects; it is best to enclose complicated commands in single quotes '.

There is no way to pass a literal %2 if % is the argument expansion character.

NAME

apropos — locate commands by keyword lookup

SYNOPSIS

apropos *keyword...*

DESCRIPTION

apropos shows which online manual sections contain in their title instances of any of the given keywords. Each keyword is considered separately and the case of letters is ignored. Words that are part of other words are considered; thus, looking for "compile" will hit all instances of "compiler" also.

EXAMPLES

Try running the following two commands.

```
apropos password
apropos editor
```

If a resulting output line starts with a string of the form *name* (*section*), then you can run the command

```
man section name
```

to get the on-line documentation for it. For instance, try

```
apropos format
```

and then

```
man 3s printf
```

to get the manual on the subroutine printf.

FILES

/usr/ucb/apropos	
/usr/lib/whatis	data base

SEE ALSO

man(1), whatis(1).

NAME

ar — archive and library maintainer for portable archives

SYNOPSIS

ar *key* [clsuv] [abi *posname*] *afile name* ...

DESCRIPTION

ar maintains groups of files combined into a single archive file. Its main use is to create and update library files as used by the link editor, though it can be used for any similar purpose.

When ar creates an archive, it creates headers in a format that is portable across all machines. The portable archive format and structure are described in detail in ar(4). The archive symbol table (described in ar(4)) is used by the link editor (ld(1)) to effect multiple passes over libraries of object files in an efficient manner. Whenever the ar(1) command is used to create or update the contents of an archive, the symbol table is rebuilt. The symbol table can be forcibly rebuilt by the s option, described later in this section.

key is one character from the set dmpqrtx. Optionally, *key* may be concatenated with one or more of the letters in the group clsuv or in the group abi.

afile is the archive file.

The *names* are constituent files in the archive file.

The meanings of the *key* characters are:

d Deletes the named files from the archive file.

m Moves the named files to the end of the archive. If a positioning character is present, then the *posname* argument must be present and, as in the r key character, specifies where the files are to be moved.

p Prints the named files in the archive.

q Quickly appends the named files to the end of the archive file. Optional positioning characters are invalid. The q command does not check whether the added members are already in the archive and is useful only to avoid quadratic behavior when creating a large archive piece-by-piece.

r Replaces the named files in the archive file; that is, if the named file is not already included in the archive file, then it will be placed into the archive. New files are placed at the

end of the archive, unless the default positioning is overridden by the specification of an optional positioning character from the set `abi`. The meaning of these positional characters is as follows:

`a` *posname*
Places the named file or files after *posname*.

`b` *posname*
Places the named file or files before *posname*.

`i` *posname*
Inserts the named file or files before *posname*. This is identical to the `b` positioning character.

`t` Prints a table of contents of the archive file. If no names are given, all files in the archive are tabled. If names are given, only those files are tabled.

`x` Extracts the named files. If no names are given, all files in the archive are extracted. In either case, `x` does not alter the archive file.

Any of the flags `clsuv` may be included in addition to a key letter. These flags have the following meaning.

`c` Create. Normally `ar` creates *afile* when it needs to. The create flag option suppresses the normal message that is produced when *afile* is created.

`l` Local. Normally `ar` places its temporary files in the directory `/tmp`. This flag option causes them to be placed in the local directory.

`s` Symbol table creation. Forces the regeneration of the archive symbol table even if `ar`(1) is not invoked with a command which will modify the archive contents. This command is useful to restore the archive symbol table after the `strip`(1) command has been used on the archive.

`u` Update. When used in conjunction with the `r` key letter, only those files whose modification dates are more recent than the modification date of the named archive *afile* are replaced in the archive.

`v` Verbose. Under the verbose flag option, `ar` gives a file-by-file description of the making of a new archive file from the old archive and the constituent files. When used with `t`, it gives a long listing of all information about the files. When

used with x, it precedes each file with a name.

EXAMPLES

The command

```
ar rc libfoo.a foo.o
```

creates an archive file containing the object code `foo.o`.

```
ar r libfoo.a bar.o
```

adds the binary file `bar.o` to the end of the archive `libfoo.a`.

```
ar ru libfoo.a *.o
```

replaces in the archive file `libfoo.a` any object modules in the current directory that were modified after the most recent modification of the archive file itself.

```
ar rb bar.o libfoo.a new.o
```

inserts the object file `new.o` into the archive `libfoo.a` before the already archived object file `bar.o`.

FILES

`/bin/ar`	
`/tmp/ar*`	temporary files

SEE ALSO

ld(1), lorder(1), strip(1), tar(1), a.out(4), ar(4).

BUGS

If the same file is mentioned twice in an argument list, it may be put in the archive twice.

NAME

as — common assembler

SYNOPSIS

as [–m] [–n] [–o *objfile*] [–R] [–V] *filename*

DESCRIPTION

The as command assembles *filename*. The following flags may be specified in any order.

–o *objfile* Put the output of assembly in *objfile*. By default, the output filename is formed by removing the .s suffix, if there is one, from the input filename and appending a .o suffix.

–n Turn off long/short address optimization. By default, address optimization takes place.

–m Run the m4 macro pre-processor on the input to the assembler.

–R Remove (unlink) the input file after assembly is completed. This flag option is off by default.

–V Write the version number of the assembler being run on the standard error output.

FILES

/bin/as

/usr/tmp/as[1-6]*XXXXXX* temporary files

SEE ALSO

adb(1), ld(1), m4(1), nm(1), strip(1), a.out(4).

"as Reference" in *A/UX Programming Languages and Tools, Volume 1*.

WARNINGS

If the –m flag option is used, keywords for m4 cannot be used as symbols (variables, functions, labels) in the input file because m4 cannot determine which are assembler symbols and which are real m4 macros (see m4(1)).

BUGS

Arithmetic expressions are permitted to have only one forward referenced symbol per expression.

NAME

asa — interpret ASA carriage control characters

SYNOPSIS

asa [*file*...]

DESCRIPTION

asa interprets the output of Fortran programs that utilize ASA carriage control characters. It processes either the *file*s whose names are given as arguments or the standard input if no file names are supplied. The first character of each line is assumed to be a control character; their meanings are:

(blank) single newline before printing

0 double newline before printing

1 new page before printing

\+ overprint previous line.

Lines beginning with other than the above characters are treated as if they began with `The first character of a line is` *not* printed. If any such lines appear, an appropriate diagnostic will appear on standard error. This program forces the first line of each input file to start on a new page.

EXAMPLES

To correctly view the output of Fortran programs which use ASA carriage control characters, asa could be used as a filter, thus:

```
a.out | asa | lp
```

and the output, properly formatted and paginated, would be directed to the line printer. Fortran output sent to a file could be viewed by:

```
asa file
```

FILES

/bin/asa

SEE ALSO

efl(1), f77(1), fpr(1), fsplit(1).

NAME

at, batch — execute commands at a later time

SYNOPSIS

at *time* [*date*] [+*increment*]
at -l [*job...*]
at -r *job...*
batch

DESCRIPTION

at and batch read commands from standard input to be executed at a later time. at allows you to specify when the commands should be executed, while jobs queued with batch will execute when system load level permits. at -r removes jobs previously scheduled with at. The -l flag option reports all jobs scheduled for the invoking user.

Standard output and standard error output are mailed to the user via mail(1) unless they are redirected elsewhere. The shell environment variables, current directory, umask, and ulimit are retained when the commands are executed. Open file descriptors, traps, and priority are lost.

Users are permitted to use at if their name appears in the file /usr/lib/cron/at.allow. If that file does not exist, the file /usr/lib/cron/at.deny is checked to determine if the user should be denied access to at. If neither file exists, only root is allowed to submit a job. The allow/deny files consist of one user name per line.

The *time* may be specified as 1, 2, or 4 digits. One and two digit numbers are taken to be hours, four digits to be hours and minutes. The time may alternately be specified as two numbers separated by a colon, meaning *hour:minute*. A suffix am or pm may be appended; otherwise a 24-hour clock time is understood. The suffix zulu may be used to indicate GMT. The special names noon, midnight, now, and next are also recognized.

An optional *date* may be specified as either a month name followed by a day number (and possibly year number preceded by an optional comma) or a day of the week (fully spelled or abbreviated to three characters). Two special days, today and tomorrow are recognized. If no *date* is given, today is assumed if the given hour is greater than the current hour and tomorrow is assumed if it is less. If the given month is less than the current month (and no year is given), next year is assumed.

The optional *increment* is simply a number suffixed by one of the following: `minutes`, `hours`, `days`, `weeks`, `months`, or `years`. (The singular form is also accepted.)

Thus legitimate commands include:

```
at 0815am Jan 24
at 8:15am Jan 24
at now + 1 day
at 5 pm Friday
```

`at` and `batch` write the job number and schedule time to standard error.

`batch` submits a batch job. It is similar to `at now`, but goes into a different queue, and will respond more promptly with any error messages.

`at -r` removes jobs previously scheduled by `at` or `batch`. The job number is the number given to you previously by the `at` or `batch` command. You can also get job numbers by typing `at -l`. You can remove only your own jobs unless you are the superuser.

EXAMPLES

The `at` and `batch` commands read from standard input the commands to be executed at a later time. `sh`(1) provides different ways of specifying standard input. Within your commands, it may be useful to redirect standard output.

This sequence can be used at a terminal:

```
batch
nroff filename > outfile
```
CONTROL-d

This sequence, which demonstrates redirecting standard error to a pipe, is useful in a shell procedure (the sequence of output redirection specifications is significant):

```
batch <<!
nroff filename 2>&1 > outfile | mail loginid
!
```

To have a job reschedule itself, invoke `at` from within the shell procedure, by including code similar to the following within the shell file:

```
echo "sh shellfile" | at 1900 thursday next week
```

If the machine is down at the scheduled time, the job is not run.

FILES

/usr/bin/at	
/usr/bin/batch	
/usr/lib/atrun	executor (run by cron(1M)
/usr/lib/cron	main cron directory
/usr/lib/cron/at.allow	list of allowed users
/usr/lib/cron/at.deny	list of denied users
/usr/lib/cron/queuedefs	scheduling information
/usr/spool/cron/atjobs	spool area

SEE ALSO

crontab(1), kill(1), mail(1), nice(1), ps(1), sh(1), cron(1M).

DIAGNOSTICS

Complaints about various syntax errors and times out of range.

NAME

`atlookup` — look up network visible entities (NVEs) registered on the AppleTalk internet

SYNOPSIS

`atlookup` [`-d`] [`-r` *nn*] [`-s` *nn*] [`-x`] [*object*[:*type*[@*zone*]]]

`atlookup -z` [`-C`]

DESCRIPTION

The `atlookup` command uses the Name Binding Protocol (NBP) to look up names and addresses of the specified NVEs.

The default is to look up all the entities (of all types) in the current zone. Specifying the *object*, *type*, or *zone* on the command line changes the scope of lookup. The command line arguments and parameters are

`-d` Print network address using decimal numbers.

`-r` *nn* Retry lookup *nn* times. The default is to try the lookup eight times.

`-s` *nn* Retry lookup every *nn* seconds. The default is to space retries one second apart.

`-x` Print the 8-bit ASCII characters on output as hexadecimal numbers of the form \XX (where X is a hex digit). This is useful when you are using a terminal other than the A/UX system console.

object Name of the object to be looked up.

type Type of the object to be looked up.

zone Zone in which the lookup is to be performed. You can use an asterisk instead of a specific zone name to indicate the current zone name. If you don't specify a zone name, the current zone is the default.

`-z` Lists all zones in the internet.

`-C` Prints zones in multiple columns.

The *object* and *type* arguments may contain wildcard characters. The symbol '=' indicates a wildcard lookup. For wildcard lookups to work correctly with all nodes any wildcard character specified must be the only character in the string. However, AppleTalk Phase 2 nodes also honor a single embedded wildcard character '='. Under this scheme one wildcard character may appear anywhere in the string and may match zero or more characters. Note,

however, that although an embedded '=' is acceptable in *object* and *type* arguements of `atlookup`, only the nodes implementing AppleTalk Phase 2 protocols respond to such a query. Hence the list of NVEs obtained may be incomplete.

Information about the NVEs is displayed in a table format, one line per NVE, containing the *object*, *type*, and *zone* names, and the network, node, and socket numbers.

EXAMPLES

This command looks up all NVEs registered on the local AppleTalk zone:

```
atlookup
```

In response the system displays output similar to this:

```
Found 5 entries in zone My-Zone
6b5b.c3.ea 3-Eyed Monster:LaserWriter
6b5b.80.fd 3-Eyed Monster Spooler:LaserWriter
6b14.84.ea Incognito :LaserWriter
6b19.a3.fd Light of Day:AFPServer
6b51.27.fd Nets-R-Us Spooler:LaserWriter
```

In an Extended AppleTalk network the command

```
atlookup L=y:=
```

displays all NVEs (of any type) in the current zone that have names starting with an *L* and ending in a *y*. For example, the output might be similar to this:

```
Found 1 entries in zone My-Zone
6b19.a3.fd Light of Day:AFPServer
```

FILES

`/usr/bin/atlookup`

SEE ALSO

`at_cho_prn`(1), `atprint`(1), `atstatus`(1); *Inside AppleTalk.*

NAME

atprint — copy data to a remote PAP server

SYNOPSIS

atprint [*object*[:*type*[@*zone*]]]

DESCRIPTION

The atprint command opens a Printer Access Protocol (PAP) AppleTalk connection to a remote PAP server, such as a LaserWriter, and then copies its standard input to the remote server until it reaches an end-of-file. You can use this command to send "raw" PostScript code to a LaserWriter or to send ASCII text to an ImageWriter. (This command won't engage in a dialog with a LaserWriter about fonts or load PostScript header files in the same manner as a Macintosh Operating System does.)

The destination PAP server is chosen as follows: If you specify a PAP server using *object*, *type*, and *zone* fields on the command line, the system uses that PAP server. Otherwise, atprint goes by default to the system wide default printer set via the Macintosh Toolbox Chooser or the at_cho_prn(1) utility.

If the atprint command succeeds in establishing a connection with the server, it sends its input there. If you specify only the name, *object*, of the printer and leave out the *type*, atprint uses LaserWriter by default. An asterisk for *zone* indicates the local zone.

The atprint command outputs a message indicating which server it is connected to before it transfers data.

EXAMPLES

This command preprocesses the file and then pipes the PostScript output to joe's printer:

```
enscript -p- filename | atprint "joe's printer"
```

WARNINGS

The atprint command does not interpret contents of input files. To print properly on a PostScript printer, ASCII files must be preprocessed through pstext(1) or enscript(1), and troff-formatted files must be pre-processed through psdit(1).

FILES

/usr/bin/atprint

SEE ALSO

at_cho_prn(1), atlookup(1), atstatus(1), enscript(1), psdit(1); pstext(1), *Inside AppleTalk; Inside PostScript/LaserWriter;* "AppleTalk Programming Guide," in *A/UX Network Applications Programming;* "Installing and Administering AppleTalk," in *A/UX Network System Administration.*

NAME

atstatus — display status from a PAP server

SYNOPSIS

atstatus [*object*[:*type*[@*zone*]]]

DESCRIPTION

atstatus gets the status string from an AppleTalk Printer Access Protocol (PAP) server, such as a LaserWriter. The command parameters are

object The name of the PAP server. Wildcards are not permitted. If you don't specify the PAP server, atstatus uses the system default. If the name contains spaces, put quotes around the name. Here is an example:

```
atstatus "Sharon's Print Shop"
```

type The type of the server. If you don't specify the *type*, the default is LaserWriter. If you specify a *zone*, you must specify a *type*.

zone The zone of the PAP server. If you don't specify the *zone*, the system defaults to *, your local zone.

FILES

/usr/bin/atstatus

SEE ALSO

at_cho_prn(1), atlookup(1), atprint(1),
Inside AppleTalk.

NAME

at_cho_prn — choose a default printer on the AppleTalk® internet

SYNOPSIS

at_cho_prn [*type*[*@zone*]]

DESCRIPTION

The at_cho_prn command displays a list of printer selections and saves the name of the printer that you select. The at_cho_prn command checks the network to determine which printers are registered on that network. If you don't enter the *zone* part of the argument on the command line, at_cho_prn lists all the zones in the internet and prompts you to choose the zone in which you'd like to select your default printer.

After you specify the zone, at_cho_prn lists the printers (of type *type*) available in that zone. If you don't use the *type* argument on the command line, at_cho_prn displays all entities of the types LaserWriter and ImageWriter. The system prompts you to select a printer by entering the appropriate number from the printer list display.

EXAMPLES

The command

```
at_cho_prn 'LaserWriter@*'
```

produces output similar to this:

```
ITEM  NET-ADDR     OBJECT : TYPE
  1: 56bf.af.fc AnnLW:LaserWriter
  2: 56bf.ac.cc TimLW:LaserWriter

ITEM number (0 to make no selection)?
```

NET-ADDR is the AppleTalk internet address of the printer's listener socket, printed in hexadecimal.

OBJECT:TYPE is the name of the registered printer and its type.

FILES

/usr/bin/at_cho_prn

SEE ALSO

atlookup(1), atprint(1), atstatus(1); *Inside AppleTalk;* "Installing and Administering AppleTalk," in *A/UX Network System Administration;* "AppleTalk Programming Guide," in *A/UX Network Applications Programming.*

NAME

at_server — a generic Printer Access Protocol (PAP) server

SYNOPSIS

at_server -c *command* -o *object* [-t *type*]

DESCRIPTION

The at_server server is a simple generic PAP server. It opens a PAP server AppleTalk® socket and registers itself on the local server with the name

object:*type*@*

When an incoming PAP request is received, at_server forks a process to read the data from the remote client and executes a command from the *command* parameter. Incoming data from the server is written to a pipe that can be read by the command as standard input. Note that the server is "one-way"; it only reads from the remote client. It does not engage in a dialogue with a client in the same manner that a LaserWriter does with the Macintosh® Operating System.

The parameters that at_server takes are

command A shell command to be executed when an incoming connection is requested.

object The object name of the server; this is required and must not be a wildcard (=).

type The type name of the server (this is optional). If *type* is omitted, it defaults to LaserWriter. Wildcards are not permitted.

EXAMPLES

```
at_server -c 'lp -dfalls -s' -o lori -t LaserWriter &
```

creates a PAP server of type LaserWriter called lori that will accept incoming PAP requests from the network. Each request will have its data spooled locally by lp for printing on printer falls.

```
at_printer -o lori -t LaserWriter < x.list
```

sends x.list to this server to be printed.

```
at_printer -o = -t LaserWriter < x.list
```

sends x.list to any available LaserWriter printer.

FILES

/usr/bin/at_server

SEE ALSO

at_cho_prn(1), at_nvelkup(1), at_printer(1), at_status(1); *Inside AppleTalk.*

NAME

awk — pattern scanning and processing language

SYNOPSIS

awk [-f *file*...] [-Fc] [*prog*] [*parameters*] [*file*...]

DESCRIPTION

awk scans each input *file* for lines that match any of a set of patterns specified in *prog*. With each pattern in *prog* there can be an associated action that will be performed when a line of a *file* matches the pattern. The set of patterns may appear literally as *prog*, or in a file specified as -f *file*. The *prog* string should be enclosed in single quotes (') to protect it from the shell.

parameters, in the form x=... y=... etc., may be passed to awk.

Files are read in order; if there are no files, the standard input is read. The filename - means the standard input. Each line is matched against the pattern portion of every pattern-action statement; the associated action is performed for each matched pattern.

An input line is made up of fields separated by white space. (This default can be changed by using FS; see below). The fields are denoted $1, $2, ...; the variable $0 refers to the entire line.

A pattern-action statement has the form:

pattern { *action* }

A missing *action* means print the line; a missing *pattern* always matches. An *action* is a sequence of statements. A statement can be one of the following:

```
if ( conditional ) statement [ else statement ]
while ( conditional ) statement
for ( expression ; conditional ; expression ) statement
break
continue
{ [ statement ] ... }
variable = expression
print [ expression-list ] [ >expression ]
printf format [ , expression-list ] [ >expression ]
next
exit
```

Statements are terminated by semicolons, newlines, or right braces. An empty *expression-list* stands for the whole line. Expressions take on string or numeric values as appropriate, and are built using the operators +, −, *, /, %, and concatenation (indicated by a blank). The C operators ++, —, +=, −=, *=, /=, and %= are also available in expressions. Variables may be scalars, array elements (denoted *x*[*i*]) or fields. Variables are initialized to the null string. Array subscripts may be any string, not necessarily numeric; this allows for a form of associative memory. String constants are quoted (").

The `print` statement prints its arguments on the standard output (or on a file if *>expr* is present), separated by the current output field separator, and terminated by the output record separator. The `printf` statement formats its expression list according to the format specified (see `printf`(3S)).

The built-in function `length` returns the length of its argument taken as a string, or of the whole line if no argument. There are also built-in functions `exp`, `log`, `sqrt`, and `int`. The last truncates its argument to an integer; `substr` (*s ,m ,n*) returns the *n*-character substring of *s* that begins at position *m*. The function `sprintf` (*fmt ,expr ,expr ,...*) formats the expressions according to the `printf`(3S) format given by *fmt* and returns the resulting string.

Patterns are arbitrary Boolean combinations (`!`, `||`, `&&`, and parentheses) of regular expressions and relational expressions. Regular expressions must be surrounded by slashes and are as in `egrep` (see `grep`(1)). Isolated regular expressions in a pattern apply to the entire line. Regular expressions may also occur in relational expressions. A pattern may consist of two patterns separated by a comma; in this case, the action is performed for all lines between an occurrence of the first pattern and the next occurrence of the second.

A relational expression is one of the following:

expression matchop regular-expression
expression relop expression

where a *relop* is any of the six relational operators in C, and a *matchop* is either ~ (for *contains*) or !~ (for *does not contain*). A *conditional* is an arithmetic expression, a relational expression, or a Boolean combination of these.

The special patterns BEGIN and END may be used to capture control before the first input line is read and after the last. BEGIN must be the first pattern, END the last.

A single character *c* may be used to separate the fields by starting the program with:

```
BEGIN { FS = c }
```

or by using the –F*c* flag option.

Other variable names with special meanings include NF, the number of fields in the current record; NR, the ordinal number of the current record; FILENAME, the name of the current input file; OFS, the output field separator (default blank); ORS, the output record separator (default newline); and OFMT, the output format for numbers (default %.6g).

EXAMPLES

The command

```
awk "length > 72" filea
```

prints lines longer than 72 characters on the standard output.

```
awk '{ print $2, $1 }' filea
```

prints the first two fields of each line in opposite order.

```
awk '{ s += $1 }
 END {print "sum is", s,
          "average is", s/NR }' filea
```

adds up the first column and prints the sum and average.

```
awk '{ for (i = NF; i > 0; --i)
           print $i }' filea
```

prints all the fields of each line in reverse order. The output prints one field per line.

```
awk "/start/, /stop/" filea
```

prints all lines between start/stop pattern pairs, for every such pair in the file.

FILES

/usr/bin/awk

SEE ALSO

grep(1), lex(1), sed(1).
"awk Reference" in *A/UX Programming Languages and Tools, Volume 2.*

BUGS

Input white space is not preserved on output if fields are involved.

There are no explicit conversions between numbers and strings. To force an expression to be treated as a number add 0 to it; to force it to be treated as a string, concatenate the null string ("") to it.

NAME

banner — generate a poster

SYNOPSIS

banner *string* ...

DESCRIPTION

banner generates its arguments (each up to 10 characters long) in large letters on the standard output.

EXAMPLES

```
banner asa
```

will cause the characters "a", "s" and "a" to be displayed as large letters on the screen.

FILES

/usr/bin/banner

SEE ALSO

banner7(1), echo(1).

NAME

banner7 — generate a large banner

SYNOPSIS

banner7 [-w[*n*]] [*message...*]

DESCRIPTION

banner7 generates a large, high-quality banner on the standard output. If *message* is omitted, banner7 prompts for and reads one line of its standard input. If -w is given, the output is scrunched down from a width of 132 columns to *n* columns, suitable for a narrow terminal. If *n* is omitted, the output defaults to 80 columns.

The output should be printed on a hard-copy device, up to 132 columns wide, with no breaks between the pages. The volume is enough that you want a printer or a fast hard-copy terminal.

BUGS

Several ASCII characters are not defined, notably <, >, [,], \, ^, _, {, }, |, and ~. Also, the characters ", ´, and & are funny looking (but in a useful way.)

The -w flag option is implemented by skipping some rows and columns. The smaller it gets, the grainier the output. Sometimes letters run together.

FILES

/usr/bin/banner7

SEE ALSO

banner(1), echo(1).

NAME

basename, dirname — isolate substrings within a pathname argument

SYNOPSIS

basename *string* [*suffix*]
dirname *string*

DESCRIPTION

basename deletes any prefix ending in / and the *suffix* (if present in *string*) from *string*, and prints the result on the standard output. It is normally used inside substitution marks (` `) within shell procedures.

dirname delivers all but the last level of the pathname in *string*.

EXAMPLES

Invoked with the argument /usr/src/cmd/cat.c,

```
cc $1
mv a.out `basename $1 '.c'`
```

compiles the named file and moves the output to a file named cat in the current directory.

```
NAME=`dirname /usr/src/cmd/cat.c`
```

sets the Bourne shell variable NAME to /usr/src/cmd.

FILES

/bin/basename
/bin/dirname

SEE ALSO

sh(1).

BUGS

The basename of / is null and is considered an error.

See at(1)

NAME

bc — arbitrary-precision arithmetic language

SYNOPSIS

bc [-c] [-l] [*file*...]

DESCRIPTION

bc is an interactive processor for a language that resembles C but provides unlimited precision arithmetic. It takes input from any files given, then reads the standard input. The -l argument stands for the name of an arbitrary precision math library. The syntax for bc programs is as follows: *L* means letter a–z; E means expression; S means statement.

Comments

are enclosed in /* and */.

Names

simple variables: *L*
array elements: *L*[*E*]
The words ibase, obase, and scale

Other operands

arbitrarily long numbers with optional sign and decimal point.
(*E*)
sqrt(E)
length (*E*) number of significant decimal digits
scale (*E*) number of digits right of decimal point
L (*E* , ... , *E*)

Operators

+ − * / % ^ (% is remainder; ^ is power)
++ -- (prefix and postfix; apply to names)
== <= >= != < >
= =+ =- =* =/ =% =^

Statements

E
{ *S* ; ... ; *S* }
if (*E*) *S*
while (*E*) *S*
for (*E* ; E ; E) *S*
null statement
break

quit

Function definitions

define *L* (*L* ,..., *L*) {
auto *L*, ... , *L*
S; ... *S*
return (*E*)
}

Functions in `-l` math library

`s`(*x*)	sine
`c`(*x*)	cosine
`e`(*x*)	exponential
`l`(*x*)	log
`a`(*x*)	arctangent
`j`(*n*,*x*)	Bessel function

All function arguments are passed by value.

The value of a statement that is an expression is printed unless the main operator is an assignment. Either semicolons or newlines may separate statements. Assignment to `scale` influences the number of digits to be retained on arithmetic operations in the manner of `dc`(1). Assignments to `ibase` or `obase` set the input and output number radix respectively.

The same letter may be used as an array, a function, and a simple variable simultaneously. All variables are global to the program. `auto` variables are pushed down during function calls. When using arrays as function arguments or defining them as automatic variables empty square brackets must follow the array name.

`bc` is actually a preprocessor for `dc`(1), which it invokes automatically, unless the `-c` (compile only) flag option is present. In this case the `dc` input is sent to the standard output instead.

EXAMPLES

```
scale = 20
define e(x){
        auto a, b, c, i, s
        a = 1
        b = 1
        s = 1
        for(i=1; 1==1; i++){
                a = a*x
                b = b*i
```

```
                c = a/b
                if(c == 0) return(s)
                s = s+c
        }
}
```

defines a function to compute an approximate value of the exponential function and

```
for(i=1; i<=10; i++) e(i)
```

prints approximate values of the exponential function of the first ten integers.

FILES

`/usr/lib/bc`	
`/usr/lib/lib.b`	mathematical library
`/usr/bin/dc`	desk calculator proper

SEE ALSO

`dc`(1).
"`bc` Reference" in *A/UX Programming Languages and Tools, Volume 2*.

BUGS

No `&&` or `||` yet.

`for` statement must have all three E's.

`quit` is interpreted when read, not when executed.

NAME

bdiff — diff large files

SYNOPSIS

bdiff *file1 file2* [*n*] [–s]

DESCRIPTION

bdiff allows processing of file which are too large for diff(1), and is used in an analogous manner to find which lines must be changed in two files to bring them into agreement. bdiff ignores lines common to the beginning of both files, splits the remainder of each file into *n*-line segments, and invokes diff upon corresponding segments. The value of *n* is 3500 by default. If the optional third argument is given, and it is numeric, it is used as the value for *n*. This is useful in those cases in which 3500-line segments are too large for diff, causing it to fail. If *file1* (*file2*) is –, the standard input is read. The optional –s (silent) argument specifies that no diagnostics are to be printed by bdiff (note, however, that this does not suppress possible exclamations by diff). If both optional arguments are specified, they must appear in the order indicated above.

The output of bdiff is exactly that of diff, with line numbers adjusted to account for the segmenting of the files (that is, to make it look as if the files had been processed whole). Note that because of the segmenting of the files, bdiff does not necessarily find a smallest sufficient set of file differences.

EXAMPLES

```
bdiff file1 file2
```

where file1 and file2 are two versions of the manual text for the cp command, produces:

```
22c22
< .IR sh (1)
---
> .IR sh (1)
35c35
< .IR chmod (2)
---
> .IR chmod (2)
50a51,56
> .SH EXAMPLES
> .IP
> cp alpha beta gamma /users/john
```

```
> .PP
> places copies of the 3 files in directory
> .BR /users/john
```

FILES

`/usr/bin/bdiff`

SEE ALSO

`diff`(1), `diff3`(1), `sdiff`(1).

NAME

bfs — big file scanner

SYNOPSIS

bfs [–] *filename*

DESCRIPTION

bfs is a read-only editor that can process much larger files than standard editors. Files may be up to 1024K bytes (the maximum possible size) and 32K lines, with up to 512 characters, including newline, per line (255 for 16-bit machines). bfs is usually more efficient than ed(1) for scanning a file, since the file is not copied to a buffer. It is most useful for identifying sections of a large file where csplit(1) may be used to divide it into more manageable pieces for editing.

Normally, the size of the file being scanned is printed, as is the size of any file written with the w command. The optional – suppresses printing of sizes. Input is prompted with * if P RETURN are typed as in ed. Prompting can be turned off again by entering another P RETURN. Note that messages are given in response to errors if prompting is turned on.

All address expressions described under ed(1) are supported. In addition, regular expressions may be surrounded with two symbols besides / and ?: > indicates downward search without wrap-around, and < indicates upward search without wrap-around. There is a slight difference in mark names: only the letters a through z may be used, and all 26 marks are remembered.

The e, g, v, k, P, p, q, w, =, ! and null commands operate as described under ed(1). Commands such as ---, +++-, +++=, -12, and +4p are accepted. Note that 1,10p and 1,10 will both print the first ten lines. The f command only prints the name of the file being scanned; there is no *remembered* filename. The w command is independent of output diversion, truncation, or crunching (see the xo, xt and xc commands, below). The following additional commands are available:

xf *file*

Further commands are taken from the named *file*. When an end-of-file is reached, an interrupt signal is received or an error occurs, reading resumes with the file containing the xf. The xf commands may be nested to a depth of 10.

`xn` List the marks currently in use (marks are set by the `k` command).

`xo` [*file*]
Further output from the `p` and null commands is diverted to the named *file*, which, if necessary, is created mode 666. If *file* is missing, output is diverted to the standard output. Note that each diversion causes truncation or creation of the file.

`:` *label*
This positions a *label* in a command file. The *label* is terminated by newline, and blanks between the `:` and the start of the *label* are ignored. This command may also be used to insert comments into a command file, since labels need not be referenced.

(`.` , `.`)`xb`/*regular expression*/*label*
A jump (either upward or downward) is made to *label* if the command succeeds. It fails under any of the following conditions:

1. Either address is not between `1` and `$`.
2. The second address is less than the first.
3. The regular expression does not match at least one line in the specified range, including the first and last lines.

On success, `.` is set to the line matched and a jump is made to *label*. This command is the only one that does not issue an error message on bad addresses, so it may be used to test whether addresses are bad before other commands are executed. Note that the command:

`xb/^/` *label*

is an unconditional jump.
The `xb` command is allowed only if it is read from someplace other than a terminal. If it is read from a pipe only a downward jump is possible.

`xt` *number*
Output from the `p` and null commands is truncated to at most *number* characters. The initial number is 255.

xv[*digit*] [*spaces*] [*value*]
The variable name is the specified *digit* following the xv. xv5100 or xv5 100 both assign the value 100 to the variable 5. xv61,100p assigns the value 1,100p to the variable 6. To reference a variable, put a % in front of the variable name. For example, using the above assignments for variables 5 and 6:

```
1,%5p
1,%5
%6
```

will all print the first 100 lines.

```
g/%5/p
```

would search globally for the characters 100 and print each line containing a match. To escape the special meaning of %, a \ must precede it.

```
g/".*\%[cds]/p
```

could be used to match and list lines containing printf of characters, decimal integers, or strings.
Another feature of the xv command is that the first line of output from a A/UX system command can be stored into a variable. The only requirement is that the first character of *value* be an !. For example:

```
.w junk
xv5!cat junk
!rm junk
!echo "%5"
xv6!expr %6 + 1
```

would put the current line into variable 5, print it, and increment the variable 6 by one. To escape the special meaning of ! as the first character of *value*, precede it with a \.

```
xv7date
```

stores the value !date into variable 7.

xbz *label*

xbn *label*
These two commands will test the last saved *return code* from the execution of a A/UX system command

(!*command*) or nonzero value, respectively, to the specified label. The two examples below both search for the next five lines containing the string `size`:

```
xv55
: l
/size/
xv5!expr %5 - 1
!if 0%5 != 0 exit 2
xbn l
xv45
: l
/size/
xv4!expr %4 - 1
!if 0%4 = 0 exit 2
xbz l
```

`xc` [*switch*]
If *switch* is `1`, output from the `p` and null commands is crunched; if *switch* is `0`, it is not. Without an argument, `xc` reverses *switch*. Initially *switch* is set for no crunching. Crunched output has strings of tabs and blanks reduced to one blank and blank lines suppressed.

EXAMPLES

```
bfs text
```

will invoke `bfs` with the file named `text`.

FILES

`/bin/bfs`

SEE ALSO

`csplit`(1), `ed`(1), `regcmp`(3X).

DIAGNOSTICS

`?` for errors in commands, if prompting is turned off. Self-explanatory error messages when prompting is on.

NAME

biff — be notified if mail arrives and who it is from

SYNOPSIS

biff [*choice*]

DESCRIPTION

biff informs the system whether you want to be notified when mail arrives during the current terminal session. *choice* can be either y or n (yes or no) so the command

```
biff y
```

enables notification; the command:

```
biff n
```

disables it. With no argument, biff prints the current notification states. When mail notification is enabled, the header and first few lines of the message will be printed on your screen whenever mail arrives. A biff y command is often included in the file .login or .profile to be executed at each login.

biff operates asynchronously. For synchronous notification use the MAIL variable of sh(1), csh(1), or ksh(1).

FILES

/usr/ucb/biff

SEE ALSO

csh(1), ksh(1) sh(1), mail(1).

NAME

bs — a compiler/interpreter for modest-sized programs

SYNOPSIS

bs [*file* [*args*]]

DESCRIPTION

bs is a remote descendant of BASIC and SNOBOL4, with a little C language thrown in. bs is designed for programming tasks where program development time is as important as the resulting speed of execution. Formalities of data declaration and file/process manipulation are minimized. Line-at-a-time debugging, the trace and dump statements, and useful run time error messages all simplify program testing. Furthermore, incomplete programs can be debugged; *inner* functions can be tested before *outer* functions have been written and vice versa.

If the command line *file* argument is provided, the file is used for input before the console is read. By default, statements read from the file argument are compiled for later execution. Likewise, statements entered from the console are normally executed immediately (see compile and execute below). Unless the final operation is an assignment, the result of an immediate expression statement is printed.

bs programs are made up of input lines. If the last character on a line is a \, the line is continued. bs accepts lines of the following form:

statement
label statement

A *label* is a *name* (see Expression Syntax later in this section) followed by a colon. A label and a variable may have the same name.

A bs statement is either an expression or a keyword followed by zero or more expressions. Some keywords (clear, compile, !, execute, include, ibase, obase, and run) are always executed as they are compiled.

Statement Syntax

expression

The *expression* is executed for its side effects (value, assignment or function call). See Expression Syntax for details about expressions. Descriptions of statement types are as follows:

`break`
`break` exits from the innermost `for/while` loop.

`clear`
Clears the symbol table and compiled statements. `clear` is executed immediately.

`compile` [*expression*]
Succeeding statements are compiled (overriding the immediate execution default). The optional expression is evaluated and used as a filename for further input. A `clear` is associated with this latter case. `compile` is executed immediately.

`continue`
`continue` transfers to the loop-continuation of the current `for/while` loop.

`dump` [*name*]
The name and current value of every nonlocal variable is printed. Optionally, only the named variable is reported. After an error or interrupt, the number of the last statement and (possibly) the user-function trace are displayed.

`exit` [*expression*]
Return to system level. The *expression* is returned as process status.

`execute`
Change to immediate execution mode (an interrupt has a similar effect). This statement does not cause stored statements to execute (see `run` later in this section).

`for` *name* = *expression expression statement*
`for` *name* = *expression expression*
...
`next`

`for` *expression , expression , expression statement*
`for` *expression , expression , expression*
...
`next`
The `for` statement repetitively executes a statement (first form) or a group of statements (second form) under control of a named variable. The variable takes on the value of the first expression, then is incremented by one on each loop, not to exceed the value of the second expression. The third and fourth forms require three expressions separated by commas.

The first of these is the initialization, the second is the test (true to continue), and the third is the loop-continuation action (normally an increment).

`fun f(` [*a*, ...]) [*v*, ...]
...
`nuf`

`fun` defines the function name, arguments, and local variables for a user-written function. Up to ten arguments and local variables are allowed. Such names cannot be arrays, nor can they be I/O associated. Function definitions may not be nested.

`freturn`

A way to signal the failure of a user-written function. See the interrogation operator (`?`), later in this section. If interrogation is not present, `freturn` merely returns zero. When interrogation *is* active, `freturn` transfers to that expression (possibly by-passing intermediate function returns).

`goto` *name*

Control is passed to the internally-stored statement with the matching label.

`ibase` *N*

`ibase` sets the input base (`radix`) to *N*. The only supported values for *N* are `8`, `10` (the default), and `16`. Hexadecimal values 10–15 are entered as `a`–`f`. A leading digit is required (that is, `f0a` must be entered as `0f0a`). `ibase` (and `obase` later in this section) are executed immediately.

`if` *expression statement*
`if` *expression*
...
[`else`
...]
`fi`

The statement (first form) or group of statements (second form) is executed if the expression evaluates to nonzero. The strings `0` and "" (null) evaluate as zero. In the second form, an optional `else` allows for a group of statements to be executed when the first group is not. The only statement permitted on the same line with an `else` is an `if`; only other `fi`'s can be on the same line with a `fi`. The elision of `else` and `if` into an `elif` is supported. Only a single `fi` is required to close an

if...elif...[else...] sequence.

include *expression*
The *expression* must evaluate to a filename. The file must contain bs source statements. Such statements become part of the program being compiled. include statements may not be nested.

obase *N*
obase sets the output base to *N* (see ibase earlier in this section).

onintr *label*
onintr
The onintr command provides program control of interrupts. In the first form, control will pass to the label given, just as if a goto had been executed at the time onintr was executed. The effect of the statement is cleared after each interrupt. In the second form, an interrupt will cause bs to terminate.

return [*expression*]
The expression is evaluated and the result is passed back as the value of a function call. If no expression is given, zero is returned.

run
The random number generator is reset. Control is passed to the first internal statement. If the run statement is contained in a file, it should be the last statement.

stop
Execution of internal statements is stopped. bs reverts to immediate mode.

trace [*expression*]
The trace statement controls function tracing. If the expression is null (or evaluates to zero), tracing is turned off. Otherwise, a record of user-function calls/returns will be printed. Each return decrements the trace expression value.

while *expression statement*
while *expression*
...
next
while is similar to for except that only the conditional expression for loop-continuation is given.

! *shell command*
An immediate escape to the Shell.

...
This statement is ignored. It is used to interject commentary in a program.

Expression Syntax

name
A *name* is used to specify a variable. Names are composed of a letter (upper or lowercase), optionally followed by letters and digits. Only the first six characters of a name are significant. Except for names declared in `fun` statements, all names are global to the program. Names can take on numeric (double float) values, string values, or can be associated with input/output (see the built-in function `open()` later in this section).

name ([*expression*[, *expression*]...])
Functions may be called by a name followed by the arguments in parentheses separated by commas. Except for built-in functions (listed later in this section), the name must be defined with a `fun` statement. Arguments to functions are passed by value.

name [*expression*[, *expression*]...]
This syntax is used to reference either arrays or tables (see built-in `table` functions later in this section). For arrays, each expression is truncated to an integer and used as a specifier for the name. The resulting array reference is syntactically identical to a name; `a[1,2]` is the same as `a[1][2]`. The truncated expressions are restricted to values between 0 and 32767.

number
A *number* is used to represent a constant value. A number is written in Fortran style, and contains digits, an optional decimal point, and possibly a scale factor consisting of an `e` followed by a possibly-signed exponent.

string
Character strings are delimited by " *characters*. The \ escape character allows the double quote (`\"`), newline (`\n`), carriage return (`\r`), backspace (`\b`), and tab (`\t`) characters to appear in a string. Otherwise, \ stands for itself.

(*expression*)
Parentheses are used to alter the normal order of evaluation.

(*expression*, *expression* [, *expression...*]) [*expression*]
The bracketed expression is used as a subscript to select a comma-separated expression from the parenthesized list. List elements are numbered from the left, starting at zero. The expression:

```
( False, True )[ a == b ]
```

has the value `true` if the comparison is true.

? *expression*
The interrogation operator tests for the success of the expression, rather than its value. At the moment, it is useful for testing for end-of-file (see examples in Programming Tips, below), the result of the `eval` built-in function, and for checking the return from user-written functions (see `freturn`). An interrogation *trap* (end-of-file, and so forth) causes an immediate transfer to the most recent interrogation, possibly skipping assignment statements or intervening function levels.

– *expression*
The result is the negation of the expression.

++*name*
Increments the value of the variable (or array reference). The result is the new value.

––*name*
Decrements the value of the variable. The result is the new value.

! *expression*
The logical negation of the expression. Watch out for the shell escape command.

expression operator expression
Common functions of two arguments are abbreviated by the two arguments separated by an operator denoting the function. Except for the assignment, concatenation, and relational operators, both operands are converted to numeric form before the function is applied.

Binary Operators

The binary operators are listeds in order of increasing precedence, as follows:

`=`

THe equal sign (=) is the assignment operator. The left operand must be a name or an array element. The result is the right operand. Assignment binds right to left, all other operators bind left to right.

`_`

Th underscore (_) is the concatenation operator.

`& |`

The logical AND character `&` has result zero if either of its arguments are zero. It has result one if both of its arguments are nonzero; the logical OR character `|` has result zero if both of its arguments are zero. It has result one if either of its arguments is nonzero. Both operators treat a null string as a zero.

`< <= > >= == !=`

The relational operators (< less than, <= less than or equal, > greater than, >= greater than or equal, == equal to, != not equal to) return one if their arguments are in the specified relation. They return zero otherwise. Relational operators at the same level extend as follows: *a>b>c* is the same as *a>b & b>c*. A string comparison is made if both operands are strings.

`+ -`

Add and subtract.

`* / %`

Multiply, divide, and remainder.

`^`

Exponentiation.

BUILT-IN FUNCTIONS

Dealing with Arguments

`arg(`*i*`)`

The value of the *i*th actual parameter on the current level of function call. At level zero, `arg` returns the *i*th command-line argument (`arg(0)` returns `bs`).

`narg()`

Returns the number of arguments passed. At level zero, the command argument count is returned.

Mathematical

`abs (`*x*`)`
The absolute value of *x*.

`atan (`*x*`)`
The arctangent of *x*. Its value is between $-\pi/2$ and $\pi/2$.

`ceil (`*x*`)`
Returns the smallest integer not less than *x*.

`cos (`*x*`)`
The cosine of *x* (radians).

`exp (`*x*`)`
The exponential function of *x*.

`floor (`*x*`)`
Returns the largest integer not greater than *x*.

`log (`*x*`)`
The natural logarithm of *x*.

`rand()`
A uniformly distributed random number between zero and one.

`sin (`*x*`)`
The sine of *x* (radians).

`sqrt (`*x*`)`
The square root of *x*.

String operations

`size (`*s*`)`
The size (length in bytes) of *s* is returned.

`format (`*F, a*`)`
Returns the formatted value of *a*. *F* is assumed to be a format specification in the style of `printf`(3S). Only the `%...f`, `%. ..e`, and `%...s` types are safe.

`index (`*x, y*`)`
Returns the number of the first position in *x* that any of the characters from *y* matches. No match yields zero.

`trans (`*s, f, t*`)`
Translates characters of the source *s* from matching characters in *f* to a character in the same position in *t*. Source characters that do not appear in *f* are copied to the result. If the string *f* is longer than *t*, source characters that match in the excess por-

tion of *f* do not appear in the result.

`substr(`*s, start, width*`)`
returns the sub-string of *s* defined by the *start* position and *width.*

`match(`*string, pattern*`)`
`mstring(`*n*`)`
The *pattern* is similar to the regular expression syntax of the ed(1) command. The characters `.`, `[`, `]`, `*`, and `$` are special. The `mstring` function returns the *n*th (1 <= *n* <= 10) sub-string of the subject that occurred between pairs of the pattern symbols `\(` and `\)` for the most recent call to `match`. To succeed, patterns must match the beginning of the string (as if all patterns began with `*`). The function returns the number of characters matched. For example

```
match("a123ab123", ".*\([a-z]\)") == 6
mstring(1) == "b"
```

File Handling

`open(`*name, file, function*`)`
`close(`*name*`)`
The *name* argument must be a `bs` variable name (passed as a string). For the `open`, the *file* argument may be either (1) a `0` (zero), `1`, or `2`, representing standard input, output, or error output, respectively, (2) a string representing a filename, or (3) a string beginning with an `!` representing a command to be executed (via `sh -c`). The *function* argument must be either `r` (read), `w` (write), `W` (write without newline), or `a` (append). After a `close`, the *name* reverts to being an ordinary variable. The initial associations are

```
open("get", 0, "r")
open("put", 1, "w")
open("puterr", 2, "w")
```

Examples are given in the following section.

`access(`*s, m*`)`
Executes `access`(2).

`ftype(`*s*`)`
Returns a single character file type indication: `f` for regular file, `p` for FIFO (that is, named pipe), `d` for directory, `b` for block special, or `c` for character special.

Tables

`table(`*name*`, `*size*`)`

A *table* in `bs` is an associatively accessed, single-dimension array. Subscripts (called *keys*) are strings (numbers are converted). The *name* argument must be a `bs` variable name (passed as a string). The *size* argument sets the minimum number of elements to be allocated. `bs` prints an error message and stops on table overflow.

`item(`*name*`, `*i*`)`

`key()`

The *item* function accesses table elements sequentially (in normal use, there is no orderly progression of key values). Where the *item* function accesses values, the *key* function accesses the subscript of the previous *item* call. The *name* argument should not be quoted. Since exact table sizes are not defined, the interrogation operator should be used to detect end-of-table, for example:

```
table("t", 100)
 ...
# If word contains the string "party",
# the following expression adds one
# to the count of that word:
++t[word]
 ...
# To print out the the key/value pairs:
for i = 0, ?(s = item (t , i)), ++i
if key() put = key()_:_s
```

`iskey(`*name*`, `*word*`)`

The `iskey` function tests whether the key *word* exists in the table *name* and returns one for true, zero for false.

Odds and Ends

`eval(`*s*`)`

The string argument is evaluated as a `bs` expression. The function is handy for converting numeric strings to numeric internal form. `eval` may also be used as a crude form of indirection, as in

```
name = "xyz"
eval("++"_ name)
```

which increments the variable `xyz`. In addition, `eval` preceded by the interrogation operator permits the user to control

`bs` error conditions. For example,

```
?eval("open(\"X\", \"XXX\", \"r\")")
```

returns the value zero if there is no file named `XXX` (instead of halting the user's program). The following executes a `goto` to the label *L* (if it exists):

```
label="L"
if !(?eval("goto "_ label)) puterr = "no label"
```

`plot` (*request, args*)

The `plot` function produces output on devices recognized by `tplot`(1G). The *requests* are as follows:

Call Function

`plot(0, term)`
: Causes further `plot` output to be piped into `tplot`(1G) with an argument of `-T`*term*.

`plot(1)`
: Erases the plotter.

`plot(2, string)`
: Labels the current point with *string*.

`plot(3,x1,y1,x2,y2)`
: Draws the line between (*x1*,*y1*) and (*x2*,*y2*).

`plot(4,x,y,r)`
: Draws a circle with center (*x*,*y*) and radius *r*.

`plot(5,x1,y1,x2,y2,x3,y3)`
: Draws an arc (counterclockwise) with center (*x1*,*y1*) and endpoints (*x2*,*y2*) and (*x3*,*y3*).

`plot(6)`
: Not implemented.

`plot(7,x,y)`
: Makes the current point (*x*,*y*).

`plot(8,x,y)`
: Draws a line from the current point to (*x*,*y*).

`plot(9,x,y)`
: Draws a point at (*x*,*y*).

`plot(10,string)`
: Sets the line mode to *string*.

`plot(11,`*x1*`,`*y1*`,`*x2*`,`*y2*`)`
Makes (*x1*,*y1*) the lower left corner of the plotting area and (*x2*,*y2*) the upper-right corner of the plotting area.

`plot(12,`*x1*`,`*y1*`,`*x2*`,`*y2*`)`
Causes subsequent *x*(*y*) coordinates to be multiplied by *x1* (*y1*) and then added to *x2* (*y2*) before they are plotted. The initial scaling is `plot(12,1.0,1.0,0.0,0.0)`

Some requests do not apply to all plotters. All requests except zero and twelve are implemented by piping characters to `tplot`(1G). See `plot`(4) for more details.

`last()`
In immediate mode, `last` returns the most recently computed value.

NOTES

Using `bs` as a calculator

```
$ bs
# Distance (inches) light travels
# in a nanosecond.
186000 * 5280 * 12 / 1e9
11.78496
...

# Compound interest
# (6% for 5 years on $1,000).
int = .06 / 4
bal = 1000
for i = 1 5*4 bal = bal + bal*int
bal - 1000
346.855007
...
exit
```

The outline of a typical `bs` program

```
# initialize things:
var1 = 1
open("read", "infile", "r")
...
# compute:
while ?(str = read)
        ...
next
# clean up:
close("read")
```

```
...
# last statement executed (exit or stop):
exit
# last input line:
run
```

Input/Output examples

```
# Copy "oldfile" to "newfile".
open("read", "oldfile", "r")
open("write", "newfile", "w")
...
while ?(write = read)
...
# close "read" and "write":
close("read")
close("write")

# Pipe between commands.
open("ls", "!ls *", "r")
open("pr", "!pr -2 -h 'List'", "w")
while ?(pr = ls) ...
...
# be sure to close (wait for) these:
close("ls")
close("pr")
```

EXAMPLES

```
bs program 1 2 3
```

compiles or executes the file named `program` as well as statements typed from standard input. The arguments `1`, `2`, and `3` are passed as arguments to the compiled/executed program.

FILES

`/bin/bs`

SEE ALSO

`ed`(1), `ksh`(1), `sh`(1), `tplot`(1G), `access`(2), `printf`(3S), `intro`(3), `plot`(4).
A/UX Programmer's Reference.

NAME

cal — generate a calendar for the specified year

SYNOPSIS

cal [[*month*] *year*]

DESCRIPTION

cal generates a calendar for the specified year. If a month is also specified, a calendar just for that month is generated. If neither is specified, a calendar for the present month is generated. The value of *year* can be between 1 and 9999. The value of *month* is a number between 1 and 12. cal makes adjustments in its output based on the specified year to reflect the calendar (Julian or Gregorian) that was in effect for the specified year.

EXAMPLES

```
cal 9 1752
```

produces a calendar for September 1752.

FILES

/usr/bin/cal

NOTES

The year is always considered to start in January.

The command cal 87 generates a calendar for 87 A.D., not 1987.

NAME

calendar — reminder service

SYNOPSIS

calendar [–]

DESCRIPTION

calendar consults the file calendar in the current directory and prints out lines that contain today's or tomorrow's date anywhere in the line. Most reasonable month-day dates such as Dec. 7, december 7, 12/7, etc., are recognized, but not 7 December. On weekends tomorrow extends through Monday.

When an argument is present, calendar does its job for every user who has a file calendar in their login directory and sends them any positive results by mail(1). Normally this is done daily by facilities in the A/UX operating system under control of cron(1M).

EXAMPLES

If the user has the following line, among other lines containing date information, in the file calendar in the login directory:

```
Monday, September 6      Labor Day Holiday
```

typing in

```
calendar
```

either on the Friday before or on the specified Monday will cause this line to be printed on the screen.

FILES

/usr/bin/calendar
./calendar
/usr/lib/calprog
/etc/passwd
/tmp/cal*
/usr/lib/crontab

SEE ALSO

at(1), leave(1), mail(1), cron(1M).

BUGS

Your calendar must be public information for you to get reminder service.
calendar's extended idea of "tomorrow" does not account for holidays.

See lp(1)

NAME

`cat` — concatenate and display the contents of named files

SYNOPSIS

`cat [-u] [-s] [-v [-t] [-e]]` *files*

DESCRIPTION

`cat` reads each *file* in sequence and writes it on the standard output.

If no input file is given, or if the argument – is encountered, `cat` reads from the standard input file. Output is buffered unless the `-u` flag option is specified. The `-s` flag option makes `cat` silent about nonexistent files.

The `-v` flag option causes nonprinting characters (with the exception of tabs, newlines and form feeds) to be displayed. Control characters are displayed as `^X` (CONTROL-*x*); the DELETE character (octal 0177) is displayed as `^?`. Non-ASCII characters (with the high bit set) are displayed as M-*x*, where *x* is the character specified by the seven low-order bits.

When used with the `-v` flag option, `-t` causes tabs to be displayed as `^I`'s and form feeds to be displayed as `^L`'s. When used with the `-v` flag option, `-e` causes a `$` character to be displayed at the end of each line (prior to the newline). The `-t` and `-e` flag options are ignored if the `-v` flag option is not specified.

EXAMPLES

```
cat file
```

displays `file`, and:

```
cat file1 file2 > file3
```

concatenates the first two files and places the result in the third.

WARNINGS

Command formats such as

```
cat file1 file2 > file1
```

will cause the original data in `file1` to be lost; therefore, take care when using shell special characters.

FILES

`/bin/cat`

SEE ALSO

cp(1), head(1), pg(1), pr(1), tail(1).

NAME

cb — C program beautifier

SYNOPSIS

cb [-s] [-j] [-l *leng*] [*file*...]

DESCRIPTION

cb reads C programs either from its arguments or from the standard input and writes them on the standard output with spacing and indentation that displays the structure of the code. Under default options, cb preserves all user newlines. Under the -s flag option, cb standardizes the code to the canonical style of Kernighan and Ritchie in *The C Programming Language*. The -j flag option causes split lines to be put back together. The -l flag option causes cb to split lines that are longer than *leng*.

EXAMPLES

If there is a C program called test.c which looks like this:

```
#define COMING 1
#define GOING 0

main ()
{
/*  This is a test of the C Beautifier  */
if (COMING)
printf ("Hello, world\n");
else
printf ("Goodbye, world\n");
}
```

Then using the cb command as shown below produces the output shown:

```
cb test.c
#define COMING 1
#define GOING 0

main ()
{
        /*  This is a test of the C Beautifier  */
        if (COMING)
                printf ("Hello, world\n");
        else
                printf ("Goodbye, world\n");
}
```

FILES

`/usr/bin/cb`

SEE ALSO

`cc`(1), `indent`(1).
The C Programming Language by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, Inc., New Jersey, 1978).

BUGS

Punctuation that is hidden in preprocessor statements will cause indentation errors.

NAME

cc — C compiler

SYNOPSIS

cc [-B *string*] [-c] [-C] [-D *symbol*[=*def*]] [-E] [-F] [-fm68881] [-g] [-I *dir*] [-L *dir*] [-l*x*] [-n] [-o *outfile*] [-O] [-p] [-P] [-R] [-s] [-S] [-t [p012al]] [-T] [-U *symbol*] [-v] [-W *c*,*arg1*[,*arg2*...]] [-X] [-Z *flags*] [-#] ... *file* ...

DESCRIPTION

cc is a front-end program that invokes the preprocessor, compiler, assembler, and link editor, as appropriate. The default is to invoke each one in turn.

Arguments whose names end with .c are taken to be C source programs. They are compiled, and each object program is left in a file whose name is that of the source, with .o substituted for .c. The .o file is normally deleted. However, if a single C program is compiled and loaded all at one time, no .o is produced, and the output is left in a file whose default name is a.out. In the same way, arguments whose names end with .s are taken to be assembly source programs and are assembled to produce a .o file.

FLAG OPTIONS

The following flag options are interpreted by cc. (Other flag options may be passed to the assembler and the linker. See ld(1) for link-editor flag options and as(1) for assembler options.)

-c Suppress the link-editing phase of the compilation and force an object file to be produced even if only one program is compiled.

-C Pass along all comments except those found on cpp(1) directive lines. The default strips out all comments.

-p Arrange for the compiler to produce code that counts the number of times each routine is called. Also, if link-editing takes place, replace the standard startoff routine by one that automatically calls monitor(3C) at the start and arranges to write out a mon.out file at normal termination of execution of the object program.

-F Do not generate inline code for the MC68881 floating-point coprocessor.

`-fm68881`
: Generate inline code for the MC68881 floating-point coprocessor. This is the default.

`-g`
: Generate additional information needed for the use of `sdb`(1).

`-l`*x*
: Same as `-l` in `ld`(1). Search the library `lib`*x*`.a`, where *x* is up to 7 characters long. A library is searched when its name is encountered, so the placement of `-l` is significant. By default, libraries are located in `LIBDIR`. If you plan to use the `-L` option, that option *must* precede `-l` on the command line.

`-L` *dir*
: Same as `-L` in `ld`(1). Change the algorithm of searching for `lib`*x*`.a` to look in *dir* before looking in `LIBDIR`. This option is effective only if it precedes the `-l` option on the command line.

`-o` *outfile*
: Same as `-o` in `ld`(1). Produce an output object file, *outfile*. The default name of the object file is `a.out`.

`-O`
: Invoke an object-code optimizer. The optimizer moves, merges, and deletes code, so symbolic debugging with line numbers could be confusing when the optimizer is used. This option may not work properly on code containing `asm` directives.

`-R`
: Have assembler remove its input file when finished.

`-W`*c*`,`*arg1*`[,`*arg2*`...]`
: Hand off the argument(s) *argi* (where *i* = 1,2, ..., *n*) to pass *c*, where *c* is one of `[p012al]` indicating preprocessor, compiler first pass, compiler second pass, optimizer, assembler, or link editor, respectively. For example,

```
-W a,-m
```

  invokes the `m4` macro preprocessor on the input to the assembler. (The `-m` flag option to `as` causes it to go through `m4`.) This must be done for a source file that contains assembler escapes.

`-s`
: Same as `-s` in `ld`(1). Strip line-number entries and symbol-table information from the output of the object file.

`-S` Compile the named C programs and leave the assembly-language output on corresponding files suffixed `.s`.

`-t [p012al]`
Find only the designated preprocessor passes whose names are constructed with the *string* argument of the `-B` flag option, that is, (`p`), compiler (`0` and `1`), optimizer (`2`), assembler (`a`), and link editor (`l`). In the absence of a `-B` option and its argument, *string* is taken to be `/lib/n`. Using `-t` with no argument is equivalent to `-tp012`.

`-T` Truncate symbol names to 8 significant characters. Many modern C compilers, as well as the proposed ANSI standard for C, allow arbitrary-length variable names. `cc` follows this convention. The `-T` option is provided for compatibility with earlier systems.

`-E` Run only `cpp`(1) on the named C programs and send the result to the standard output.

`-P` Run only `cpp`(1) on the named C programs and leave the result on corresponding files suffixed `.i`.

`-D`*symbol*[=*def*]
Define the external *symbol* to the preprocessor and give it the value *def* (if specified). If no *def* is given, *symbol* is defined as 1. This mechanism is useful with the conditional statements in the preprocessor by allowing symbols to be defined as external to the source file.

`-U`*symbol*
Undefine *symbol* to the preprocessor.

`-I`*dir* Search for `#include` files (whose names do not begin with `/`) in *dir* before looking in the directories on the standard list. Thus, `#include` files whose names are enclosed in `` `` '' `` (double quotes) are initially searched for in the directory of the `.c` file currently being compiled, then in directories named in `-I` flag options, and finally in directories on a standard list. For `#include` files whose names are enclosed in <>, the directory of the *file* argument is not searched.

`-B`*string* Construct pathnames for substitute preprocessor, compiler, assembler, and link-editor passes by concatenating

string with the suffixes `cpp`, `comp`, `optim`, `as`, and `ld`. If *string* is empty, it is taken to be `/lib/o`. For example, versions of the C compiler, assembler, and link editor can be found in the directory `/usr/lib/big`. These tools operate just like their standard counterparts, except that their symbol tables are very large. If you receive an overflow error message when you compile your program with the standard versions, you may wish to switch to the alternate versions using

`cc -B /usr/lib/big -o` *filename filename*`.c`

You should have 4 MB or more of main memory in order to use the big versions of these programs safely.

`-v` Print the command line for each subprocess executed.

`-X` Ignored by A/UX® for Motorola 68020 host processor.

`-n` Arrange for the loader to produce an executable which is linked in such a manner that the text can be made read-only and shared (nonvirtual) or paged (virtual).

`-#` Without actually starting the program, echoe the names and arguments of subprocesses that would have started. This is a special debug option.

`-Z`*flags* Special *flags* to override the default behavior (see NOTES in this section). Currently recognized flags are:

- `c` Suppress returning pointers in both a0 and d0.
- `n` Emit no code for stack-growth.
- `m` Use Motorola SGS-compatible stack growth code.
- `p` Use `tst.b` stack probes.
- `E` Ignore all environment variables.
- `F` Flip byte order in multicharacter character constants.
- `I` Emit inline code for the MC68881 floating-point coprocessor.
- `l` Suppress selection of a loader command file.
- `t` Do not delete temporary files.
- `S` Compile to be SVID-compatible. Link the program with a library module that calls

setcompat(2) with the COMPAT_SVID flag set. Define only the SYSV_SOURCE feature test macro.

P Compile for the POSIX environment. Link the program with a library module that calls setcompat(2) with the COMPAT_POSIX flag set. Define only the POSIX_SOURCE feature test macro.

B Compile to be BSD-compatible. Link the program with a library module that calls setcompat(2) with the COMPAT_BSD flag set. Define only the BSD_SOURCE feature test macro.

Other arguments are taken to be either link-editor flag-option arguments or C-compatible object programs, typically produced by an earlier cc run or perhaps by libraries of C-compatible routines. These programs, together with the results of any compilations specified, are link edited (in the order given) to produce an executable program with the name a.out unless the -o flag option of the link editor is used.

FILES

/usr/bin/cc	
file.c	input file
file.o	object file
file.s	assembly language file
a.out	link-edited output
/usr/tmp/mc68?	temporary
/lib/cpp	preprocessor
/lib/comp	compiler
/lib/optim	optimizer
/bin/as	assembler, as(1)
/bin/ld	link editor, ld(1)
/lib/libc.a	standard library, see (3)
/lib/libposix.a	POSIX library, see (2P) and (3P)
/lib/libbsd.a	BSD library
/lib/libsvid.a	SVID library
/usr/lib/shared.ld	loader command file for shared text or paged programs
/usr/lib/unshared.ld	loader command file for unshared text programs

`/lib/crt0.o`	run-time startoff
`/lib/mcrt0.o`	run-time startoff for profiling

SEE ALSO

`as`(1), `dis`(1), `ld`(1), `setcompat`(2).
The C Programming Language by B. W. Kernighan and D. M. Ritchie, (New Jersey, Prentice-Hall: 1978); "`cc` Command Syntax" in *A/UX Programming Languages and Tools, Volume 1*; "A/UX POSIX Environment," in *A/UX Programming Languages and Tools, Volume 1.*

DIAGNOSTICS

The diagnostics produced by the C compiler are sometimes cryptic. Occasional messages may be produced by the assembler or link editor.

WARNINGS

By default, the return value from a C program is completely random. The only two guaranteed ways to return a specific value are to call `exit` explicitly (see `exit`(2)) or to leave the function `main()` with a `return` (*expression*) statement.

NOTES

This version of `cc` is based on the `cc` released with the Motorola SGS and has been changed in the following ways:

- The `-Z` flag option has been added to explicitly control generation of stack-growth code for cross-development environments or generation of stand-alone code. The Motorola SGS looks for an environment variable called `M68000` and generates stack-growth code if the variable is set to `STACKCHECK`. This `cc` defaults to stack probes on 68000 host processors and no stack-growth code on the Macintosh II® 68020 processors.
- The default is to produce shared text programs. To produce nonshared text programs, you must run `ld`.
- When `cc` is used with the `-g` flag option, the arguments `-u _dbargs -lg` are inserted in the command line for the link phase. This causes the contents of `libg.a` to be linked in. Note that the Motorola SGS only generates the loader argument `-lg`, which is not sufficient to cause loading of the library's contents.

- The -v (verbose) flag option has been added to print the command line for each subprocess executed. This helps to isolate problems to a specific phase of the compilation process by showing exactly what cc is doing, so that each phase can be run by hand, if necessary.
- The Motorola SGS compiler expects functions that return pointers or structures to return their values in a0 and expects other functions to return their values in d0/d1. Because of the large body of existing code that has inconsistent type declarations. This version of the compiler emits code to return pointers in both a0 and d0 by copying a0 to d0 just prior to returning. This copy operation can be suppressed with the -Zc flag option, thus generating slightly smaller code.

See compact(1)

NAME

cdc — change the delta commentary of an SCCS delta

SYNOPSIS

cdc [-m[*mrlist*]] -r *SID* [-y[*comment*]] *file* ...

DESCRIPTION

cdc changes the "delta commentary", for the SID specified by the -r keyletter, of each named SCCS file.

A "delta commentary" is defined to be the Modification Request (MR) and comment information normally specified via the delta(1) command (-m and -y keyletters).

If a directory is named, cdc behaves as though each file in the directory were 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 (see WARNINGS); each line of the standard input is taken to be the name of an SCCS file to be processed.

Arguments to cdc, which may appear in any order, consist of *keyletter* arguments, and filenames.

All the described *keyletter* arguments apply independently to each named file:

-r*SID* Used to specify the SCCS Identification (SID) string of a delta for which the delta commentary is to be changed.

-m[*mrlist*] If the SCCS file has the v flag option set (see admin(1)) then a list of MR numbers to be added and/or deleted in the delta commentary of the SID specified by the -r keyletter *may* be supplied. A null MR list has no effect.

MR entries are added to the list of MRs in the same manner as that of delta(1). In order to delete an MR, precede the MR number with the character ! (see EXAMPLES). If the MR to be deleted is currently in the list of MRs, it is removed and changed into a comment line. A list of all deleted MRs is placed in the comment section of the delta commentary and preceded by a comment line stating that they were deleted.

If –m is not used and the standard input is a terminal, the prompt MRs? is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. The MRs? prompt always precedes the comments? prompt (see –y*keyletter*).

MRs in a list are separated by blanks and/or tab characters. An unescaped newline character terminates the MR list.

Note that if the v flag option has a value (see admin(1)), it is taken to be the name of a program (or shell procedure) which validates the correctness of the MR numbers. If a nonzero exit status is returned from the MR number validation program, cdc terminates and the delta commentary remains unchanged.

–y[*comment*] Arbitrary text used to replace the *comment*(s) already existing for the delta specified by the –r keyletter. The previous comments are kept and preceded by a comment line stating that they were changed. A null *comment* has no effect.

If –y is not specified and the standard input is a terminal, the prompt comments? is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped newline character terminates the *comment* text.

The exact permissions necessary to modify the SCCS file are documented in the "SCCS Reference" in *AIUX Programming Languages and Tools, Volume 2*. Simply stated, they are either (1) if you made the delta, you may change its delta commentary; or (2) if you own the file and directory, you may modify the delta commentary.

EXAMPLES

```
cdc -r1.6 -m"bl78-12345 !bl77-54321
bl79-00001" -ytrouble s.file
```

adds bl78-12345 and bl79-00001 to the MR list, removes bl77-54321 from the MR list, and adds the comment trouble to delta 1.6 of s.file.

```
cdc -r1.6 s.file
MRs? !bl77-54321 bl78-12345 bl79-00001
comments? trouble
```

does the same thing.

WARNINGS

If SCCS filenames are supplied to the cdc command via the standard input (– on the command line), then the –m and –y keyletters must also be used.

FILES

/usr/bin/cdc

SEE ALSO

admin(1), delta(1), get(1), help(1), prs(1), sccsfile(4). "SCCS Reference" in *A/UX Programming Languages and Tools, Volume 2*.

DIAGNOSTICS

Use help(1) for explanations.

NAME

cflow — generate C flowgraph

SYNOPSIS

cflow [-d*num*] [-i_] [-ix] [-r] *file* ...

DESCRIPTION

cflow analyzes a collection of C, yacc, lex, assembler, and object files and attempts to build a graph charting the external references. Files suffixed in .y, .l, .c, and .i are yacc'd, lex'd, and C-preprocessed (bypassed for .i files) as appropriate and then run through the first pass of lint(1). (The -I, -D, and -U flag options of the C-preprocessor are also understood.) Files suffixed with .s are assembled and information is extracted (as in .o files) from the symbol table. The output is collected and turned into a graph of external references which is displayed upon the standard output.

Each line of output begins with a reference (i.e., line) number, followed by a suitable number of tabs indicating the level. Then the name of the global (normally only a function not defined as an external or beginning with an underscore; see below for the -i inclusion flag option) a colon and its definition. For information extracted from C source, the definition consists of an abstract type declaration (e.g., char *), and, delimited by angle brackets, the name of the source file and the line number where the definition was found. Definitions extracted from object files indicate the file name and location counter under which the symbol appeared (e.g., *text*). Leading underscores in C-style external names are deleted.

Once a definition of a name has been printed, subsequent references to that name contain only the reference number of the line where the definition may be found. For undefined references, only <> is printed.

When the nesting level becomes too deep, the -e flag option of pr(1) can be used to compress the tab expansion to something less than every eight spaces.

The following flag options are interpreted by cflow:

-r Reverse the caller:callee relationship producing an inverted listing showing the callers of each function. The listing is also sorted in lexicographical order by callee.

`-ix` Include external and static data symbols. The default is to include only functions in the flowgraph.

`-i_` Include names that begin with an underscore. The default is to exclude these functions (and data if `-ix` is used).

`-d`*num* The *num* decimal integer indicates the depth at which the flowgraph is cut off. By default this is a very large number. Attempts to set the cutoff depth to a nonpositive integer will be met with contempt.

EXAMPLES

Given the following in `file.c`:

```
int   i;

main()
{
      f();
      g();
      f();
}

f()
{
      i = h();
}
```

the command:

```
cflow -ix file.c
```

produces the output:

```
1     main: int(), <file.c 4>
2           f: int(), <file.c 11>
3                 h: <>
4                 i:int, <file.c 1>
5           g: <>
```

DIAGNOSTICS

Complains about bad flag options. Complains about multiple definitions and only believes the first. Other messages may come from the various programs used (e.g., the C-preprocessor).

FILES

/usr/bin/cflow	
/usr/lib/lpfx	filters line(1) output into dag input
/usr/lib/nmf	converts nm output into dag input
/usr/lib/dag	graph maker
/usr/lib/flip	reverser

SEE ALSO

as(1), cc(1), cpp(1), lex(1), lint(1), nm(1), pr(1), yacc(1).

BUGS

Files produced by lex(1) and yacc(1) cause the reordering of line number declarations which can confuse cflow. To get proper results, feed cflow the yacc or lex input.

NAME

changesize — change the fields of the SIZE resource of a file

SYNOPSIS

/mac/bin/changesize [-v] [-p*prefsize*] [-m*minsize*] [±*option*] *file*

DESCRIPTION

changesize is based on an MPW tool that prints the fields of the SIZE resource of an application and allows the user to modify any of the fields of the SIZE resource. The format of the SIZE resource contains MultiFinder flags followed by the preferred size and minimum size of the application.

FLAG OPTIONS

The following flag options are interpreted by changesize:

-v Print the values of fields in the SIZE resource and then exit without changing anything.

-p*prefsize*
Specify an amount of memory in which the application will run effectively and which MultiFinder attempts to secure upon launch of the application. This value is expressed in units of kilobytes (K).

±*option*
Set or clear the MultiFinder flag specified by *option*. +*option* sets the MultiFinder flag, -*option* clears the flag. Multiple options can be specified at the same time on the command line. The MultiFinder flags that can be modified are:

SaveScreen
For SWITCHER compatibility. Normally, this is set to 0.

SuspendResume
When set, this bit signifies that the application knows how to process suspend/resume events.

OptionSwitch
For SWITCHER compatibility. Normally, this is set to 1.

CanBackground
Receive null events while in the background, if set.

MultiFinderAware
Take responsibility for activating and deactivation

any windows in response to a suspend/resume event, if set.

`OnlyBackground`
Set this flag if your application does not have a user interface and will not run in the foreground.

`GetFrontClicks`
Set this flag is you want to receive the mouse-down and mouse-up events used to bring your application to the foreground when the user clicks in one of the windows of your application while it is suspended.

`ChildDiedEvents`
Normally, applications set this to 0. Debuggers may set this flag to 1.

`32BitCompatible`
Set this flag if your application is 32-bit clean.

EXAMPLES

To print the fields of the size resource, use

```
/mac/bin/changesize -v file
```

To set the `32BitCompatible` flag, clear the `CanBackground` flag, set the preferred memory size to 500 KB, using

```
/mac/bin/changesize +32BitCompatible
      -CanBackground -p500 file
```

checkcw(1)

See cw(1)

See eqn(1)

NAME

checkinstall — check installation of boards

SYNOPSIS

```
/etc/checkinstall ethertalk
```

DESCRIPTION

checkinstall performs a quick test to see if the named board has been installed or not. The only board type currently supported is the Apple EtherTalk board, which is indicated by the argument ethertalk.

FILES

/etc/checkinstall

SEE ALSO

etheraddr(1M).

NAME

checkmm, checkmm1 — check documents formatted with the mm macros

SYNOPSIS

checkmm *file* ...

DESCRIPTION

checkmm stands for "check memorandum macros." Use checkmm to check for syntax errors in files that have been prepared for the mm(1) or mmt(1) command. For example, checkmm checks that you have a .DE (display end macro) corresponding to every .DS (display start macro).

The output for checkmm is the number of lines checked, and a list of macros that are unfinished because of missing macros. If you do not include a file name on the command line, checkmm takes input from standard input.

FILES

/usr/bin/checkmm
/usr/bin/checkmm1

SEE ALSO

eqn(1), mm(1), mmt(1), mvt(1), neqn(1), tbl(1), mm(5).
"Other Text Processing Tools" and "mm Reference" in *A/UX Text Processing Tools.*

DIAGNOSTICS

"checkmm Cannot open *file*" if *file* is unreadable. The remaining output of the program is diagnostic of the source file.

See checkmm(1)

NAME

checknr — check nroff/troff files

SYNOPSIS

checknr [-a.*x1*.*y1*.*x2*.*y2*.....*xn*.*yn*] [-c.*x1*.*x2*.*x3*.....*xn*] [-f] [-s] [*file*...]

DESCRIPTION

checknr checks a list of nroff(1) or troff(1) input files for certain kinds of errors involving mismatched opening and closing delimiters and unknown commands. If no files are specified, checknr checks the standard input. Delimiters checked are:

(1) Font changes using \f*x*...\fP.

(2) Size changes using \s*x*...\s0.

(3) Macros that come in *open*...*close* forms, for example, the .TS and .TE macros, which must always come in pairs.

checknr operates on the ms(5) macro package only.

Additional pairs of macros may be added to the list using the -a flag option. This must be followed by groups of six characters, each group defining a pair of macros. The six characters are a period, the first macro name, another period, and the second macro name. For example, to define a pair .BS and .ES, use:

```
checknr -a.BS.ES
```

The -c flag option causes commands (macros) to be considered "defined" which would otherwise be complained about as undefined. For instance, user-defined macros are not part of the ms macro package, and thus would be considered undefined. Any macros to be defined for checknr follow the -c with no spaces. For example, to define the macros .XX and .YY, use:

```
checknr -c.XX.YY
```

The -f flag option requests checknr to ignore \f font changes.

The -s flag option requests checknr to ignore \s size changes.

checknr is intended to be used on documents that are prepared with checknr in mind, much the same as lint(1). It expects a certain document writing style for \f and \s commands, in that each \f*x* must be terminated with \fP and each \s*x* must be terminated with \s0. While it will work to go directly into the next font or to specify the original font or point size explicitly, and many existing documents actually do this, such a practice will pro-

duce complaints from `checknr`. Since it is probably better to use the `\fP` and `\s0` forms anyway, you should think of this as a contribution to your document preparation style.

FILES

`/usr/ucb/checknr`

SEE ALSO

`nroff`(1), `troff`(1), `ms`(5).
"Other Text Processing Tools" and "`ms` Reference" in *A/UX Text Processing Tools.*

DIAGNOSTICS

Complains about unmatched delimiters.
Complains about unrecognized commands.
Various complaints about the syntax of commands.

BUGS

There is no way to define a 1-character macro name using `-a`.
Does not recognize certain reasonable constructs correctly, such as conditionals.

NAME

chfn — change finger entry

SYNOPSIS

chfn [*loginname*]

DESCRIPTION

chfn is used to change information about users. This information is used by the finger program, among others. It consists of the user's "real life" name, office room number, office phone number, and home phone number. chfn prompts the user for each field. Included in the prompt is a default value, which is enclosed between brackets. The default value is accepted simply by typing RETURN. To enter a blank field, type the word none. Below is a sample run:

```
Name [Biff Studsworth II]:
Room number (Exs: 597E or 197C) []: 521E
Office Phone (Ex: 1632) []: 1863
Home Phone (Ex: 5557532) [5441546]: none
```

chfn allows phone numbers to be entered with or without hyphens. Because finger knows only about 4-digit extensions, chfn insists upon a four digit number (after the hyphens are removed) for office phone numbers.

It is a good idea to run finger after running chfn to make sure everything is the way you want it.

The optional argument *loginname* is used to change another person's finger information. This can only be done by the superuser.

FILES

/usr/ucb/chfn
/etc/passwd
/etc/ptmp

SEE ALSO

finger(1), passwd(4)

BUGS

The encoding of the office and extension information is installation-dependent.

For historical reasons, the user's name, etc are stored in the passwd file. This is a bad place to store the information.

Because two users may try to write the `passwd` file at once, a synchronization method was developed. On rare occasions, a message that the password file is "busy" will be printed. In this case, `chfn` sleeps for a while and then tries to write to the `passwd` file again.

See chown(1)

NAME

chmod — change the permissions of a file

SYNOPSIS

chmod *mode file* ...

DESCRIPTION

The permissions of the named *files* are changed according to *mode*, which may be absolute or symbolic. An absolute *mode* is an octal number constructed from the R of the following modes:

4000	set user ID on execution
2000	set group ID on execution
1000	sticky bit, see chmod(2)
0400	read by owner
0200	write by owner
0100	execute (search in directory) by owner
0070	read, write, execute (search) by group
0007	read, write, execute (search) by others

A symbolic *mode* has the form:

[*who*] *op permission* [*op permission*]

The *who* part is a combination of the letters u (for user's permissions), g (group) and o (other). The letter a stands for ugo, the default if *who* is omitted.

op can be + to add *permission* to the file's mode, – to take away *permission*, or = to assign *permission* absolutely (all other bits will be reset).

permission is any combination of the letters r (read), w (write), x (execute), s (set owner or group ID) and t (save text, or sticky); u, g, or o indicate that *permission* is to be taken from the current mode. Omitting *permission* is only useful with = to take away all permissions.

Multiple symbolic modes separated by commas may be given. Operations are performed in the order specified. The letter s is only useful with u or g and t only works with u.

Only the owner of a file (or the superuser) may change its mode. Only the superuser may set the sticky bit. In order to set the group ID, the group of the file must correspond to your current group ID.

EXAMPLES

`chmod 755` *filename*

changes the mode of `filename` to: read, write, execute (`400+200+100`) by owner; read, execute (`40+10`) for group; read, execute (`4+1`) for others. An `ls -l` of filename shows [`-rwxr-xr-x` *filename*] that the requested mode is in effect.

`chmod =` *filename*

will take away all permissions from *filename*, including yours.

`chmod o-w` *file*

denies write permission to others.

`chmod +x` *file*

makes a file executable.

FILES

`/bin/chmod`

SEE ALSO

`ls`(1), `chown`(1), `csh`(1), `ksh`(1), `sh`(1), `chmod`(2).
A/UX Essentials
A/UX Local System Administration

NAME

chown, chgrp — change the owner or group of a file

SYNOPSIS

chown *owner file* ...

chgrp *group file* ...

DESCRIPTION

chown changes the owner of the *files* to *owner*. The owner may be either a decimal user ID or a login name found in the password file.

chgrp changes the group ID of the *files* to *group*. The group may be either a decimal group ID or a group name found in the group file.

If either command is invoked by other than the superuser, and the *files* specified are either local or remoted mounted from another System V system, the set-user ID and set-group ID bits of the file mode, 04000 and 02000 respectively, will be cleared.

EXAMPLES

```
chown doc filea fileb filec
```

would make doc the owner of the three files.

FILES

/bin/chown
/bin/chgrp
/etc/group
/etc/passwd

SEE ALSO

chmod(1), chown(2), group(4), passwd(4).

NAME

chsh — change default login shell

SYNOPSIS

chsh *name* [*shell*]

DESCRIPTION

chsh is a command similar to passwd(1) except that it is used to change the login shell field of the password file rather than the password entry. If no *shell* is specified then the shell reverts to the default login shell /bin/sh. Otherwise only /bin/csh or /bin/ksh can be specified as the shell unless you are the superuser.

An example use of this command would be

```
chsh rusty /bin/csh
```

FILES

/usr/ucb/chsh

SEE ALSO

csh(1), passwd(1), passwd(4).

NAME

ci — check in RCS revisions

SYNOPSIS

ci [-r[*rev*]] [-f[*rev*]] [-k[*rev*]] [-l[*rev*]] [-u[*rev*]] [-q[*rev*]] [-m*msg*] [-n*name*] [-N*name*] [-s*state*] [-t[*txtfile*]] *files*

DESCRIPTION

ci stores new revisions into RCS files. Each filename ending in ,v is taken to be an RCS file and all others are assumed to be working files containing new revisions. ci deposits the contents of each working file into the corresponding RCS file.

Pairs of RCS files and working files may be specified in three ways (see also the example section of co(1)).

(1) Both the RCS file and the working file are given. The RCS filename is of the form *path1*/*workfile*,v and the working filename is of the form *path2*/*workfile*, where *path1* and *path2* are (possibly different or empty) paths and *workfile* is a filename.

(2) Only the RCS file is given. Then the working file is assumed to be in the current directory and its name is derived from the name of the RCS file by removing *path1*/ and the suffix ,v.

(3) Only the working file is given. Then the name of the RCS file is derived from the name of the working file by removing *path2*/ and appending the suffix ,v.

If the RCS file is omitted or specified without a path, then ci looks for the RCS file, first in the directory ./RCS and then in the current directory.

For ci to work, the caller's login must be on the access list, unless the access list is empty or the caller is the superuser or the owner of the file. To append a new revision to an existing branch, the tip revision on that branch must be locked by the caller. Otherwise, only a new branch can be created. This restriction is not enforced for the owner of the file, unless locking is set to strict (see rcs(1)). A lock held by someone else may be broken with the rcs command.

Normally, ci checks whether the revision to be deposited is different from the preceding one. If it is not different, ci either cancels the deposit (if -q is given) or asks whether to cancel (if -q is omitted). A deposit can be forced with the -f option.

For each revision deposited, ci prompts for a log message. The log message should summarize the change and must be terminated with a line containing a single . or a CONTROL-D. If several files are checked in, ci asks whether to reuse the previous log message. If the standard input is not a terminal, ci suppresses the prompt and uses the same log message for all files. See also -m.

The number of the deposited revision can be given by any of the options -r, -f, -k, -l, -u, or -q (see -r).

If the RCS file does not exist, ci creates it and deposits the contents of the working file as the initial revision (default number: 1.1). The access list is initialized to empty. Instead of the log message, ci requests descriptive text (see -t).

Options

-r[*rev*] assigns the revision number *rev* to the checked-in revision, releases the corresponding lock, and deletes the working file. This is also the default.

If *rev* is omitted, ci derives the new revision number from the caller's last lock. If the caller has locked the tip revision of a branch, the new revision is appended to that branch. The new revision number is obtained by incrementing the tip revision number. If the caller locked a non-tip revision, a new branch is started at that revision by incrementing the highest branch number at that revision. The default initial branch and level numbers are 1. If the caller holds no lock but is the owner of the file, and locking is not set to strict, then the revision is appended to the trunk.

If a revision number is indicated by *rev*, it must be higher than the latest one on the branch to which *rev* belongs, or *rev* must start a new branch.

If *rev* indicates a branch instead of a revision, the new revision is appended to that branch. The level number is obtained by incrementing the tip revision number of that branch. If *rev* indicates a non-existing branch, that branch is created with the initial revision numbered *rev.1*.

Exception: On the trunk, revisions can be appended to the end but not inserted.

`-f`[*rev*] forces a deposit; the new revision is deposited even though it is the same as the preceding one.

`-k`[*rev*] searches the working file for keyword values to determine its revision number, creation date, author, and state (see `co`(1)), and assigns these values to the deposited revision, rather than computing them locally. A revision number given by a command option overrides the number in the working file. This option is useful for software distribution. A revision that is sent to several sites should be checked in with the `-k` option at these sites to preserve its original number, date, author, and state.

`-l`[*rev*] works like `-r`, except it performs an additional `co -l` for the deposited revision. Thus, the deposited revision is immediately checked out again and locked. This is useful for saving a revision although one wants to continue editing it after the checkin.

`-u`[*rev*] works like `-l`, except that the deposited revision is not locked. This is useful if one wants to process (that is, compile) the revision immediately after checkin.

`-q`[*rev*] quiet mode; diagnostic output is not printed. A revision that is the same as the preceding one is not deposited, unless `-f` is given.

`-m`*msg* uses the string *msg* as the log message for all revisions checked in.

`-n`*name* assigns the symbolic name *name* to the number of the checked-in revision. `ci` prints an error message if *name* is already assigned to another number.

`-N`*name* same as `-n`, except that it overrides a previous assignment of *name*.

`-s`*state* sets the state of the checked-in revision to the identifier *state*. The default is *Exp*.

`-t`[*txtfile*] writes descriptive text into the RCS file (deletes the existing text). If *txtfile* is omitted, `ci` prompts the user for text supplied from the standard input, terminated with a line containing a single . or CONTROL-D. Otherwise, the descriptive text is copied from the file *txtfile*. During initialization, descriptive

text is requested even if -t is not given. The prompt is suppressed if standard input is not a terminal.

DIAGNOSTICS

For each revision, ci prints the RCS file, the working file, and the number of both the deposited and the preceding revision. The exit status always refers to the last file checked in, and is 0 if the operation was successful, 1 if otherwise.

FILE MODES

An RCS file created by ci inherits the read and execute permissions from the working file. If the RCS file already exists, ci preserves its read and execute permissions. ci always turns off all write permissions of RCS files.

FILES

The caller of the command must have read/write permission for the directories containing the RCS file and the working file, and read permission for the RCS file itself. A number of temporary files are created. A semaphore file is created in the directory containing the RCS file. ci always creates a new RCS file and unlinks the old one. This strategy makes links to RCS files useless.

DISCLAIMER

IDENTIFICATION

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

SEE ALSO

co(1), ident(1), rcs(1), rcsdiff(1), rcsintro(1), rcsmerge(1), rlog(1), sccstorcs(1M), rcsfile(4).

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

clear — clear terminal screen

SYNOPSIS

```
clear
```

DESCRIPTION

clear clears your screen if this is possible. It looks in the environment for the terminal type (TERM) and capabilities string (TERMCAP). If TERMCAP is not found in the environment, it looks in /etc/termcap to figure out how to clear the screen.

EXAMPLES

```
clear
```

clears the screen.

FILES

/bin/clear
/etc/termcap

SEE ALSO

tput(1C), termcap(4), environ(5).

NAME

cmdo — build commands interactively

SYNOPSIS

cmdo [*command-name*]

DESCRIPTION

cmdo helps you build A/UX commands using specialized Macintosh dialog boxes. The dialog boxes make it easy to select options, choose files, and access help information, as well as build compound command lines.

Commands with many options and parameters may employ one or more nested dialog boxes.

cmdo may be run in one of three ways:

1. Double-click a command's icon from the Finder.
2. Enter a complete cmdo command line:

   cmdo *command-name*

3. Type a partial command line containing only the name of the command within a CommandShell window, then select Commando from the Edit menu. The Command-key equivalent for selecting Commando from the menu is COMMAND-K.

The first two methods of invoking cmdo execute the command. The third method pastes the command line arguments onto the command line so that more commands, flag options, and arguments can be added to create the desired command line. A command line can even be created with cmdo consisting of many commands piped together, also known as a compound command line. Pressing the RETURN key executes the command.

EXAMPLES

```
cmdo ls
```

displays the ls Commando dialog box.

FILES

/etc/cmdo

SEE ALSO

CommandShell(1).

NAME

cmp — compare two files

SYNOPSIS

cmp [-l] [-s] *file1 file2*

DESCRIPTION

The two files are compared. (If *file1* is -, the standard input is used.) Under default flag options, cmp makes no comment if the files are the same; if they differ, it announces the byte and line number at which the difference occurred. If one file is an subset of the other, that fact is noted.

FLAG OPTIONS

The following flag options are interpreted by cmp:

-l Print the byte number (decimal) and the differing bytes (octal) for each difference.

-s Print nothing for differing files; return codes only.

EXAMPLES

```
cmp alpha beta
```

will report if the files are different and at what point they differ, such as:

```
alpha beta differ: char 33, line 2
```

FILES

/bin/cmp

SEE ALSO

bdiff(1), comm(1), diff(1), diff3(1), diffmk(1).

DIAGNOSTICS

Exit code 0 is returned for identical files, 1 for different files, and 2 for an inaccessible or missing argument.

NAME

co — check out RCS revisions

SYNOPSIS

co [-l[*rev*]] [-p[*rev*]] [-q[*rev*]] [-d*date*] [-r[*rev*]] [-s*state*] [-w[*login*]] [-j*joinlist*] *files*

DESCRIPTION

co retrieves revisions from RCS files. Each filename ending in ,v is taken to be an RCS file. All other files are assumed to be working files. co retrieves a revision from each RCS file and stores it in the corresponding working file.

Pairs of RCS files and working files may be specified in three ways (see the EXAMPLES section later in this entry).

(1) Both the RCS file and the working file are given. The RCS filename is of the form *path1*/*workfile*,v and the working filename is of the form *path2*/*workfile*, where *path1* and *path2* are (possibly different or empty) paths and *workfile* is a filename.

(2) Only the RCS file is given. Then the working file is created in the current directory and its name is derived from the name of the RCS file by removing *path1*/ and the suffix ,v.

(3) Only the working file is given. Then the name of the RCS file is derived from the name of the working file by removing *path2*/ and appending the suffix ,v.

If the RCS file is omitted or specified without a path, then co looks for the RCS file, first in the directory ./RCS and then in the current directory.

Revisions of an RCS file may be checked-out locked or unlocked. Locking a revision prevents overlapping updates. A revision checked out for reading or processing (for example, compiling) need not be locked. A revision checked out for editing and later check-in must normally be locked. Locking a revision currently locked by another user fails. (A lock may be broken with the rcs(1) command.) co with locking requires the caller to be on the access list of the RCS file, unless he is the owner of the file or the superuser, or the access list is empty. co without locking is not subject to access-list restrictions.

A revision is selected by number, check-in date/time, author, or state. If none of these options is specified, the latest revision on the trunk is retrieved. When the options are applied in combination, the latest revision that satisfies all of them is retrieved. The options for date/time, author, and state, retrieve a revision on the *selected branch.* The selected branch is either derived from the revision number (if given) or is the highest branch on the trunk. A revision number may be attached to one of the options `-l`, `-p`, `-q`, or `-r`.

A `co` command applied to an RCS file with no revisions creates a zero-length file. `co` always performs keyword substitution, as follows.

FLAG OPTIONS

`-l`[*rev*] Locks the checked-out revision for the caller. If omitted, the checked-out revision is not locked. See option `-r` for handling of the revision number *rev*.

`-p`[*rev*] Prints the retrieved revision on the standard output rather than storing it in the working file. This option is useful when `co` is part of a pipe.

`-q`[*rev*] Quiet mode; diagnostics are not printed.

`-d`*date* Retrieves the latest revision on the selected branch whose checkin date/time is less than or equal to *date*. The date and time may be given in free format and are converted to local time. Examples of acceptable formats for *date* are:

```
22-April-1985, 17:20-CDT
2:25 AM, Dec. 29, 1987
Tue-PDT, 1986, 4pm Jul 21
Fri Apr 16 15:52:25 EST 1988
```

The last example illustrates the format produced by `ctime`(3) and `date`(1). Most fields in the date and time may be defaulted. `co` determines the defaults in this order: year, month, day, hour, minute, and second (that is, from most to least significant). At least one of these fields must be provided. For omitted fields that are of higher significance than the highest provided field, the current values are assumed. For all other omitted fields, the lowest possible values are assumed. For example, the date `20, 10:30` defaults to 10:30:00 of the 20th of the

current month and current year. The *date* specified on the command line must be in quotation marks if it contains spaces.

`-r`[*rev*] retrieves the latest revision whose number is less than or equal to *rev*. If *rev* indicates a branch rather than a revision, the latest revision on that branch is retrieved. *rev* is composed of one or more numeric or symbolic fields separated by a period, (`.`). The numeric equivalent of a symbolic field is specified with the `-n` option of the commands `ci` and `rcs`.

`-s`*state* retrieves the latest revision on the selected branch whose state is set to *state*.

`-w`[*login*] retrieves the latest revision on the selected branch which was checked in by the user with login name *login*. If the argument *login* is omitted, the caller's login name is assumed.

`-j`*joinlist* generates a new revision which is the join of the revisions on *joinlist*. The *joinlist* is a comma-separated list of pairs of the form *rev2*:*rev3*, where *rev2* and *rev3* are (symbolic or numeric) revision numbers. For the initial pair, *rev1* denotes the revision selected by the options `-1`,...,`-w`. For all other pairs, *rev1* denotes the revision generated by the previous pair. (Thus, the output of one join becomes the input to the next.)

For each pair, `co` joins revisions *rev1* and *rev3* with respect to *rev2*. This means that all changes that transform *rev2* into *rev1* are applied to a copy of *rev3*. This is particularly useful if *rev1* and *rev3* are the ends of two branches that have *rev2* as a common ancestor. If *rev1* < *rev2* < *rev3* are on the same branch, joining generates a new revision which is like *rev3*, but with all changes that lead from *rev1* to *rev2* undone. If changes from *rev2* to *rev1* overlap with changes from *rev2* to *rev3*, `co` prints a warning and includes the overlapping sections, delimited by the lines `<<<<<<< rev1`, `=======`, and `>>>>>>> rev3`.

For the initial pair, *rev2* may be omitted. The default is the common ancestor. If any of the argu-

ments indicate branches, the latest revisions on those branches are assumed. If the option `-l` is present, the initial *rev1* is locked.

KEYWORD SUBSTITUTION

Strings of the form `$`*keyword*`$` and `$`*keyword*`:...$` embedded in the text are replaced with strings of the form `$`*keyword*`:` *value* `$`, where *keyword* and *value* are pairs as listed. Keywords may be embedded in literal strings or comments to identify a revision.

Initially, the user enters strings of the form `$`*keyword*`$`. On checkout, `co` replaces these strings with strings that are of the form `$`*keyword*`:` *value* `$`. If a revision containing strings of the latter form is checked back in, the value fields will be replaced during the next checkout. Thus, the keyword values are automatically updated on checkout.

Keywords and their corresponding values are as follows:

`$Author$`	The login name of the user who checked in the revision.
`$Date$`	The date and time the revision was checked in.
`$Header$`	A standard header containing the RCS filename, the revision number, the date, the author, and the state.
`$Locker$`	The login name of the user who locked the revision (empty if not locked).
`$Log$`	The log message supplied during checkin, preceded by a header containing the RCS filename, the revision number, the author, and the date. Existing log messages are *not* replaced. Instead, the new log message is inserted after `$Log:... $`. This is useful for accumulating a complete change log in a source file.
`$Revision$`	The revision number assigned to the revision.
`$Source$`	The full pathname of the RCS file.
`$State$`	The state assigned to the revision with `rcs -s` or `ci -s`.

DIAGNOSTICS

The RCS filename, the working filename, and the revision number retrieved are written to the diagnostic output. The exit status always refers to the last file checked out, and is 0 if the operation was successful, 1 if otherwise.

EXAMPLES

Suppose the current directory contains a subdirectory `RCS` with an RCS file `io.c,v`. Then all of the following commands retrieve the latest revision from `RCS/io.c,v` and store it into `io.c`.

```
co io.c
co RCS/io.c,v
co io.c,v
co io.c RCS/io.c,v
co io.c io.c,v
co RCS/io.c,v io.c
co io.c,v io.c
```

FILE MODES

The working file inherits the read and execute permissions from the RCS file. In addition, the owner-write permission is turned on unless the file is checked out unlocked and locking is set to `strict` (see `rcs`(1)).

If a file with the name of the working file already exists and has write permission, `co` cancels the checkout if `-q` is given, or asks whether to cancel if `-q` is not given. If the existing working file is not writable, it is deleted before the checkout.

FILES

The caller of the command must have write permission in the working directory, read permission for the RCS file, and either read permission (for reading) or read/write permission (for locking) in the directory which contains the RCS file.

A number of temporary files are created. A semaphore file is created in the directory of the RCS file to prevent simultaneous updates.

DISCLAIMER

This reference manual entry describes a utility that Apple understands to have been released into the public domain by its author or authors. Apple has included this public domain utility for your convenience. Use it at your own discretion. Often the source code can be obtained if additional requirements are met, such as the purchase of a site license from an author or institution.

IDENTIFICATION

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

SEE ALSO

ci(1), ident(1), rcs(1), rcsdiff(1), rcsintro(1), rcsmerge(1), rlog(1), rcsfile(4), sccstorcs(1M).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

LIMITATIONS

The option -d gets confused in some circumstances and accepts no date before 1970. There is no way to suppress the expansion of keywords, except by writing them differently. In nroff and troff, this is done by embedding the null-character \& into the keyword.

BUGS

The option -j does not work for files that contain lines with a single ..

NAME

col — filter text containing printer control sequences for use at a display device

SYNOPSIS

col [-b] [-f] [-p] [-x]

DESCRIPTION

col reads from the standard input and writes onto the standard output. It performs the line overlays implied by reverse linefeeds (ASCII code ESCAPE-7), and by forward and reverse half-linefeeds (ESCAPE-9 and (ESCAPE-8). col is particularly useful for filtering multicolumn output made with the .rt command of nroff and output resulting from use of the tbl(1) preprocessor.

If the -b flag option is given, col assumes that the output device in use is not capable of backspacing. In this case, if two or more characters are to appear in the same place, only the last one read will be output.

Although col accepts half-line motions in its input, it normally does not emit them on output. Instead, text that would appear between lines is moved to the next lower full-line boundary. This treatment can be suppressed by the -f (fine) flag option; in this case, the output from col may contain forward half-linefeeds (ESCAPE-9), but will still never contain either kind of reverse line motion.

Unless the -x flag option is given, col will convert white space to tabs on output wherever possible to shorten printing time.

The ASCII control characters SO (\017) and SI (\017) are assumed by col to start and end text in an alternate character set. The character set to which each input character belongs is remembered, and on output SI and SO characters are generated as appropriate to ensure that each character is printed in the correct character set.

On input, the only control characters accepted are space, backspace, tab, return, SI , SO , T (\013), and escape followed by 7, 8, or 9. The VT character is an alternate form of full reverse linefeed, included for compatibility with some earlier programs of this type. All other nonprinting characters are ignored.

Normally, col will ignore any unknown escape sequences found in its input; the -p flag option may be used to cause col to generate these sequences as regular characters, subject to overprinting

from reverse line motions. The use of this flag option is highly discouraged unless the user is fully aware of the textual position of the escape sequences.

EXAMPLES

```
nroff -mm filea | col
```

pipes multicolumn `nroff` output through the `col` filter to enable proper creation of columns.

FILES

`/usr/bin/col`

SEE ALSO

`colcrt`(1), `nroff`(1), `tbl`(1).

NOTES

The input format accepted by `col` matches the output produced by `nroff` with either the `-T37` or `-Tlp` flag options. Use `-T37` (and the `-f` flag option of `col`) if the ultimate disposition of the output of `col` will be a device that can interpret half-line motions, and `-Tlp` otherwise.

BUGS

Cannot back up more than 128 lines.
Allows at most 800 characters, including backspaces, on a line.
Local vertical motions that would result in backing up over the first line of the document are ignored. As a result, the first line must not have any superscripts.

NAME

colcrt — filter nroff output for terminal previewing

SYNOPSIS

colcrt [-] [-2] [*file*]

DESCRIPTION

colcrt provides virtual half-line and reverse line feed sequences for terminals without such capability, and on which overstriking is destructive. Half-line characters and underlining (changed to dashing "–") are placed on newlines in between the normal output lines.

The optional - suppresses all underlining. It is especially useful for previewing allboxed tables from tbl(1).

The option –2 causes all half-lines to be printed, effectively double spacing the output. Normally, a minimal space output format is used which will suppress empty lines. The program never suppresses two consecutive empty lines, however. The –2 flag option is useful for sending output to the line printer when the output contains superscripts and subscripts which would otherwise be invisible.

A typical use of colcrt would be

```
tbl exum2.n | nroff -mm | colcrt | more
```

FILES

/usr/ucb/colcrt

SEE ALSO

col(1), more(1), nroff(1), troff(1), ul(1).

BUGS

Should fold underlines onto blanks even with the - flag option so that a true underline character would show; if we did this, however, colcrt wouldn't get rid of cu'd underlining completely.

Can't back up more than 102 lines.

General overstriking is lost; as a special case "|" overstruck with "–" or underline becomes "+".

Lines are trimmed to 132 characters.

Some provision should be made for processing superscripts and subscripts in documents which are already double-spaced.

NAME

colrm — remove columns from a file

SYNOPSIS

colrm *startcol* [*endcol*]

DESCRIPTION

colrm removes selected columns from a file. Input is taken from standard input. Output is sent to standard output.

If colrm is called with one parameter, the columns of each line are removed starting with the specified column. If colrm is called with two parameters, the columns from the first column to the last column are removed.

Column numbering starts with column 1.

FILES

/usr/ucb/colrm

SEE ALSO

awk(1), cut(1), expand(1), sed(1).

NAME

comb — combine SCCS deltas

SYNOPSIS

comb [-c*list*] [-o] [-p*sid*] [-s] *file* ...

DESCRIPTION

comb generates a shell script (see sh(1)) which, when run, will reconstruct the given SCCS files by combining some series of changes. Then the reconstructed files will, hopefully, be smaller than the original files. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, comb behaves as though each file in the directory were 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 input is taken to be the name of an SCCS file to be processed; non-SCCS files and unreadable files are silently ignored.

The generated shell procedure is written on the standard output.

The keyletter arguments are as follows (each is explained as though only one named file is to be processed, but the effects of any keyletter argument apply independently to each named file):

-p*SID* The SCCS Identification string (SID) of the oldest delta to be preserved. All older deltas are discarded in the reconstructed file.

-c*list* A *list* (see get(1) for the syntax of a *list*) of deltas to be preserved. All other deltas are discarded.

-o For each get -e generated, this argument causes the reconstructed file to be accessed at the release of the delta to be created, otherwise the reconstructed file would be accessed at the most recent ancestor (see "SCCS Reference" in *AIUX Programming Languages and Tools, Volume 2*). Use of the -o keyletter may decrease the size of the reconstructed SCCS file. It may also alter the shape of the delta tree of the original file.

-s This argument causes comb to generate a shell script which, when run, will produce a report giving, for each file: the filename, size (in blocks) after combining, original size (also in blocks), and percentage change computed by:

100 * (original – combined) / original

It is recommended that before any SCCS files are actually combined, one should use this flag option to determine exactly how much space is saved by the combining process.

If no keyletter arguments are specified, `comb` will preserve only leaf deltas and the minimal number of ancestors needed to preserve the tree.

EXAMPLES

```
comb s.file1 > tmp1
```

produces a shell script saved in `tmp1` which will remove from the SCCS-format file, `s.file1`, all deltas previous to the last set of changes, i.e., removes the capability to return to earlier versions.

FILES

`/usr/bin/comb`

`s.COMB`	The name of the reconstructed SCCS file.
`comb???`	Temporary.

SEE ALSO

`admin`(1), `delta`(1), `get`(1), `help`(1), `prs`(1), `sh`(1), `sccsfile`(4).
"SCCS Reference" in *A/UX Programming Languages and Tools, Volume 2*.

DIAGNOSTICS

Use `help`(1) for explanations.

BUGS

`comb` may rearrange the shape of the tree of deltas. It may not save any space; in fact, it is possible for the reconstructed file actually to be larger than the original.

NAME

comm — select or reject lines common to two sorted files

SYNOPSIS

comm [-[123]] *file1 file2*

DESCRIPTION

comm reads *file1* and *file2* which should be ordered in ASCII collating sequence (see sort(1)), and produces a three-column output: lines only in *file1* lines only in *file2* and lines in both files. The file name - means the standard input.

Flags 1, 2, or 3 suppress printing of the corresponding column. Thus comm -12 prints only the lines common to the two files; comm -23 prints only lines in the first file but not in the second; comm -123 is a no-op.

EXAMPLES

```
comm -12 filea fileb
```

prints only the lines common to filea and fileb.

```
comm -23 filea fileb
```

prints only lines in the first file but not in the second.

```
comm -123 filea fileb
```

is not an option, as it suppresses all output.

```
comm -3 filea fileb
```

prints only the lines that differ in the two files.

FILES

/usr/bin/comm

SEE ALSO

bdiff(1), cmp(1), diff(1), diff3(1), diffmk(1), sort(1), uniq(1).

NAME

CommandShell — A/UX® Toolbox application for managing command-interpretation windows and moderating access to the A/UX console window

SYNOPSIS

CommandShell [-b *pid*] [-u] [-q]

DESCRIPTION

CommandShell provides a Macintosh user interface to A/UX users. Within CommandShell windows, you can enter A/UX command lines for processing by one of the available shells.

A/UX commands can be entered with the aid of Macintosh copy and paste operations. The source of text that can be copied to the Clipboard includes any previous command lines in the same window as well as any text available in other windows (including other Macintosh application windows). The CommandShell windows are scrollable, so you can make previous commands or resultant outputs available for copying.

A/UX commands can also be built semiautomatically by entering the command name at the start of a line and then choosing the Commando menu item from the Edit menu of CommandShell. After entering information in the dialog box, the command line is changed to include all the flag options and arguments that were generated with the help of the Commando dialog. This same function is also available through the cmdo command, which is described in cmdo(1).

When CommandShell is started normally, one window is displayed by default unless you have saved a previous window layout with more than one window. More windows can be created by choosing New from the File menu or pressing COMMAND-N. A new window appears in front of the existing window or windows. The title bar of each window is numbered in sequence according to its creation order. Normally you can create up to 15 windows.

When you create a new window, it appears in front of and to the lower right of the previous window. This position obscures the windows behind the front window. You can use the tiling commands to view the contents of all the windows. For specific tiling commands, see the section describing the Window menu.

Besides managing command-interpretation windows, CommandShell moderates access to the A/UX console window. This window is one of the places where the A/UX environment and the Macintosh desktop environment meet. Kernel error messages are routed to this window so they do not disturb the bit-mapped display of Macintosh applications.

The Macintosh user interface is an integral part of the A/UX boot process, supported in part by the A/UX console window of CommandShell. This special window is the place where all boot messages are routed and the place where you enter responses in the event that one of the boot processes requires user input. The A/UX console window is unlike any other CommandShell window because it can never be closed (although it can be hidden or obscured). During the boot process, CommandShell disables many of the functions in its menus (see the -b flag option).

All the status messages that are normally directed to the system console during startup no longer appear. Many of these messages are not useful to anyone besides the system administrator. A progress bar is presented to users in place of the startup messages. When multiuser mode is entered, the progress bar is replaced by a login dialog box.

If you wish to inspect the boot messages that resulted when the system was last booted, you can choose the A/UX System Console window from the Window menu of CommandShell. Within multiuser mode, a CommandShell process has to be started manually through a command request or automatically through the Login preferences that you maintain at the time you log in to the system (see Login(1M)).

To view the A/UX console window, perform these steps:

1. Choose CommandShell from the Apple menu.
2. Choose A/UX System Console from the Window menu.
3. Use the scroll bar to scroll back the contents of the window.

Processes that run as part of a startup script, such as /etc/sysinitrc, or as another part of the booting process may occasionally ask for user input. For example, suppose you add another Ethernet card to your system. Then suppose that while rebooting, the system needs to request address information about the new card. You will not see a prompt for this information directly. Instead an alert box will appear in front of the A/UX

boot progress bar. The alert box will say that an A/UX process requires input. Clicking OK in the alert box will cause the the A/UX console window to appear and the alert box to disappear. The A/UX console window will contain messages prompting you for input. At these junctures, you will be permitted to provide input into the window. At all other times, the window is for reading purposes only.

The alerting process that has just been described is referred to as a *notification system*. A similar notification system has been created for the handling of A/UX kernel messages. Alert boxes are displayed that encourage you to inspect the text of the A/UX error messages that has been tucked away in the normally-hidden A/UX console window.

Here is what you should do in response to this form of notification:

1. After reading the alert box, click OK.
2. Make `CommandShell` the selected application if it is not already the application in control of the active window.
3. Inspect the A/UX kernel errors by choosing A/UX System Console from the Window menu of `CommandShell`. When chosen, this menu item makes active the normally hidden A/UX console window. If it was hidden, it will now be visible. It should contain all the error messages generated since the last time the system was booted (if you scroll back, you can see the old messages). It will also let you respond to any prompts for input.

If after checking the window messages in the A/UX console window, you see that the error information regards system or network performance (such as a number of retries before successful transmission of a network packet), then no further corrective action is required. Sometimes, however, the error message may indicate a serious error condition, such as "`file system full`," "`file system corrupt`," or "`fork failed: too many processes`." In these cases your current work is subject to loss if the error condition persists.

You can change the way the system notifies you of a process needing input or a system message when `CommandShell` is not the active application. To set notification preferences, choose Notification Levels from the Preferences menu. Choosing

Notification Levels displays a dialog box in which you can select to be notified by an alert box, a flashing icon in the menu bar, or both. User preferences are normally stored in the `.cmdshellprefs` files located in the home directories of each user (see "*Managing CommandShell preferences*"). The default notification is to display an alert box because console messages may indicate a fundamental system problem that you should know about immediately.

When `CommandShell` is run in the normal way and is the active application, a Quit menu item is not available under the File menu (see the `-q` flag option for an alternative way of running `CommandShell`). Instead of a Quit function, which would leave you with no way to view or enter responses directed to the system console, `CommandShell` has the menu item Close All Windows. This menu item kills all shell processes and closes all of the windows, except the A/UX console window that accepts console messages. If you simply wish to set aside `CommandShell` windows without killing active processes, choose Hide Command Shell Windows from the Apple menu.

The combined A/UX and desktop environment supported with `CommandShell` is unique in terms of the flexibility you have for performing file manipulation chores. To delete a file, you can drag it in its iconic form on the desktop to the Trash icon and choose Empty Trash from the Special menu. Alternately, you can open a `CommandShell` window to run the A/UX command `rm`. The latter way may be the quicker way to perform a sophisticated operation. For example, consider the task of removing all files in the current folder (or directory) containing the letters "`.tmp`." The easy way is to use a command line such as

```
rm *.tmp*
```

Since A/UX is a multitasking system, other windows besides the active one receive processor attention. But when you use a command, only the active window (the window with horizontal lines in the title bar) receives the instruction. All mouse and keyboard inputs are reliably intercepted by the active window. However, if the active appliaction has no open windows, character sequences may be lost.

MENU OPTIONS

CommandShell displays menus titled File, Edit, Window, Fonts, Commands, and Preferences in the menu bar at the top of the screen, plus the Apple menu at the far left of the menu bar. To choose a menu item, position the pointer on the menu title, press and hold the mouse button, and move the pointer to the menu item while holding down the mouse button. Release the mouse button when the pointer highlights the desired item.

Many menu items can be chosen from the keyboard by holding down COMMAND (not CONTROL) and typing a character. The character required is shown beside the Command-key symbol in the menu. Such Command-key equivalents may be entered as lowercase; you don't need to hold down SHIFT as well.

The following sections describe the actions performed by the various CommandShell menu items.

The Apple Menu

At the far left of the menu bar, the Apple symbol is the title of a menu that contains some general Macintosh desk accessories and some menu items specifically related to CommandShell. The menu items related to CommandShell include:

About CommandShell
: Display a dialog box that gives version information.

Hide Command Shell Windows
: Hide CommandShell windows temporarily without killing active processes.

CommandShell
: Make CommandShell the selected application and make its associated window that was most recently active the currently active window once again (if any were opened before).

The File Menu

The menu items in the File menu allow you to do such things as create and close windows and select printing options.

New
: Create a new window. The windows are numbered sequentially according to their creation order. The Command-key equivalent for the New menu item is COMMAND-N.

Open...
: Launch a UNIX command or launch an editor if the highlighted file is a text file. The

Command-key equivalent for Open menu item is COMMAND-O.

Close — Close the active window. Before you close a window, make sure that you write the contents of the window to a disk if you want to save your work. The Command-key equivalent for the Close menu item is COMMAND-W.

Save Selection — Save the contents of a `CommandShell` window in an A/UX file. The text you want saved must be selected (the text appears highlighted).

Save Preferences — Save all window settings, layout information, and notification level settings.

Restore From Preferences — Restore window settings and layout to that specified in the preferences file. Open any saved windows that have been closed. Run initial command in windows that do not already have a command running.

Page Setup — Display a dialog box that lets you set the paper size, orientation, and reduction or enlargement for subsequent printing actions.

Print Selection — Print selected text from the active window. Use the Chooser desk accessory, available in the Apple menu, to specify which printer to use. Use the Page Setup menu item, just described, to specify paper size, orientation, and scale.

Close All Windows — Close all windows at once. All the windows disappear. Before you close the windows, make sure that you write the contents of each window to a disk if you want to save your work. If you don't write the contents to a disk, they are lost.

The Edit Menu

The menu items in the Edit menu help you do such things as move text around and perform certain global formatting actions.

Undo — Reverse the most recent text change. If you choose Undo a second time, the change is reinstated. The Command-key equivalent for the Undo menu item

is COMMAND-Z.

Cut — Copy the currently selected text in the active window to the Clipboard and then delete it from the window. This menu item is used with desk accessories only; otherwise it is disabled. The Command-key equivalent for the Cut menu item is COMMAND-X.

Copy — Copy the currently selected text in the active window to the Clipboard without deleting it from the window. The Command-key equivalent for the Copy menu item is COMMAND-C.

Paste — Inserts the contents of the Clipboard at the current text cursor location. The Command-key equivalent for the Paste menu item is COMMAND-V.

Clear — Delete the currently selected text from the active window. This menu item is used with desk accessories only; otherwise it is disabled. The Command-key equivalent for the Clear menu item is DELETE.

Select All — Select the entire document shown in the active window. The Command-key equivalent for the Select All menu item is COMMAND-A.

Commando — Build commands semiautomatically. Choose this after entering the command name at the start of a line. Afterward, a dialog box is displayed that depicts all the features of the command. Upon exiting the dialog, the command line that you started to specify at the outset is changed to include all of the flag options and arguments that were generated with the help of the Commando dialog. The Command-key equivalent for the Commando menu item is COMMAND-K.

The Window Menu

The menu items in the Window menu help you arrange and display `CommandShell` windows. The menu is divided into three parts. The upper part of the menu contains menu items that help you arrange windows in various formats. The middle part contains menu items that help you size and order the windows. The lower part contains a list of all windows currently available in

CommandShell. When you choose one of the window names in the lower list, CommandShell makes the corresponding window the active window. The names of currently available windows are listed in the order they were originally created.

The menu items in the top part of the menu do the following:

Tile — Position windows in a right-to-left, then top-to-bottom sequence. You must have more than one window on the desktop to use this menu option. The Command-key equivalent for the Tile menu item is COMMAND-T.

Tile Horizontal — Position windows from top to bottom on the screen in their creation order. The windows are stretched to fit the width of the screen. The height of each window is adjusted to accommodate the number of windows.

Tile Vertical — Position windows from left to right on the screen in their creation order. The windows are stretched to fit the height of the screen. The width is adjusted to accommodate the number of windows.

Standard Positions — Reposition the windows in the original stacked order, from front to back.

The items in the middle part of the menu do the following:

Standard Size — Resize a window to its original dimensions. The Command-key equivalent for the Standard Size menu item is COMMAND-S.

Full Height — Stretch a window to the full height of the screen. The Command-key equivalent for the Full Height menu item is COMMAND-F.

Zoom Window — Make the window larger. The window automatically is resized to fit the whole screen. You can return a window to its previous size by choosing the Zoom Window menu item again.

Hide *window-name* — Make *window-name* temporarily disappear. The window is no longer visible, but is still available. To show a window that has been hid-

den, choose the window name from the Window menu again. The window appears in front of other open windows and becomes the active window. The Command-key equivalent for the Hide Terminal item is COMMAND-H.

Show All Windows
: Show all windows that have been hidden.

Last Window
: Makes the previously active window the active window once again, making it visible if it was hidden, and making it the recipient for any keyboard or mouse inputs. Repeating this selection, returns the windows to their original states. The Command-key equivalent for the Last Window menu item is COMMAND-L.

Rotate Window
: Move the rear window to the front of all the other windows. The Command-key equivalent for the Rotate Window menu item is COMMAND-R.

The menu items in the bottom part of this menu are window names for all currently available `CommandShell` windows. Selecting a window name makes it the active window, which also makes it visible as the front window. Among any other windows listed is the ever-present A/UX console window.

A/UX System Console
: Make the A/UX console window the active window. This window is used to view console messages. The Command-key equivalent for the A/UX System Console menu item is COMMAND-0 (zero).

The Fonts Menu

The Fonts menu lets you choose the type of font and the point size of the font for text entered or displayed in the active `CommandShell` window.

The Commands Menu

The menu items in the Commands menu help you set defaults for recording information and allow you to clean up the screen.

Don't Record Lines Off Top/Record Lines Off Top
: When `CommandShell` is invoked, it is set to record a preset number of lines as they scroll

out of view. If you do not want to store the lines for possible review later, you can stop the recording of lines for a particular window. If the default setting is left unchanged, the menu item Don't Record Lines Off Top appears in the Commands menu. Choose it so the lines are not recorded. If the lines are not being recorded, the menu item Record Lines Off Top appears in the menu. Choose it to record the lines off the top of the window as they scroll out of view from a particular window.

Clear Lines Off Top
: Erase recorded lines and make them no longer available for review within a particular window. The scroll bar disappears in the active `CommandShell` window.

Redraw Screen
: Clean up the screen if textual output affects the bitmapped display.

The Preferences Menu

The menu items in the Preferences menu help you select how you want to be notified of system messages, choose your default window settings, and allow you to set your preferred window configuration.

Notification Levels...
: Set the notification level for console messages. A dialog box appears in which you choose how you are notified of console messages. The choices are an alert box, a flashing icon in the menu bar, or both.

New Window Settings...
: Specify the default title prefix, window cascade origin, window size, font name, font size, and number of lines saved off the top of the window. A dialog box appears for these specifications.

Active Window Settings...
: Specify the settings for the active window. A dialog box appears allowing you to specify window title, size, and position; whether or not to save lines off the top; and the initial command to run in the window when it opens.

Managing CommandShell preferences

Preferences are normally saved in the `.cmdshellprefs` file located in the home directory. To maintain more than one set of preferences, you can establish a different filename as the file for storing preferences. For example, to allow one set of preferences (window sizes and so forth) to be saved for use with a large display device and another set of preferences to be saved for use with a smaller display device, you can reset the `CommandShell` variable that controls the file that is used to maintain these settings. The name of this variable is `CMDSHELLPREFS`. This variable can be set to be something other than `.cmdshellprefs`. When reset, the variable is normally assigned the name of a file relative to the user's home directory, although an absolute pathname is also acceptable. The choice of an absolute pathname is particularly helpful when your home directory is served to you through a file server which you access through several different A/UX systems. In such a case, each system can contain a preferences file that is customized for its own hardware.

FLAG OPTIONS

When invoked without any flag options, `CommandShell` starts with one active window on the desktop. The window is titled "CommandShell 1." The following flag options modify this default behavior:

`-b` *pid*

Used at boot time to start `CommandShell` in a background layer without any windows. With this option, `CommandShell` accepts window management commands only as far as permitting the displaying of the A/UX console window. You do this by choosing CommandShell from the Apple menu and then choosing A/UX System Console from the Window menu. No command-interpretation windows can be displayed. The treatment of kernel errors or requests for input are treated as has been described previously. *pid* is the process ID of `macsysinit` (see `brc`(1M)). After taking control of the system console, `CommandShell` sends a signal to this PID to exit. This signal heralds the continuation of all the remaining startup processes, which can continue with the assurance that alerts boxes will be produced as necessary to notify users of the need for interaction with messages directed to the system console.

-u Specify that a user has logged in and start CommandShell in a background layer. When CommandShell is made the active application, the user's preferred (or default) CommandShell window layout is established. The default is to open a single CommandShell window. When started this way, individual user preferences can be stored using Save Preferences in the File menu.

-q Do not include a Quit menu item in the File menu and bring CommandShell into the foreground.

FILES

/mac/bin/CommandShell	object file
/mac/bin/%CommandShell	resource fork
$HOME/.cmdshellprefs	default preferences file

SEE ALSO

StartMonitor(1M), brc(1M), startmac(1M), startmsg(1M).

NAME

compact, uncompact, ccat — compress and uncompress files

SYNOPSIS

compact [*name...*]

uncompact [*name...*]

ccat [*file...*]

DESCRIPTION

compact compresses the named files using an adaptive Huffman code. If no filenames are given, the standard input is compacted to the standard output. compact operates as an on-line algorithm. Each time a byte is read, it is encoded immediately according to the current prefix code. This code is an optimal Huffman code for the set of frequencies seen so far. It is unnecessary to prefix a decoding tree to the compressed file since the encoder and the decoder start in the same state and stay synchronized. Furthermore, compact and uncompact can operate as filters. In particular, the command sequence

```
| compact | uncompact |
```

operates as a (very slow) no-op.

When an argument *file* is given, it is compacted and the resulting file is placed in *file*.C. *file* is unlinked. The first two bytes of the compacted file code the fact that the file is compacted. This code is used to prohibit recompaction.

The amount of compression to be expected depends on the type of file being compressed. Typical values of compression are: Text (38%), Pascal Source (43%), C Source (36%) and Binary (19%). These values are the percentages of file bytes reduced.

uncompact restores the original file from a file compressed by compact. If no filenames are given, the standard input is uncompacted to the standard output.

ccat cats the original file from a file compressed by compact, without uncompressing the file.

RESTRICTION

The last segment of the filename must contain fewer than thirteen characters to allow space for the appended .C.

FILES

```
/usr/ucb/compact
/usr/ucb/ccat
/usr/ucb/uncompact
*.C
```

SEE ALSO

pack(1),
Gallager, Robert G., *Variations on a Theme of Huffman*, I.E.E.E. Transactions on Information Theory, vol. IT-24, no. 6, November 1978, pp. 668–674.

NAME

compress, uncompress, zcat — compress and expand data

SYNOPSIS

compress [-f] [-v] [-c] [-V] [-b *maxbits*] [*files*]
uncompress [-f] [-v] [-c] [-V] [*files*]
zcat [-V] [*files*]

DESCRIPTION

compress reduces the size of the named files using adaptive Lempel-Ziv coding. Whenever possible, each file is replaced by one with the extension .z, while keeping the same ownership modes, access, and modification times. If no files are specified, the standard input is compressed to the standard output. Compressed files can be restored to their original form using uncompress or zcat.

FLAG OPTIONS

-f force compression of *files*. This is useful for compressing an entire directory, even if some of the files do not actually shrink. If -f is not given and compress is running in the foreground, the user is prompted as to whether an existing file should be overwritten.

-c make compress or uncompress write to the standard output; no files are changed. The nondestructive behavior of zcat is identical to that of uncompress -c.

-b *maxbits*

use *maxbits* as the maximum number of bits to use in codes when compressing file. compress uses a modified Lempel-Ziv algorithm according to which common substrings in the file are first replaced by 9-bit codes 257 and up. When code 512 is reached, the algorithm switches to 10-bit codes and continues to use more bits until the limit, specified by the -b flag, is reached (default 16). The *maxbits* specification must be between 9 and 16. (The default can be changed in the source to allow compress to be run on a smaller machine.) After the *maxbits* limit is attained, compress periodically checks the compression ratio. If it is increasing, compress continues to use the existing code dictionary. However, if the compression ratio decreases, compress discards the table of substrings and rebuilds it from scratch. This allows the algorithm to adapt to the next

"block" of the file.

-v print a message yielding the percentage of reduction for each file compressed.

-V print the current version and compile options on the standard output.

Note that the -b flag is omitted for uncompress, since the *maxbits* parameter specified during compression is encoded within the output, along with a magic number to ensure that neither decompression of random data nor recompression of compressed data is attempted.

The amount of compression obtained depends on the size of the input, the maximum number of bits per code, and the distribution of common substrings. Typically, text such as source code or English is reduced by 50 to 60 percent. Compression is generally much better than that achieved by Huffman coding (as used in pack) or adaptive Huffman coding (compact), and takes less time to compute.

Exit status is normally 0; if the last file is larger after (attempted) compression, the status is 2; if an error occurs, exit status is 1.

DIAGNOSTICS

This section lists the error messages displayed on the screen followed by a description of how to recover from the error.

```
Usage: compress [-dfvcV] [-b maxbits] [file...]
```

Invalid options were specified on the command line.

```
Missing maxbits
```

maxbits must follow -b.

file: not in compressed format

The file specified to uncompress has not been compressed.

file: compressed with *xx* bits,
can only handle *yy* bits

file was compressed by a program that could deal with more bits than the compress code on this machine. Recompress the file with smaller *maxbits*.

file: already has .Z suffix -- no change

The file is assumed to be already compressed. Rename the file and try again.

file: filename too long to tack on .Z

The file cannot be compressed because its name is longer than 12 characters. Rename and try again.

file already exists; do you wish to overwrite (y or n)?

Respond y if you want the output file to be replaced; n if not.

uncompress: corrupt input

A SIGSEGV violation was detected, which usually means that the input file has been corrupted.

Compression: *xx.xx*%

Percentage of the input saved by compression. (Relevant only for -v.)

-- not a regular file: unchanged

When the input file is not a regular file (for example, a directory), it is left unaltered.

-- has *xx* other links: unchanged

The input file has links; it is left unchanged. See ln(1) for more information.

-- file unchanged

No savings is achieved by compression. The input remains virgin.

DISCLAIMER

This reference manual entry describes a utility that Apple understands to have been released into the public domain by its author or authors. Apple has included this public domain utility for your convenience. Use it at your own discretion. Often the source code can be obtained if additional requirements are met, such as the purchase of a site license from an author or institution.

FILES

/usr/ucb/compress
/usr/ucb/uncompress
/usr/ucb/zcat

SEE ALSO

compact(1), pack(1).
Terry A. Welch, "A Technique for High Performance Data Compression," *IEEE Computer*, Vol. 17, No. 6 (June 1984), pages 8–19.

BUGS

Although compressed files are compatible between machines with large memory, `-b 12` should be used for file transfer to architectures with a small process data space (64K or less, as exhibited by the DEC PDP series, the Intel 80286, etc.).

NAME

conv — swap bytes in COFF files

SYNOPSIS

conv [–] [–a] [–o] [–p] [–s] –t *target file* ...

DESCRIPTION

The conv command converts object files from their current format to the format of the *target* machine. The converted *file* is written to *file*.v.

Flag options are:

–	read *files* from standard input.
–a	If the input file is an archive, produce the output file in the old archive format.
–o	If the input file is an archive, produce the output file in the UNIX 6.0 (Version 6) portable archive format.
–p	UNIX V.0 random access archive format. This is the default.
–s	Byte-swap all bytes in object file. This is useful only for 3B20 object files which are to be swab-dumped from a DEC machine to a 3B20.
–t *target*	Convert the object file to the byte ordering of the machine (target) to which the object file is being shipped. This may be another host or a target machine. Legal values for *target* are: pdp, vax, ibm, i86, x86, b16, n3b, m32, and m68k.

conv can be used to convert all object files in common object file format. It can be used on either the source (sending) or target (receiving) machine.

conv is meant to ease the problems created by a multihost cross-compilation development environment. conv is best used within a procedure for shipping object files from one machine to another.

conv will recognize and produce archive files in three formats: the UNIX pre-V.0 format, the V.0 random access format, and the 6.0 portable ASCII.

EXAMPLES

```
echo *.out | conv - -t m68k
```

FILES

```
/bin/conv
```

DIAGNOSTICS

All diagnostics for the `conv` command are intended to be self-explanatory. Fatal diagnostics on the command lines cause termination. Fatal diagnostics on an input file cause the program to continue to the next input file.

WARNINGS

`conv` does not convert archives from one format to another if both the source and target machines have the same byte ordering.

NAME

cp — copy files

SYNOPSIS

cp [-i] [-r] *file1 file2*

cp [-i] [-r] *file... directory*

DESCRIPTION

file1 is copied onto *file2*. The mode and owner of *file2* are preserved if it already existed; otherwise, the mode of the source file is used (all bits set in the current umask value are cleared).

In the second form, one or more *files* are copied into the *directory* with their original filenames.

cp refuses to copy a file onto itself.

If the -i flag option is specified, cp will prompt the user with the name of the file whenever the copy will cause an old file to be overwritten. An answer of y will cause cp to continue. Any other answer will prevent it from overwriting the file.

If the -r flag option is specified and any of the source files are directories, cp copies each subtree rooted at that name; in this case, the destination must be a directory.

FILES

/bin/cp

SEE ALSO

cat(1), pr(1), mv(1), rcp(1N).

WARNINGS

cp does not copy the description of special files, but attempts to copy the contents of the special file. This often occurs when using the -r flag option for a recursive copy. For example, cp will hang when trying to copy a named pipe or tty device. When a disk node is being copied, the contents of the disk partition will be copied. To copy the description of the special files, use cpio(1).

NAME

cpio — copy files to or from a cpio archive

SYNOPSIS

cpio -o[acBFv]
cpio -i[Bcdmrtuvfs Sb6] [*patterns*]
cpio -p[adlmuv] *directory*

DESCRIPTION

cpio -o (copy out) reads the standard input to obtain a list of pathnames and copies those files onto the standard output together with pathname and status information. The list of pathnames must contain only one file per line. (Thus, only certain commands, such as find or ls without the -C option, will work in a pipeline to cpio.) Output is padded to a 512-byte boundary. When cpio -o prints a message *xxx* blocks, it indicates how many blocks were written.

cpio -i (copy in) extracts files from the standard input, which is assumed to be the product of a previous cpio -o. Only files with names that match *patterns* are selected. *patterns* are given in the name-generating notation of sh(1). In *patterns*, the meta-characters ?, *, and [...] match the slash / character. Multiple *patterns* may be specified but if none are, the default for *patterns* is * (that is, select all files). The extracted files are conditionally created and copied into the current directory tree based on the flag options described later. The permissions of the files will be those of the previous cpio -o. The owner and group assigned to the files will be that of the current user unless the user is superuser, which causes cpio to retain the owner and group assigned to the files from the previous cpio -o. When cpio -i prints a message *xxx* blocks, it indicates how many blocks were read from the collection.

cpio -p (pass) reads the standard input to obtain a list of pathnames of files that are conditionally created and copied into the destination *directory* tree based on the flag options described later. When cpio -p prints a message *xxx* blocks, it indicates how many blocks were written.

cpio does not follow symbolic links.

The meanings of the available flag options are

a Resets access times of input files after they have been copied.

B	Input/output is to be blocked 5,120 bytes to the record (does not apply to the *pass* flag option; meaningful only with data directed to or from 3.5-inch disks).
d	*directories* are to be created as needed.
c	Writes *header* information in ASCII character form for portability.
r	Interactively *rename* files. If the user types a null line, the file is skipped.
t	Prints a *table of contents* of the input. No files are created.
u	Copies *unconditionally* (normally, an older file will not replace a newer file with the same name).
v	*verbose*: Causes a list of filenames to be printed. When used with the `t` flag option, the table of contents looks like the output of an `ls -l` command (see `ls`(1)).
l	Whenever possible, links files rather than copying them. Usable only with the `-p` flag option.
m	Retains previous file-modification time. This flag option is ineffective on directories that are being copied.
f	Copies in all files except those in *patterns*.
F	When used with the `-o` flag and when the output device is a Macintosh II floppy drive, the `F` flag will cause each floppy to be formatted after it is inserted into the drive. This formatting is for 800K drives only, so only 800K floppy disk should be used.
s	Swaps bytes. Use only with the `-i` flag option.
S	Swaps halfwords. Use only with the `-i` flag option.
b	Swaps both bytes and halfwords. Use only with the `-i` flag option.
6	Processes an old (that is, UNIX System Sixth Edition format) file. Useful only with `-i` flag option.

EXAMPLES

The pipeline

```
ls | cpio -o > /dev/rdsk/c8d0s0
```

copies the contents of a directory into an archive.

```
cd olddir
find . -depth -print | cpio -pdl newdir
```

duplicates a directory hierarchy.

The simple case

```
find . -depth -print | cpio -oB > /dev/rdsk/c8d0s0
```

may be handled more efficiently by:

```
find . -cpio /dev/rdsk/c8d0s0
```

FILES

`/bin/cpio`

SEE ALSO

`ar`(1), `dd`(1), `find`(1), `ls`(1), `tar`(1), `cpio`(4).

BUGS

Pathnames are restricted to 128 characters.

If there are too many uniquely linked files, the program runs out of the memory needed to keep track of them and, thereafter, linking information is lost.

Only the superuser may copy special files.

NAME

cpp — the C language preprocessor

SYNOPSIS

/lib/cpp [-C] [-D*name*[=*def*]] [-I*dir*] [-P] [-U*name*] [-M[*prefix*]] [-Y] [*ifile* [*ofile*]]

DESCRIPTION

cpp is the C language preprocessor that is invoked as the first pass of any C compilation using the cc(1) command. The output of cpp is acceptable as input to the next pass of the C compiler. As the C language evolves, cpp and the rest of the C compilation package will be modified to follow these changes. Therefore, the use of cpp other than in this framework is not suggested. The preferred way to invoke cpp is through the cc(1) command because the functionality of cpp may someday be moved elsewhere. See m4(1) for a general macro processor.

cpp optionally accepts two filenames as arguments. The input for the preprocessor is *ifile*, which is also known as the *source* file, and the output is *ofile*. If not supplied, *ifile* and *ofile* default to standard input and standard output, except in the case of the –M option, which requires *ifile* to be present and have a suffix.

The following flag options to cpp are recognized:

–C Pass along all comments except those found on cpp directive lines. By default, cpp strips C-style comments.

–D*name*
–D*name*=*def*
Define *name* as if by a #define directive. If no =*def* is given, *name* is defined as 1.

–I*dir*
Search for #include files (whose names do not begin with /) in *dir* before looking in the directories on the standard list. When this flag 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 flag 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.

–M[*prefix*]
Generate `make` dependency statements from the `#include` (see below) statements contained in *ifile*. The dependency statements can later be incorporated into a description file for use by `make` in determining the `#include` files on which *ifile* depends. The dependency statements are of the form:

target: *include1* [*include2* ...]

where *target* is a name created by subsituting the suffix of *ifile* with '`.o`'. For example, if the name of *ifile* is `main.c`, *target* will be `main.o`. If *prefix* is present and the `#include` file in *ifile* is found in the standard list, *prefix* is applied as shown below:

target: *prefix/include1* [*prefix/include2* ...]

For example, if the source file `main.c` includes `<stdio.h>` and the –M option is used, the following dependency statement is generated:

```
main.o:      /usr/include/stdio.h
```

If –M'`$(INC)`' is specified, the following dependency statement is generated:

```
main.o:      $(INC)/stdio.h
```

Only unique `#include` files are kept; duplicates are discarded. The resulting *ofile* will contain only dependency statements. Error messages are written on standard error. If `cpp` is invoked with the –M option and *ifile* does not have a suffix, `cpp` exits with an error message.

–P Turn off the default production of line control information. Line control information is used by the next pass of the C compiler to generate useful error messages about the line on which an error occurred. Line control lines have the form

 # lineno file

where *lineno* is the line number within the *file* at which `cpp` found the line. Line control information is useful because, as `cpp` reads each `#include` file and writes its contents to *ofile*, synchronization between the text lines in *ifile* and the

text lines in *ofile* is lost. For example, if the following *ifile* is compiled without line control information

```
#include <stdio.h>

main()
{
        x =) 2;
}
```

the C compiler will generate the following error message:

```
"", line 157: x undefined
"", line 157: syntax error
```

To the C compiler, having included `stdio.h` in *ofile*, line 157 is the true line number at which the error occurred. With line control information enabled, which is the default, `cc` can display the correct line number as shown below:

```
"", line 5: x undefined
"", line 5: syntax error
```

−Uname

Remove 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 variant:	_SYSV_SOURCE _BSD_SOURCE _AUX_SOURCE

`−Y` Prevent `cpp` from searching the standard list, which at present consists of `/usr/include`, when processing `#include` files. This option is useful when used in conjunction with the `−I` option. When `cpp` cannot find a `#include` file in the directories specified by one or more `−I` options, its normal behavior is to search the standard list. If the `−Y` option is used in addition to the `−I` option and `cpp` cannot find a `#include` file in the directory specified by one or more `−I` options, `cpp` writes an error message on standard error, does not search the standard list, and continues processing. Thus, use of the `−Y` option insures that

cpp does not silently include the wrong #include file in cases where the -I option is incorrectly specified or the desired #include file is erroneously missing from the specified directory.

Two special names are understood by cpp. The name __LINE__ is defined as the current line number (as a decimal integer) as known by cpp, and __FILE__ is defined as the current filename (as a C string) as known by cpp. They can be used anywhere, including in macros, just as any other defined name.

All cpp directives start with lines begun by #. The directives are:

#define *name token-string*
Replace subsequent instances of *name* with *token-string*.

#define *name* (*arg*, . . . , *arg*) *token-string*
Replace subsequent instances of *name* followed by a left parenthesis [(], a list of comma-separated tokens, and a right parenthesis [)] by *token-string* where each occurrence of *arg* in the *token-string* is replaced by the corresponding token in the comma-separated list. Notice that there can be no space between *name* and the left parenthesis [(].

#undef *name*
Cause the definition of *name*, if any, to be undefined.

#include *"filename"*
#include <*filename*>
Include at this point the contents of *filename*, which will then be run through cpp. When the <*filename*> notation is used, *filename* is only searched for in the standard places. See the -I flag option above for more detail.

#line *integer-constant "filename"*
Cause cpp to generate line-control information for the next pass of the C compiler. The line number of the next line is *integer-constant*, and *filename* is the file where it comes from. If "*filename*" is not given, the current filename is unchanged.

#endif
End a section of lines begun by a test directive (#if, #ifdef, or #ifndef). Each test directive must have a matching #endif.

`#ifdef` *name*
Displays the lines following in the output if and only if *name* was the subject of a previous `#define` without being the subject of an intervening `#undef`.

`#ifndef` *name*
Does not display the lines following in the output if and only if *name* has been the subject of a previous `#define` without being the subject of an intervening `#undef`.

`#if` *constant-expression*
Displays the lines following in the output if and only if *constant-expression* evaluates to nonzero. All binary nonassignment C operators, the `?:` operator, the unary `-`, `!`, and `~` operators are legal in *constant-expression*. The precedence of the operators is the same as defined by the C language. There is also a unary operator `defined` that can be used in *constant-expression* in these two forms: `defined (` *name* `)` or `defined` *name*. This allows the utility of `#ifdef` and `#ifndef` in an `#if` directive. Only these operators, integer constants, and names that are known by `cpp` should be used in *constant-expression*. In particular, the `sizeof` operator is not available.

`#elif` *constant-expression*
Displays the lines following the output if and only if *constant-expression* is true and the preceding `#if` *constant-expression* with which the subject `#elif` is paired is false. The formation of *constant-expression* is subject to the restrictions described for `#if` above.

`#else`
Reverse the notion of the test directive that matches this directive. If lines previous to this directive are ignored, the lines following appear in the output. If lines previous to this directive are not ignored, the lines following do not appear in the output.

`#ident ...`
This directive is maintained for historical reasons, but does not cause `cpp` to do any special processing nor to generate any output to *ofile*.

`#pragma ...`
Lines beginning with this directive are written to *ofile* without modification.

The test directives and the possible `#else` directives can be nested.

FILES

`/lib/cpp`
`/usr/include`

SEE ALSO

`cc`(1), `m4`(1), `make`(1).

"Other Programming Tools" in *A/UX Programming Languages and Tools, Volume 2*.

DIAGNOSTICS

The error and warning messages produced by `cpp` are self-explanatory and are written to standard output. The line number and filename where the error occurred are printed with the diagnostic. If *ifile* is a relative pathname, `cpp` also prints the absolute path of *ifile* in the form:

ifile (*absolute path*) : *message*

NOTES

When newline characters were found in argument lists for macros to be expanded, previous versions of `cpp` put out the newlines as they were found and expanded. The current version of `cpp` replaces these newlines with blanks to alleviate problems that the previous versions had when this occurred.

NAME

crontab — user crontab utility

SYNOPSIS

crontab [*file*]

crontab -l

crontab -r

DESCRIPTION

crontab is a utility which aids in the use of the cron process scheduling program. A crontab file stipulates the timetable for regular process scheduling. crontab copies the specified file, or standard input if no file is specified, into a directory that holds all users' crontabs. If standard input is used, an EOF (CONTROL-D by default in the A/UX standard distribution) must be entered to terminate the processes. The -r flag option removes a user's crontab from the crontab directory. crontab -l will list the crontab file for the invoking user.

You are permitted to use crontab if your name appears in the file /usr/lib/cron/cron.allow. If that file does not exist, the file /usr/lib/cron/cron.deny is checked to determine if you should be denied access to crontab. If neither file exists, only root is allowed to submit a job. The allow/deny files consist of one username per line.

A crontab file consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns that specify the following:

minute (0–59),
hour (0–23),
day of the month (1–31),
month of the year (1–12),
day of the week (0–6with 0=Sunday).

Each of these patterns may be either an asterisk (meaning all legal values), or a list of elements separated by commas. An element is either a number, or two numbers separated by a minus sign (meaning an inclusive range). Note that the specification of days may be made by two fields (day of the month and day of the week). If both are specified as a list of elements, both are adhered to.

For example,

```
0 0 1,15 * 1
```

would run a command on the first and fifteenth of each month, as well as on every Monday. To specify days by only one field, the other field should be set to * (for example,

```
0 0 * * 1
```

would run a command only on Mondays). Thus, a secondary meaning of asterisk is "use the other field".

The sixth field of a line in a `crontab` file is a string that is executed by the shell at the specified times. A percent character in this field (unless escaped by \) is translated to a newline character. Only the first line (up to a `%` or end-of-line) of the command field is executed by the shell. The other strings following the percent character are made available to the command as standard input. `cron` reads only one line at a time. For example,

```
0 0 * * 1 cat %GO%HOME%EARLY%
```

would mail the output

```
GO
HOME
EARLY
```

to the user at the requested time.

The shell is invoked from your `$HOME` directory with an `arg0` of `sh`. Users who desire to have their `.profile` executed must do so explicitly in the `crontab` file. `cron` supplies a default environment for every shell, defining

```
HOME
LOGNAME
SHELL(=/bin/sh)
PATH(=:/bin:/usr/bin:/usr/lbin)
```

Note: Users should remember to redirect the standard output and standard error of their commands! If this is not done, any generated output or errors will be mailed to the user (via `mail(1)`).

FILES

/usr/bin/crontab
/usr/lib/crontab
/usr/lib/cron
/usr/spool/cron/crontabs
/usr/lib/cron/log
/usr/lib/cron/cron.allow
/usr/lib/cron/cron.deny

SEE ALSO

at(1), sh(1), cron(1M).

NAME

crypt — encode/decode

SYNOPSIS

crypt [*password*]

DESCRIPTION

crypt reads from the standard input and writes on the standard output. The *password* is a key that selects a particular transformation. If no *password* is given, crypt demands a key from the terminal and turns off printing while the key is being typed in. crypt encrypts and decrypts with the same key:

```
crypt key <clear >cypher
crypt key <cypher | pr
```

will print the clear text file, clear.

Files encrypted by crypt are compatible with those treated by both the ed and ex editors in encryption mode.

The security of encrypted files depends on three factors: the fundamental method must be hard to solve; direct search of the key space must be infeasible; *sneak paths* by which keys or clear text can become visible must be minimized. The security of this scheme should not be relied on, for reasons described herein.

crypt implements a one-rotor machine designed along the lines of the German Enigma, but with a 256-element rotor. Methods of attack on such machines are known, but not widely; moreover, the amount of work required is likely to be large.

The transformation of a key into the internal settings of the machine is deliberately designed to be expensive, i.e., to take a substantial fraction of a second to compute. If keys are restricted to (for example) three lowercase letters, however, encrypted files may be read by expending only a substantial fraction of five minutes of machine time.

Since the key is an argument to the crypt command, it is potentially visible to users executing ps(1) or a derivative. To minimize this possibility, crypt takes care to destroy any record of the key immediately upon entry. The choice of keys and key security are the most vulnerable aspect of crypt.

EXAMPLES

```
crypt asa < sleeper.c > zzz
```

will use the string `asa` as key to the encryption algorithm to encrypt the contents of `sleeper.c`, and place the encrypted output in file `zzz`. File `zzz` at this point will be unreadable. Note that the original file, `sleeper.c`, remains in readable form. To obtain readable printout of the file `zzz`, it could be decoded as follows:

```
crypt < zzz
```

After the response:

```
Enter key:
```

the user types in: `asa`.

FILES

`/bin/crypt`
`/dev/tty`
for typed key

SEE ALSO

ed(1), ex(1), makekey(1), stty(1), vi(1), crypt(3C).

BUGS

If output is piped to `nroff` and the encryption key is *not* given on the command line, `crypt` may leave terminal modes in a strange state (see `stty`(1)).

If two or more files encrypted with the same key are concatenated and an attempt is made to decrypt the result, only the contents of the first of the original files will be decrypted correctly.

NOTES

This utility is not provided with international distributions.

NAME

csh — run the C shell, a command interpreter with C-like syntax

SYNOPSIS

csh [-c] [-e] [-f] [-i] [-n] [-s] [-t] [-v] [-V] [-x] [-X] [*arg...*]

DESCRIPTION

csh is a command language interpreter incorporating a history mechanism (see "History Substitutions"), job control facilities (see "Jobs"), and a C-like syntax. In order to use its job control facilities, users of csh must enable the generation of suspend characters with stty(1).

An instance of csh begins by executing commands from the file .cshrc in the home directory of the invoker. If this is a login shell, then it also executes commands from the file .login (also in the home directory). It is typical for users on CRT's to put the tset(1) command in their .login file.

In the normal case, the shell will then begin reading commands from the terminal, prompting with %. Processing of arguments and the use of the shell to process files containing command scripts will be described later.

The shell then repeatedly performs the following actions: a line of command input is read and broken into *words*. This sequence of words is placed on the command history list and then parsed. Finally each command in the current line is executed.

When a login shell terminates, it executes commands from the file .logout in the user's home directory.

Lexical Structure

The shell splits input lines into words at blanks and tabs, with the following exceptions. The characters &, |, ;, <, >, (, and) form separate words. If doubled in &&, ||, <<, or >>, these pairs form single words. These parser metacharacters may be made part of other words, or prevented their special meaning, by preceding them with \. A newline preceded by a \ is equivalent to a blank.

In addition strings enclosed in matched pairs of quotations, `, ', or ", form parts of a word; metacharacters in these strings, including blanks and tabs, do not form separate words. These quotations have semantics to be described subsequently. Within pairs of ' or " characters, a newline preceded by a \ gives a true newline character.

When the shell's input is not a terminal, the character # introduces a comment which continues to the end of the input line. It is prevented this special meaning when preceded by \ and when using `, ', and " quotation.

Commands

A simple command is a sequence of words, the first of which specifies the command to be executed. A simple command or a sequence of simple commands separated by | characters forms a pipeline. The output of each command in a pipeline is connected to the input of the next. Sequences of pipelines may be separated by ;, and are then executed sequentially. A sequence of pipelines may be executed without immediately waiting for it to terminate by following it with an &.

Any of the above may be placed in () to form a simple command (which may be a component of a pipeline, and so forth). It is also possible to separate pipelines with || or &&, indicating, as in the C language, that the second is to be executed only if the first fails or succeeds, respectively (see "Expressions").

Jobs

The shell associates a *job* with each pipeline. It keeps a table of current jobs, printed by the `jobs` command, and assigns them small integer numbers. When a job is started asynchronously with &, the shell prints a line which looks like

```
[1] 1234
```

indicating that the job which was started asynchronously was job number 1 and had one (top-level) process, whose process ID was 1234.

If you are running a job and wish to do something else, you may hit the key CONTROL-Z which sends a stop signal to the current job. The shell will then normally indicate that the job has been `Stopped`, and print another prompt. You can then manipulate the state of this job, putting it in the background with the `bg` command, or run some other commands and then eventually bring the job back into the foreground with the foreground command `fg`. A CONTROL-Z takes effect immediately and is like an interrupt in that pending output and unread input are discarded when it is typed. There is another special key, CONTROL-Y, which does not generate a stop signal until a program attempts to `read`(2) it. This can usefully be typed ahead when you have prepared some commands for a job which you wish to stop after the program has

read them.

A job being run in the background will stop if it tries to read from the terminal. Background jobs are normally allowed to produce output, but this can be disabled by giving the command "`stty tostop`". If you set this tty option, then background jobs will stop when they try to produce output as they do when they try to read input.

There are several ways to refer to jobs in the shell. The character `%` introduces a job name. If you wish to refer to job number 1, you can name it as `%1`. Just naming a job brings it to the foreground; thus `%1` is a synonym for `fg %1`, bringing job 1 back into the foreground. Similarly, saying `%1&` resumes job 1 in the background. Jobs can also be named by prefixes of the string typed in to start them, if these prefixes are unambiguous; thus `%ex` would normally restart a suspended ex(1) job, if there were only one suspended job whose name began with the string `ex`. It is also possible to say `%?`*string* which specifies a job whose text contains *string*, if there is only one such job.

The shell maintains a notion of the current and previous jobs. In output pertaining to jobs, the current job is marked with a + and the previous job with a -. The abbreviation `%+` refers to the current job and `%-` refers to the previous job. For close analogy with the syntax of the `history` mechanism (described later), `%%` is also a synonym for the current job.

Status Reporting

This shell learns immediately whenever a process changes state. It normally informs you whenever a job becomes blocked so that no further progress is possible, but only just before it prints a prompt. This is done so that it does not otherwise disturb your work. If, however, you set the shell variable `notify`, the shell will notify you immediately of changes of status in background jobs. There is also a shell command `notify` which marks a single process so that its status changes will be immediately reported. By default, `notify` marks the current process; simply say `notify` after starting a background job to mark it.

When you try to leave the shell while jobs are stopped, you will be warned that

```
You have stopped jobs.
```

You may use the `jobs` command to see what they are. If you do

this or immediately try to exit again, the shell will not warn you a second time, and the suspended jobs will be terminated.

Substitutions

In this section, various transformations that the shell performs on the input are described, in the order in which they occur.

History Substitutions

History substitutions place words from previous command input in portions of new commands, making it easy to repeat commands, repeat arguments of a previous command in the current command, or fix spelling mistakes in the previous command with little typing and a high degree of confidence. History substitutions begin with the character ! and may begin *anywhere* in the input stream (with the proviso that they *do not* nest). This ! may be preceded by an \ to prevent its special meaning; for convenience, a ! is passed unchanged when it is followed by a blank, tab, newline, =, or (. (History substitutions also occur when an input line begins with ^. This special abbreviation will be described later.) Any input line which contains history substitution is echoed on the terminal before it is executed as it could have been typed without history substitution.

Commands input from the terminal which consist of one or more words are saved on the history list. The history substitutions reintroduce sequences of words from these saved commands into the input stream. The size of this list is controlled by the `history` variable; the previous command is always retained, regardless of its value. Commands are numbered sequentially from 1.

For definiteness, consider the following output from the `history` command

```
 9  write zach
10  ex write.c
11  cat oldwrite.c
12  diff *write.c
```

The commands are shown with their event numbers. It is not usually necessary to use event numbers, but the current event number can be made part of the prompt by placing an ! in the prompt string.

With the current event 13, we can refer to previous events by event number `!11`, relatively as in `!-2` (referring to the same event), by a prefix of a command word as in `!d` for event 12 or

`!wri` for event 9, or by a string contained in a word in the command as in `!?mic?` also referring to event 9. These forms, without further modification, simply reintroduce the words of the specified events, each separated by a single blank. As a special case `!!` refers to the previous command; thus `!!` alone is essentially a redo.

To select words from an event, we can follow the event specification by a `:` and a designator for the desired words. The words of an input line are numbered from 0, the first (usually the command) word being 0, the second word (first argument) being 1, and so forth. The basic word designators are

`0`	first (command) word
n	*n*th argument
`^`	first argument, that is, `1`
`$`	last argument
`%`	word matched by (immediately preceding) `?`*s*`?` search
x-*y*	range of words
-*y*	abbreviates `0-`*y*
`*`	abbreviates `^-$`, or nothing if only 1 word in event
x`*`	abbreviates *x*`-$`
x`-`	like *x*`*` but omitting word `$`

The `:` separating the event specification from the word designator can be omitted if the argument selector begins with a `^`, `$`, `*`, `-`, or `%`. After the optional word designator can be placed a sequence of modifiers, each preceded by a `:`. The following modifiers are defined:

`h`	Remove a trailing pathname component, leaving the head.
`r`	Remove a trailing *.xxx* component, leaving the root name.
`e`	Remove all but the extension *.xxx* part.
`s/`*l*`/`*r*`/`	Substitute *l* for *r*.
`t`	Remove all leading pathname components, leaving the tail.

`&`	Repeat the previous substitution.
`g`	Apply the change globally, prefixing the above, for example `g&`.
`p`	Print the new command but do not execute it.
`q`	Quote the substituted words, preventing further substitutions.
`x`	Like `q`, but break into words at blanks, tabs and newlines.

Unless preceded by a `g`, the modification is applied only to the first modifiable word. With substitutions, it is an error for no word to be applicable.

The left side of substitutions are not regular expressions in the sense of the editors, but rather strings. Any character may be used as the delimiter in place of /; a \ quotes the delimiter into the *l* and *r* strings. The character & on the right side is replaced by the text from the left. A \ quotes & also. A null *l* uses the previous string either from a *l* or from a contextual scan string *s* in `!?`*s*`?`. The trailing delimiter in the substitution may be omitted if a newline follows immediately, as may the trailing `?` in a contextual scan.

A history reference may be given without an event specification, for example `!$`. In this case the reference is to the previous command unless a previous history reference occurred on the same line, in which case this form repeats the previous reference. Thus `!?foo?^ !$` gives the first and last arguments from the command matching `?foo?`.

A special abbreviation of a history reference occurs when the first nonblank character of an input line is a `^`. This is equivalent to `!:s^`, providing a convenient shorthand for substitutions on the text of the previous line. Thus `^lb^lib` fixes the spelling of `lib` in the previous command. Finally, a history substitution may be surrounded with `{` and `}` if necessary to insulate it from the characters which follow. Thus, after `ls -ld ~paul` we might do `!{l}a` to do `ls -ld ~paula`, while `!la` would look for a command starting `la`.

Quotations with ' and "

The quotation of strings by ' and " can be used to prevent all or some of the remaining substitutions. Strings enclosed in ' are prevented any further interpretation. Strings enclosed in " may be

expanded as described later.

In both cases the resulting text becomes (all or part of) a single word; only in one special case (see "Command Substitution") does a " quoted string yield parts of more than one word; ' quoted strings never do.

Alias Substitution

The shell maintains a list of aliases which can be established, displayed, and modified by the `alias` and `unalias` commands. After a command line is scanned, it is parsed into distinct commands and the first word of each command, left-to-right, is checked to see if it has an alias. If it does, then the text which is the alias for that command is reread with the history mechanism available, as though that command were the previous input line. The resulting words replace the command and argument list. If no reference is made to the history list, then the argument list is left unchanged.

Thus, if the alias for `ls` is `ls -l`, the command `ls /usr` would map to `ls -l /usr`, the argument list here being undisturbed. Similarly if the alias for `lookup` was `grep !^ /etc/passwd`, then `lookup tim` would map to `grep tim /etc/passwd`.

If an alias is found, the word transformation of the input text is performed and the aliasing process begins again on the reformed input line. Looping is prevented if the first word of the new text is the same as the old by flagging it to prevent further aliasing. Other loops are detected and cause an error.

Note that the mechanism allows aliases to introduce parser metasyntax. Thus, we can say

```
alias print 'pr \!* | lpr '
```

to make a command which uses `pr` to send its arguments to the line printer.

Variable Substitution

The shell maintains a set of variables, each of which has as value a list of zero or more words. Some of these variables are set by or referred to by the shell. For instance, the `argv` variable is an image of the shell's argument list, and words of this variable's value are referred to in special ways.

The values of variables may be displayed and changed by using the `set` and `unset` commands. Of the variables referred to by the shell, a number are toggles; the shell does not care what their value is, only whether they are set or not. For instance, the `verbose` variable is a toggle which causes command input to be echoed. The setting of this variable results from the `-v` flag option.

Other operations treat variables numerically. The `@` command permits numeric calculations to be performed and the result be assigned to a variable. Variable values are, however, always represented as (zero or more) strings. For the purposes of numeric operations, the null string is considered to be zero, and the second and subsequent words of multiword values are ignored.

After the input line is aliased and parsed, and before each command is executed, variable substitution is performed, keyed by `$` characters. This expansion can be prevented by preceding the `$` with a `\` except within double quotes where it *always* occurs, and within single quotes where it *never* occurs. Strings quoted by `` ` `` are interpreted later (see "Command Substitution"), so `$` substitution does not occur there until later, if at all. A `$` is passed unchanged if followed by a blank, tab, or end-of-line.

Input/output redirections are recognized before variable expansion, and are variable expanded separately. Otherwise, the command name and entire argument list are expanded together. It is thus possible for the first (command) word at this point to generate more than one word, the first of which becomes the command name, and the rest of which become arguments.

Unless enclosed in `"` or given the `:q` modifier, the results of variable substitution may eventually be command and filename substituted. Within `"`, a variable whose value consists of multiple words expands to a (portion of) a single word, with the words of the variables value separated by blanks. When the `:q` modifier is applied to a substitution, the variable will expand to multiple words with each word separated by a blank and quoted to prevent later command or filename substitution.

The following metasequences are provided for introducing variable values into the shell input. Except as noted, it is an error to reference a variable which is not set.

$name

$\{$*name*$\}$
Are replaced by the words of the value of variable *name*, each separated by a blank. Braces insulate *name* from following characters which would otherwise be part of it. Shell variables have names consisting of up to 18 letters and digits starting with a letter. The underscore character is considered a letter.

If *name* is not a shell variable, but is set in the environment, then that value is returned (but : modifiers and the other forms given later are not available in this case).

$*name*[*selector*]
$\{$*name*[*selector*]$\}$
May be used to select only some of the words from the value of *name*. The selector is subjected to $ substitution and may consist of a single number or two numbers separated by a -. The first word of a variables value is numbered 1. If the first number of a range is omitted it defaults to 1. If the last member of a range is omitted it defaults to $#*name*. The selector * selects all words. It is not an error for a range to be empty if the second argument is omitted or in range.

$#*name*
$\{$#*name*$\}$
Gives the number of words in the variable *name*. This is useful for later use in a [*selector*].

$0 Substitutes the name of the file from which command input is being read. An error occurs if the name is not known.

$*number*
$\{$*number*$\}$
Equivalent to $argv[*number*].

$* Equivalent to $argv[*].

The modifiers :h, :t, :r, :q, and :x may be applied to the substitutions above, as may :gh, :gt, and :gr. If braces { } appear in the command form, then the modifiers must appear within the braces.

Note: The current implementation allows only one : modifier on each $ expansion.

The following substitutions may not be modified with : modifiers.

`$?`*name*
`${?`*name*`}`
Substitutes the string `1` if *name* is set, `0` if it is not.

`$?0`
Substitutes `1` if the current input filename is known, `0` if it is not.

`$$` Substitutes the (decimal) process number of the (parent) shell.

`$<` Substitutes a line from the standard input, with no further interpretation thereafter. It can be used to read from the keyboard in a shell script.

Command and Filename Substitution
The remaining substitutions, command and filename substitution, are applied selectively to the arguments of built-in commands. This means that portions of expressions which are not evaluated are not subjected to these expansions. For commands which are not internal to the shell, the command name is substituted separately from the argument list. This occurs very late, after input-output redirection is performed, and in a child of the main shell.

Command Substitution
Command substitution is indicated by a command enclosed in `` ` ``. The output from such a command is normally broken into separate words at blanks, tabs, and newlines, with null words being discarded; this text then replacing the original string. Within `"`, only newlines force new words; blanks and tabs are preserved.

In any case, the single final newline does not force a new word. Note that it is thus possible for a command substitution to yield only part of a word, even if the command outputs a complete line.

Filename Substitution
If a word contains any of the characters `*`, `?`, `[`, or `{`, or begins with the character `~`, then that word is a candidate for filename substitution, also known as *globbing*. This word is then regarded as a pattern and replaced with an alphabetically sorted list of file names which match the pattern. In a list of words specifying filename substitution, it is an error if no pattern matches an existing filename, but it is not required that each pattern match. Only the metacharacters `*`, `?`, and `[` imply pattern matching, the charac-

ters ~ and { being more akin to abbreviations.

In matching filenames, the character . at the beginning of a filename or immediately following a /, as well as the character /, must be matched explicitly. The character * matches any string of characters, including the null string. The character ? matches any single character. The sequence [...] matches any one of the characters enclosed. Within [...], a pair of characters separated by - matches any character lexically between the two.

The character ~ at the beginning of a filename is used to refer to home directories. Standing alone, that is, as ~, it expands to the invokers home directory as reflected in the value of the variable home. When followed by a name consisting of letters, digits and - characters, the shell searches for a user with that name and substitutes the home directory; thus ~paul might expand to /usr/paul and ~paul/chmach to /usr/paul/chmach. If the character ~ is followed by a character other than a letter or / or appears, but not at the beginning of a word, it is left undisturbed.

The metanotation a{b,c,d}e is shorthand for abe ace ade. Left to right order is preserved, with results of matches being sorted separately at a low level to preserve this order. This construct may be nested. Thus,

```
~source/s1/{oldls,ls}.c
```

expands to

```
/usr/source/s1/oldls.c /usr/source/s1/ls.c
```

(whether or not these files exist without any chance of error) if the home directory for source is /usr/source. Similarly,

```
../{memo,*box}
```

might expand to

```
../memo ../box ../mbox
```

(Note that memo was not sorted with the results of matching *box.) As a special case {, }, and {} are passed undisturbed.

Input/Output

The standard input and standard output of a command may be redirected with the following syntax:

< *name*

Open file *name* (which is first variable, command, and

filename expanded) as the standard input.

<< *word*
Read the shell input up to a line which is identical to *word*. *word* is not subjected to variable, filename, or command substitution, and each input line is compared to *word* before any substitutions are done on this input line. Unless a quoting \, ", ', or ` appears in *word*, variable and command substitution is performed on the intervening lines, allowing \ to quote $, \, and `. Commands which are substituted have all blanks, tabs, and newlines preserved, except for the final newline which is dropped. The resultant text is placed in an anonymous temporary file which is given to the command as standard input.

> *name*
>! *name*
>& *name*
>&! *name*
The file *name* is used as standard output. If the file does not exist, then it is created; if the file exists, it is truncated and its previous contents are lost. If the variable `noclobber` is set, then either the file must not exist or be a character special file (for example, a terminal or `/dev/null`) or an error results. This helps prevent accidental destruction of files. In this case, the ! forms can be used and suppress this check. The forms involving & route the diagnostic output as well as the standard output into the specified file. *name* is expanded in the same way as < input filenames are.

>> *name*
>>& *name*
>>! *name*
>>&! *name*
Uses file *name* as standard output like >, but places output at the end of the file. If the variable `noclobber` is set, then it is an error for the file not to exist unless one of the ! forms is given. Otherwise similar to >.

A command receives the environment in which the shell was invoked as modified by the input-output parameters and the presence of the command in a pipeline. Thus, unlike some previous shells, commands that run from a file of shell commands have no access to the text of the commands by default; rather they receive the original standard input of the shell. The << mechanism should

be used to present inline data. This permits shell command scripts to function as components of pipelines and allows the shell to block-read its input. Note that the default standard input for a command run detached is *not* modified to be the empty file `/dev/null`; rather the standard input remains as the original standard input of the shell. If this is a terminal and if the process attempts to read from the terminal, then the process will block and the user will be notified (see "Jobs".)

Diagnostic output may be directed through a pipe with the standard output. Simply use the form `|&` rather than just `|`.

Expressions

A number of the built-in commands (to be described subsequently) take expressions, in which the operators are similar to those of C, with the same precedence. These expressions appear in the `@`, `exit`, `if`, and `while` commands. The following operators are available:

```
|| && | ^ & == != =~ !~ <=
>= < > << >> + - * / % ! ~ ( )
```

Here the precedence increases to the right with those on the same line having equal precedence:

```
== != =~ !~
<= >= < >
<< >>
+ -
* / %
```

The `==`, `!=`, `=~`, and `!~` operators compare their arguments as strings; all others operate on numbers. The operators `=~` and `!~` are like `==` and `!=` except that the right side is a pattern (containing, for example, `*`, `?`, and instances of `[...]`) against which the left operand is matched. This reduces the need for use of the `switch` statement in shell scripts when all that is really needed is pattern matching.

Strings which begin with `0` are considered octal numbers. Null or missing arguments are considered `0`. The result of all expressions are strings, which represent decimal numbers. It is important to note that no two components of an expression can appear in the same word; except when adjacent to components of expressions which are syntactically significant to the parser

```
& | < > ( )
```

they should be surrounded by spaces.

Also available in expressions as primitive operands are command executions enclosed in { and } and file enquiries of the form -*l name* where *l* is one of

r	read access
w	write access
x	execute access
e	existence
o	ownership
z	zero size
f	plain file
d	directory

The specified name is command- and filename-expanded and then tested to see if it has the specified relationship to the real user. If the file does not exist or is inaccessible, then all enquiries return false, that is 0. Command executions succeed, returning true, that is 1, if the command exits with status 0, otherwise they fail, returning false, that is 0. If more detailed status information is required then the command should be executed outside of an expression and the variable status examined.

Control Flow

The shell contains a number of commands which can be used to regulate the flow of control in command files (shell scripts) and (in limited but useful ways) from terminal input. These commands all operate by forcing the shell to reread or skip in its input and, due to the implementation, restrict the placement of some of the commands.

The foreach, switch, and while statements, as well as the if-then-else form of the if statement require that the major keywords appear in a single simple command on an input line as shown later.

If the shell's input is not seekable, the shell buffers up input whenever a loop is being read and performs seeks in this internal buffer to accomplish the rereading implied by the loop. (To the extent that this allows, backward goto's will succeed on nonseekable inputs.)

Built-in Commands

Built-in commands are executed within the shell. If a built-in command occurs as any component of a pipeline except the last then it is executed in a subshell.

`alias`
`alias` *name*
`alias` *name wordlist*
: The first form prints all aliases. The second form prints the alias for *name*. The final form assigns the specified *wordlist* as the alias of *name*; *wordlist* is command- and filename-substituted. *name* is not allowed to be `alias` or `unalias`.

`alloc`
: Shows the amount of dynamic core in use, broken down into used and free core, and the address of the last location in the heap. `alloc` used with an argument shows each used and free block on the internal dynamic memory chain indicating its address, size, and whether it is used or free. This is a debugging command and may not work in production versions of the shell; it requires a modified version of the system memory allocator.

`bg`
`bg` `%`*job* ...
: Puts the current or specified jobs into the background, continuing them if they were stopped.

`break`
: Causes execution to resume after the `end` of the nearest enclosing `foreach` or `while`. The remaining commands on the current line are executed. Multilevel breaks are thus possible by writing them all on one line.

`breaksw`
: Causes a break from a `switch`, resuming after the `endsw`.

`case` *label*`:`
: A label in a `switch` statement as discussed later under `switch`.

`cd`
`cd` *name*
`chdir`
`chdir` *name*
: Changes the shell's working directory to directory *name*. If

no argument is given, then change to the home directory of the user. If *name* is not found as a subdirectory of the current directory (and does not begin with /, ./, or ../), then each component of the variable `cdpath` is checked to see if it has a subdirectory *name*. Finally, if all else fails but *name* is a shell variable whose value begins with /, then its value is tried to see if it is a directory.

`continue`
Continues execution of the nearest enclosing `while` or `foreach`. The rest of the commands on the current line are executed.

`default:`
Labels the default case in a `switch` statement. The default should come after all `case` labels.

`dirs`
Prints the directory stack; the top of the stack is at the left, the first directory in the stack being the current directory.

`echo` *wordlist*
`echo -n` *wordlist*
The specified words are written to the shells standard output, separated by spaces and terminated with a newline unless the `-n` flag option is specified.

`else`
`end`
`endif`
`endsw`
See the description of the `foreach`, `if`, `switch`, and `while` statements later in this section.

`eval` *arg...*
The arguments are read as input to the shell (as in `sh`(1)) and the resulting command(s) is executed in the context of the current shell. This is usually used to execute commands generated as the result of command or variable substitution, since parsing occurs before these substitutions. See `tset`(1) for an example of using `eval`.

`exec` *command*
The specified command is executed in place of the current shell.

`exit`
`exit` (*expr*)
The shell exits either with the value of the `status` variable (first form) or with the value of the specified *expr* (second form).

`fg`
`fg` %*job*...
Brings the current or specified jobs into the foreground, continuing them if they were stopped.

`foreach` *name* (*wordlist*)
...
`end`
The variable *name* is successively set to each member of *wordlist* and the sequence of commands between this command and the matching `end` are executed. (Both `foreach` and `end` must appear singly on separate lines.)

The built-in command `continue` may be used to continue the loop prematurely and the built-in command `break` to terminate it prematurely. When this command is read from the terminal, the loop is read up once prompting with `?` before any statements in the loop are executed. If you make a mistake typing in a loop at the terminal, you can interrupt it.

`glob` *wordlist*
Like `echo` but no \ escapes are recognized and words are delimited by null characters in the output. Useful for programs which wish to use the shell to filename-expand a list of words.

`goto` *word*
The specified *word* is filename- and command-expanded to yield a string of the form *label*. The shell rewinds its input as much as possible and searches for a line of the form *label*: possibly preceded by blanks or tabs. Execution continues after the specified line.

`hashstat`
Prints a statistics line indicating how effective the internal hash table has been at locating commands (and avoiding any `exec`). An `exec` is attempted for each component of the *path* where the hash function indicates a possible hit, and in each component which does not begin with a `/`.

`history`
`history` *n*
`history -r` *n*
`history` *h* `n`
Displays the history event list; if *n* is given, only the `n` most recent events are printed. The `-r` flag option reverses the order of printout to be most recent first rather than oldest first. The `-h` flag option causes the history list to be printed without leading numbers and is used to produce files suitable for sourcing using the `-h` flag option to `source`.

`if (`*expr*`)` *command*
If the specified *expr* evaluates true, then the single *command* with arguments is executed. In the interactive shell, the `if` statement can only accept one simple command after the *expr* and in the same line as *expr*. Variable substitution on *command* happens early, at the same time it does for the rest of the `if` command. The *command* must be a simple command, not a pipeline, a command list, or a parenthesized command list. Input/output redirection occurs even if *expr* is false, when *command* is *not* executed (this is a bug).

`if (`*expr*`) then`
...
`else if (`*expr2*`) then`
...
`else`
...
`endif`
If the specified *expr* is true, then the first command is executed. In the interactive shell, the `if then` statement can only accept one simple command after `then`. This command must be specified on the same line as `then`. If the specified *expr2* is true, then the command to the `else` or `else if` are executed, and so forth. Any number of `else if` pairs are possible; only one `endif` is needed. The `else` part is likewise optional. (The words `else` and `endif` must appear at the beginning of input lines; the `if` must appear alone on its input line or after an `else`.)

`jobs`
`jobs -l`
Lists the active jobs; the `-l` flag option lists process ID's in addition to the normal information.

`kill %`*job*
`kill -`*sig* `%`*job*...
`kill` *pid*
`kill -`*sig* `pid...`
`kill -l`

Sends either the TERM (terminate) signal or the specified signal to the specified jobs or processes. Signals are either given by number or by names (as given in `/usr/include/signal.h`, stripped of the prefix `SIG`). The signal names are listed by `kill -l`. There is no default; just saying `kill` does not send a signal to the current job. A *pid* of 0 means the current process (that is, this invocation of the C shell). Consequently, `kill -9 0` terminates the current C shell and possibly logs you off. If the signal being sent is TERM (terminate) or HUP (hangup), then the job or process will be sent a CONT (continue) signal as well.

`login`

Terminates a login shell, replacing it with an instance of `/bin/login`. This is one way to log off, included for compatibility with sh(1).

`logout`

Terminates a login shell. Especially useful if `ignoreeof` is set.

`nice`
`nice +`*number*
`nice` *command*
`nice +`*number command*

The first form sets the `nice` for this shell to 4. The second form sets the `nice` to the given number. The final two forms run command at priority 4 and `number` respectively. The superuser may specify negative niceness by using

`nice -`*number* ...

Command is always executed in a subshell, and the restrictions placed on commands in simple `if` statements apply.

`nohup`
`nohup` *command*

The first form can be used in shell scripts to cause hangups to be ignored for the remainder of the script. The second form causes the specified command to be run with hangups ig-

nored. All processes detached with & are run effectively without hangups.

`notify`
`notify %`*job*...
Causes the shell to notify the user asynchronously when the status of the current or specified jobs changes; normally notification is presented before a prompt. This is automatic if the shell variable `notify` is set.

`onintr`
`onintr -`
`onintr` *label*
Controls the action of the shell on interrupts. The first form restores the default action of the shell on interrupts which is to terminate shell scripts or to return to the terminal command input level. The second form `onintr -` causes all interrupts to be ignored. The final form causes the shell to execute a `goto label` when an interrupt is received or a child process terminates because it was interrupted.

In any case, if the shell is running detached and interrupts are being ignored, all forms of `onintr` have no meaning and interrupts continue to be ignored by the shell and all invoked commands.

`popd`
`popd +`*n*
Pops the directory stack, returning to the new top directory. With the argument +*n*, `popd` discards the *n*th entry in the stack. The elements of the directory stack are numbered from 0 starting at the top.

`pushd`
`pushd` *name*
`pushd +`*n*
With no arguments, `pushd` exchanges the top two elements of the directory stack. Given a *name* argument, `pushd` changes to the new directory (like `cd`) and pushes the old current working directory (as in `csw`) onto the directory stack. With a numeric argument, `pushd` rotates the *n*th argument of the directory stack around to be the top element and changes into it. The members of the directory stack are numbered from the top starting at 0.

`rehash`
Causes the internal hash table of the contents of the directories in the `path` variable to be recomputed. This is needed if new commands are added to directories in the `path` while you are logged in. This should only be necessary if you add commands to one of your own directories, or if a systems programmer changes the contents of one of the system directories.

`repeat` *count command*
The specified *command* (which is subject to the same restrictions as the *command* in the one line `if` statement above) is executed *count* times. I/O redirections occur exactly once, even if *count* is 0.

`set`
`set` *name*
`set` *name=word*
`set` *name*`[`*index*`]=`*word*
`set` *name=* `(`*wordlist*`)`
The first form of the command shows the value of all shell variables. Variables which have other than a single word as value print as a parenthesized word list. The second form sets *name* to the null string. The third form sets *name* to the single *word*. The fourth form sets the *index* component of *name* to *word*; this component must already exist. The final form sets *name* to the list of words in *wordlist*. In all cases the value is command- and filename-expanded.

These arguments may be repeated to set multiple values in a single set command. Note however, that variable expansion happens for all arguments before any setting occurs.

`setenv` *name value*
Sets the value of the environment variable *name* to be *value*, a single string. The most commonly used environment variables `USER`, `TERM`, and `PATH` are automatically imported to and exported from the `csh` variables `user`, `term`, and `path`; there is no need to use `setenv` for these.

`shift`
`shift` *variable*
The members of `argv` are shifted to the left, discarding `argv[1]`. It is an error for `argv` not to be set or to have less than one word as value. The second form performs the

same function on the specified variable.

`source` *name*
`source -h` *name*
: The shell reads commands from *name*. `source` commands may be nested; if they are nested too deeply, the shell may run out of file descriptors. An error in a `source` at any level terminates all nested `source` commands. Normally, input during `source` commands is not placed on the history list; the `-h` flag option causes the commands to be placed in the history list without being executed.

`stop %`*job* ...
: Stops the current or specified job which is executing in the background.

`suspend`
: Causes the shell to stop in its tracks, much as if it had been sent a stop signal with CONTROL-Z. This is most often used to stop shells started by `su`(1).

```
switch (string)
case str1:
...
    breaksw
...
default:
...
    breaksw
endsw
```
: Each case label is successively matched, against the specified *string* which is first command- and filename-expanded. The file metacharacters `*`, `?`, and `[...]` may be used in the case labels, which are variable-expanded. If none of the labels match before a `default` label is found, then the execution begins after the `default` label. Each `case` label and the `default` label must appear at the beginning of a line. The command `breaksw` causes execution to continue after the `endsw`; otherwise control may fall through `case` labels and `default` labels as in C. If no label matches and there is no default, execution continues after the `endsw`.

`time`
`time` *command*
: With no argument, a summary of time used by this shell and

its children is printed. If arguments are given, the specified simple command is timed and a time summary as described under the `time` variable is printed. If necessary, an extra shell is created to print the time statistic when the command completes.

`umask`
`umask` *value*
The file creation mask is displayed (first form) or set to the specified value (second form). The mask is given in octal. Common values for the mask are 002, giving all access to the group and read and execute access to others, or 022, giving all access except no write access for users in the group or others.

`unalias` *pattern*
All aliases whose names match the specified pattern are discarded. Thus all aliases are removed by `unalias *`. It is not an error for there to be nothing to `unalias`.

`unhash`
Use of the internal hash table to speed location of executed programs is disabled.

`unset` *pattern*
All variables whose names match the specified pattern are removed. Thus all variables are removed by `unset *` which has noticeably distasteful side-effects. It is not an error for nothing to be `unset`.

`unsetenv` *pattern*
Removes all variables whose name match the specified pattern from the environment. See also the `setenv` command and `printenv`(1).

`wait`
All background jobs are waited for. If the shell is interactive, then an interrupt can disrupt the wait, at which time the shell prints names and job numbers of all jobs known to be outstanding.

`while (`*expr*`)`
...
`end`
While the specified expression evaluates nonzero, the commands between the `while` and the matching `end` are evaluated. `break` and `continue` may be used to terminate

or continue the loop prematurely. (The `while` and `end` must appear alone on their input lines.) Prompting occurs here, the first time through the loop, as for the `foreach` statement if the input is a terminal.

`%`*job*
Brings the specified job into the foreground.

`%`*job* `&`
Continues the specified job in the background.

`@`
`@` *name=expr*
`@` *name*`[`*index*`]=`*expr*
The first form prints the values of all the shell variables. The second form sets the specified *name* to the value of *expr*. If the expression contains <, >, `&`, or `|`, then at least this part of the expression must be placed within `()`. The third form assigns the value of *expr* to the *index* argument of *name*. Both *name* and its *index* component must already exist.

The operators `*=`, `+=`, and so forth, are available as in C. The space separating the name from the assignment operator is optional. Spaces are, however, mandatory in separating components of *expr* which would otherwise be single words.

Special postfix `++` and `--` operators increment and decrement *name* respectively; for example `@ i++`.

Predefined and Environment Variables

The following variables have special meaning to the shell. Of these, `argv`, `cwd`, `home`, `path`, `prompt`, `shell`, and `status` are always set by the shell. Except for `cwd` and `status`, this setting occurs only at initialization; these variables then will not be modified except explicitly by the user.

This shell copies the environment variable `USER` into the variable `user`, `TERM` into `term`, and `HOME` into `home`, and copies these back into the environment whenever the normal shell variables are reset. The environment variable `PATH` is likewise handled; it is not necessary to worry about its setting other than in the file `.cshrc`, as inferior `csh` processes will import the definition of `path` from the environment and re-export it if you change it.

`argv` Set to the arguments to the shell, it is from this variable that positional parameters are substituted, that is, `$1` is replaced by `$argv[1]`, and

so forth.

`cdpath` Gives a list of alternate directories searched to find subdirectories in `chdir` commands.

`cwd` The full pathname of the current directory.

`echo` Set when the `-x` command line option is given. `echo` causes each command and its arguments to be echoed just before it is executed. For nonbuilt-in commands, all expansions occur before echoing. Built-in commands are echoed before command and filename substitution, since these substitutions are then done selectively.

`histchars` Can be given a string value to change the characters used in history substitution. The first character of its value is used as the history substitution character, replacing the default character `!`. The second character of its value replaces the character `^` in quick substitutions.

`history` Can be given a numeric value to control the size of the history list. Any command which has been referenced in this many events will not be discarded. Values of `history` that are too large may run the shell out of memory. The last executed command is always saved on the history list.

`home` The home directory of the invoker, initialized from the environment. The filename expansion of `~` refers to this variable.

`ignoreeof` If set, the shell ignores end-of-file from input devices which are terminals. This prevents shells from accidentally being killed by CONTROL-D's.

`mail` The files where the shell checks for mail. This is done after each command completion which will result in a prompt, if a specified interval has elapsed. The shell says

```
You have new mail
```

if the file exists with an access time not greater

than its modification time.

If the first word of the value of `mail` is numeric, it specifies a different mail checking interval, in seconds, than the default, which is 10 minutes.

If multiple mail files are specified, then the shell says

```
New mail in name
```

when there is mail in the file *name*.

`noclobber` As described in the section "Input/Output," restrictions are placed on output redirection to insure that files are not accidentally destroyed and that >> redirections refer to existing files.

`noglob` If set, filename expansion is inhibited. This is most useful in shell scripts which are not dealing with filenames, or after a list of filenames has been obtained and further expansions are not desirable.

`nonomatch` If set, it is not an error if a filename expansion does not match any existing files; rather the primitive pattern is returned. It is still an error for the primitive pattern to be malformed, that is, `echo [` still gives an error.

`notify` If set, the shell notifies asynchronously of job completions. The default is to present job completions just before printing a prompt.

`path` Each word of the path variable specifies a directory in which commands are to be sought for execution. A null word specifies the current directory. If there is no `path` variable, then only full pathnames will execute. The usual search path is `.`, `/bin`, and `/usr/bin`, but this may vary from system to system. For the superuser, the default search path is `/etc`, `/bin`, and `/usr/bin`. A shell which is given neither the `-c` nor the `-t` flag option will normally hash the contents of the directories in the `path` variable after reading `.cshrc`, and each time the `path` variable is reset. If new com-

mands are added to these directories while the shell is active, it may be necessary to give the `rehash` or the commands may not be found.

`prompt` The string which is printed before each command is read from an interactive terminal input. If a `!` appears in the string, it will be replaced by the current event number unless a preceding `\` is given. Default is `%` or `#` for the superuser.

`savehist` a numeric value is given to control the number of entries of the history list that are saved in `~/.history` when the user logs out. Any command which has been referenced in that number of events will be saved. During start up the shell sources `~/.history` into the history list, enabling history to be saved across logins. Values of `savehist` that are too large will slow down the shell during start up.

`shell` The file in which the shell resides. This is used in forking shells to interpret files which have execute bits set, but which are not executable by the system. (See the description of "Nonbuilt-in Command Execution" later.) Initialized to the (system-dependent) home of the shell.

`status` The status returned by the last command. If it terminated abnormally, then 0200 is added to the status. Built-in commands which fail return exit status `1`; all other built-in commands set status `0`.

`time` Controls automatic timing of commands. If set, then any command which takes more than this many cpu seconds will cause a line to be printed when it terminates. This line shows user, system, and real times and a utilization percentage which is the ratio of user plus system times to real time

`verbose` Set by the `-v` flag option, causes the words of each command to be printed after history substitution.

Nonbuilt-in Command Execution

When a command to be executed is found to not be a built-in command, the shell attempts to execute the command via `execve(2)`. Each word in the variable `path` names a directory from which the shell will attempt to execute the command. If it is given neither a `-c` nor a `-t` flag option, the shell will hash the names in these directories into an internal table so that it will only try an `exec` in a directory if there is a possibility that the command resides there. This greatly speeds command location when a large number of directories are present in the search path. If this mechanism has been turned off (via `unhash`), or if the shell was given a `-c` or `-t` argument (and in any case for each directory component of `path` which does not begin with a /), the shell concatenates with the given command name to form a pathname of a file which it then attempts to execute.

Parenthesized commands are always executed in a subshell. Thus,

```
(cd ; pwd) ; pwd
```

prints the *home* directory, leaving you where you were (printing this after the home directory), while

```
cd ; pwd
```

leaves you in the *home* directory. Parenthesized commands are most often used to prevent `chdir` from affecting the current shell.

If the file has execute permissions but is not an executable binary to the system, then it is assumed to be a file containing shell commands and a new shell is spawned to read it.

If there is an `alias` for `shell`, then the words of the alias will be prefixed to the argument list to form the shell command. The first word of the `alias` should be the full pathname of the shell (for example `$shell`). Note that this is a special, late occurring, case of `alias` substitution, and only allows words to be prefixed to the argument list without modification.

Argument List Processing

If argument 0 to the shell is - then this is a login shell. The flag options are interpreted as follows:

`-c` Commands are read from the (single) following argument which must be present. Any remaining arguments are placed in `argv`.

`-e` The shell exits if any invoked command terminates abnormally or yields a nonzero exit status.

`-f` The shell will start faster, because it will neither search for nor execute commands from the file `.cshrc` in the invoker's home directory.

`-i` The shell is interactive and prompts for its top-level input, even if it appears to not be a terminal. Shells are interactive without this option if their inputs and outputs are terminals.

`-n` Commands are parsed, but not executed. This aids in syntactic checking of shell scripts.

`-s` Command input is taken from the standard input.

`-t` A single line of input is read and executed. A \ may be used to escape the newline at the end of this line and continue onto another line.

`-v` Causes the `verbose` variable to be set, with the effect that command input is echoed after history substitution.

`-x` Causes the `echo` variable to be set, so that commands are echoed immediately before execution.

`-V` Causes the `verbose` variable to be set even before `.cshrc` is executed.

`-X` Causes the `echo` variable to be set even before `.cshrc` is executed.

After processing of flag arguments, if arguments remain but none of the `-c`, `-i`, `-s`, or `-t` flag options were given, the first argument is taken as the name of a file of commands to be executed. The shell opens this file and saves its name for possible resubstitution by `$0`. Since many systems use the standard Bourne shell (`/bin/sh`), whose shell scripts are not compatible with this shell, the shell will execute such a *standard* shell if the first character of a script is not a #, that is, if the script does not start with a comment. Remaining arguments initialize the variable `argv`.

Signal Handling

The shell normally ignores `quit` signals. Jobs running detached (either by `&` or the `bg` or `%...&` commands) are immune to signals generated from the keyboard, including hangups. Other signals have the values which the shell inherited from its parent. The shells handling of interrupts and terminate signals in shell scripts can be controlled by `onintr`. Login shells catch the `ter-`

minate signal; otherwise this signal is passed on to children from the state in the shell's parent. In no case are interrupts allowed when a login shell is reading the file `.logout`.

FILES

`/bin/csh`	
`~/.cshrc`	read at beginning of execution by each shell.
`/etc/cshrc`	global file read by login shell before `~/.cshrc`
`~/.login`	read by login shell, after `.cshrc` at login.
`~/.logout`	read by login shell, at logout.
`/bin/sh`	standard shell, for shell scripts not starting with a #.
`/tmp/sh*`	temporary file for <<.
`/etc/passwd`	source of home directories for ~*name*.

LIMITATIONS

Words can be no longer than 1024 characters. The system limits argument lists to 10240 characters. The number of arguments to a command which involves filename expansion is limited to 1/6th the number of characters allowed in an argument list. Command substitutions may substitute no more characters than are allowed in an argument list. To detect looping, the shell restricts the number of `alias` substitutions on a single line to 20.

SEE ALSO

`ksh`(1), `sh`(1), `access`(2), `exec`(2), `fork`(2), `pipe`(2), `sigvec`(2), `umask`(2), `wait`(2), `killpg`(3N), `a.out`(4), `environ`(5), `tty`(7).
"C Shell Reference" in *A/UX User Interface*.

BUGS

When a command is restarted from a stop, the shell prints the directory it started in if this is different from the current directory; this can be misleading (that is, wrong) as the job may have changed directories internally.

Shell built-in functions are not stoppable/restartable. Command sequences of the form

a ; b ; c

are also not handled gracefully when stopping is attempted. If you suspend *b*, the shell will then immediately execute *c*. This is especially noticeable if this expansion results from an `alias`. It

suffices to place the sequence of commands in `()` to force it to a subshell.

```
(a ; b ; c)
```

Control over tty output after processes are started is primitive; perhaps this will inspire someone to work on a good virtual terminal interface. In a virtual terminal interface, much more interesting things could be done with output control.

Alias substitution is most often used to simulate shell procedures; shell procedures should be provided rather than aliases.

Commands within loops, prompted for by `?`, are not placed in the `history` list. Control structure should be parsed rather than recognized as built-in commands, allowing control commands to be placed anywhere, to be combined with `|`, and to be used with `&`, and `;` metasyntax.

It should be possible to use the `:` modifiers on the output of command substitutions. All and more than one `:` modifier should be allowed on `$` substitutions.

Symbolic links fool the shell. In particular, `dirs` and

```
cd ..
```

don't work properly once you've crossed through a symbolic link.

NAME

csplit — context split

SYNOPSIS

csplit [-f *prefix*] [-k] [-s] *file arg1* [...*argn*]

DESCRIPTION

csplit reads *file* and separates it into *n*+1 sections, defined by the arguments *arg1*... *argn*. By default, the sections are placed in files named xx00... xx*n* (*n* may not be greater than 99). These sections get the following pieces of *file*:

00: From the start of *file* up to (but not including) the line referenced by *arg1*.

01: From the line referenced by *arg1* up to the line referenced by *arg2*.

.

.

.

n+1: From the line referenced by *argn* to the end of *file*.

If the *file* argument is a – then standard input is used.

The flag options to csplit are:

-s csplit normally prints the character counts for each file created. If the -s flag option is present, csplit suppresses the printing of all character counts.

-k csplit normally removes created files if an error occurs. If the -k flag option is present, csplit leaves previously created files intact.

-f *prefix* If the -f flag option is used, the created files are named *prefix*00 ... *prefixn*. The default is **xx00** ... xx*n*.

The arguments (*arg1* ... *argn*) to csplit can be a combination of the following:

/rexp/ A file is to be created for the section from the current line up to (but not including) the line containing the regular expression *rexp*. The current line becomes the line containing *rexp*. This argu-

ment may be followed by an optional + or – some number of lines (e.g., /Page/–5).

%rexp% This argument is the same as */rexp/*, except that no file is created for the section.

lnno A file is to be created from the current line up to (but not including) *lnno*. The current line becomes *lnno*.

{*num*} Repeat argument. This argument may follow any of the above arguments. If it follows a *rexp* type argument, that argument is applied *num* more times. If it follows *lnno*, the file will be split every *lnno* lines (*num* times) from that point.

Enclose all *rexp* type arguments that contain blanks or other characters meaningful to the Shell in the appropriate quotes. Regular expressions may not contain embedded newlines. csplit does not affect the original file; it is the user's responsibility to remove it.

EXAMPLES

```
csplit -f cobol file '/procedure
division/' /par5./  /par16./
```

creates four files, "cobol00 . . . cobol03". After editing the split files, they can be recombined as follows:

```
cat cobol0[0-3] > file
```

Note that this example overwrites the original file.

```
csplit -k file  100  {99}
```

splits the file at every 100 lines, up to 10,000 lines. The -k flag option causes the created files to be retained if there are less than 10,000 lines; however, an error message would still be printed.

```
csplit -k prog.c  '%main(%'  '/^}/+1'  {20}
```

assuming that prog.c follows the normal C coding convention of ending routines with a } at the beginning of the line, this example will create a file containing each separate C routine (up to 21) in prog.c.

FILES

/usr/bin/csplit

SEE ALSO

ed(1), fsplit(1), sh(1), split(1), regexp(5).

DIAGNOSTICS

Self explanatory except for:

```
arg - out of range
```

which means that the given argument did not reference a line between the current position and the end of the file.

NAME

ct — spawn getty to a remote terminal

SYNOPSIS

ct [-h] [-v] [-w*n*] [-s*speed*] *telno* ...

DESCRIPTION

ct dials the phone number of a modem that is attached to a terminal, and spawns a getty(1M) process to that terminal. *telno* is a telephone number, with equals signs for secondary dial tones and minus signs for delays at appropriate places. If more than one telephone number is specified, ct will try each in succession until one answers; this is useful for specifying alternate dialing paths.

ct will try each line listed in the file

```
/usr/lib/uucp/L-devices
```

until it finds an available line with appropriate attributes or runs out of entries. If there are no free lines, ct will ask if it should wait for one, and if so, for how many minutes it should wait before it gives up. ct will continue to try to open the dialers at one-minute intervals until the specified limit is exceeded. The dialogue may be overridden by specifying the *-wn* flag option, where *n* is the maximum number of minutes that ct is to wait for a line.

Normally, ct will hang up the current line, so that that line can answer the incoming call. The -h flag option will prevent this action. If the -v flag option is used, ct will send a running narrative to the standard error output stream.

The data rate may be set with the -s flag option, where *speed* is expressed in baud. The default rate is 300.

After the user on the destination terminal logs out, ct prompts, Reconnect?. If the response begins with the letter n the line will be dropped; otherwise, getty will be started again and the login: prompt will be printed.

Of course, the destination terminal must be attached to a modem that can answer the telephone.

EXAMPLES

```
ct -w15 -s1200 555-9999
```

dials from the terminal the given modem phone number (555-9999), spawning a login process at 1200 baud. If the dialer line is busy, ct will continue to try to open the dialer at one-minute in-

tervals for a total of 15 minutes (as set by the -w flag option).

FILES

/bin/ct
/usr/lib/uucp/L-devices
/usr/adm/ctlog

SEE ALSO

cu(1C), login(1), uucp(1C).

NAME

ctags — maintain a tags file for a C program

SYNOPSIS

ctags [-a] [-u] [-w] [-x] *name* ...

DESCRIPTION

ctags makes a tags file for ex(1) and vi(1) from the specified C, Fortran, and Pascal sources.

A tags file gives the locations of specified objects (in this case functions) in a group of files. Each line of the tags file contains the function name, the file in which it is defined, and a scanning pattern used to find the function definition. These are given in separate fields on the line, separated by blanks or tabs. Using the tags file, ex can quickly find these function definitions.

FLAG OPTIONS

-a Causes the output to be appended to the tags file instead of rewriting it.

-u Causes the specified files to be *updated* in tags, that is, all references to them are replaced by new values. (Beware: this flag option is implemented in a way which is rather slow; it is usually faster to simply rebuild the *tags* file.)

-w Suppresses warning diagnostics.

-x If the -x flag option is given, ctags produces a list of function names, the line number and file name on which each is defined, as well as the text of that line and prints this on the standard output.

Files whose name ends in .c or .h are assumed to be C source files and are searched for C routine and macro definitions.

The tag *main* is treated specially in C programs. The tag formed is created by prefixing M to the name of the file, with a trailing .c removed, if any, and leading pathname components also removed. This makes use of ctags practical in directories with more than one program.

EXAMPLES

```
ctags *.c *.h
```

puts the tags from all the .c and .h files into the tagsfile tags.

FILES

`/usr/bin/ctags`	
`tags`	output tags file

SEE ALSO

`ex`(1), `vi`(1).

BUGS

Not all warning diagnostics are suppressed by `-w`.

If `ctags`(1) is interrupted while executing under the `-u` flag option, a temporary file named `OTAGS` is left in the current directory.

NAME

ctrace — C program debugger

SYNOPSIS

ctrace [-b] [-e] [-f*functions*] [-l*n*] [-o] [-p *'s'*] [-P] [-r*f*] [-s] [-t*n*] [-u] [-v*functions*] [-x] [*file*]

DESCRIPTION

ctrace allows you to follow the execution of a C program, statement by statement. The effect is similar to executing a shell procedure with the -x flag option. ctrace reads the C program in *file* (or from standard input if you do not specify *file*), inserts statements to print the text of each executable statement and the values of all variables referenced or modified, and writes the modified program to the standard output. You must put the output of ctrace into a temporary file because the cc(1) command does not allow the use of a pipe. You then compile and execute this file.

As each statement in the program executes, it will be listed at the terminal, followed by the name and value of any variables referenced or modified in the statement, followed by any output from the statement. Loops in the trace output are detected and tracing is stopped until the loop is exited or a different sequence of statements within the loop is executed. A warning message is printed every 1000 times through the loop to help you detect infinite loops. The trace output goes to the standard output, so that you may put it into a file for examination with an editor or the bfs(1) or tail(1) commands.

The only flag options you will commonly use are:

-f *functions* Trace only these functions.

-v *functions* Trace all but these functions.

You may want to print variables in other formats besides the default. Long and pointer variables are always printed as signed integers. Pointers to character arrays are also printed as strings if appropriate. char, short, and int variables are also printed as signed integers and, if appropriate, as characters. Double variables are printed as floating point numbers in scientific notation. You may request that variables be printed in additional formats, if appropriate, with these flag options:

-o Octal
-x Hexadecimal
-u Unsigned
-e Floating point

These flag options are used only in special circumstances:

-l *n* Check *n* consecutively executed statements for looping trace output, instead of the default of 20. Use 0 to get all the trace output from loops.

-s Suppress redundant trace output from simple assignment statements and string copy function calls. This flag option can hide a bug caused by use of the = operator in place of the == operator.

-t *n* Trace *n* variables per statement instead of the default of 10 (the maximum number is 20). The DIAGNOSTICS section explains when to use this flag option.

-P Run the C preprocessor on the input before tracing it. You can also use the -D, -I, and -U cc(1) preprocessor flag options.

These flag options are used to tailor the run-time trace package when the traced program will run in another environment than that of the UNIX system:

-b Use only basic functions in the trace code, that is, those in ctype(3C), printf(3S), and string(3C). These are usually available even in cross-compilers for microprocessors. In particular, this flag option is needed when the traced program runs under an operating system that does not have signal(3), fflush(3S), longjmp(3C), or setjmp(3C).

-p *'s'*
Change the trace print function from the default of 'printf('. For example, 'fprintf(stderr' would send the trace to the standard error output.

-r *f* Use file *f* in place of the runtime.c trace function package. This lets you change the entire print function, instead of just the name and leading arguments (see the -p flag option).

EXAMPLES

If the file `lc.c` contains this C program:

```
 1 #include <stdio.h>
 2 main()   /* count lines in input */
 3 {
 4    int c, nl;
 5
 6    nl = 0;
 7    while ((c = getchar()) != EOF)
 8          if (c = '\n')
 9                ++nl;
10    printf("%d\n", nl);
11 }
```

and you enter these commands and test data:

```
cc lc.c
a.out
1
```

(CONTROL-d),

the program will be compiled and executed. The output of the program will be the number 2, which is not correct because there is only one line in the test data. The error in this program is common, but subtle. If you invoke `ctrace` with these commands:

```
ctrace lc.c > temp.c
cc temp.c
a.out
```

the output will be:

```
 2 main()
 6    nl = 0;
      /* nl == 0 */
 7    while ((c = getchar()) != EOF)
```

The program is now waiting for input. If you enter the same test data as before, the output will be:

```
      /* c == 49 or '1' */
 8          if (c = '\n')
            /* c == 10 or '\n' */
 9                ++nl;
```

```
                    /* nl == 1 */
 7      while ((c = getchar()) != EOF)
        /* c == 10 or '\n' */
 8            if (c = '\n')
              /* c == 10 or '\n' */
 9                  ++nl;
                    /* nl == 2 */

 7                  /* repeating */
```

If you now enter an end-of-file character (CONTROL-d), the final output will be:

```
        /* c == -1 */
10      printf("%d\n", nl);
        /* nl == 2 */2
        /* return */
```

Note that the program output printed at the end of the trace line for the `nl` variable. Also note the `return` comment added by `ctrace` at the end of the trace output. This shows the implicit return at the terminating brace in the function.

The trace output shows that variable `c` is assigned the value "`1`" in line 7, but in line 8 it has the value "`\n`". Once your attention is drawn to this `if` statement, you will probably realize that you used the assignment operator (=) in place of the equal operator (==). During code reading, it is easy to miss this error.

EXECUTION-TIME TRACE CONTROL

The default operation for `ctrace` is to trace the entire program file, unless you use the `-f` or `-v` flag options to trace specific functions. This does not give you statement by statement control of the tracing, nor does it let you turn the tracing off and on when executing the traced program.

You can do both of these by adding `ctroff()` and `ctron()` function calls to your program to turn the tracing off and on, respectively, at execution time. Thus, you can code arbitrarily complex criteria for trace control with `if` statements, and you can even conditionally include this code because `ctrace` defines the CTRACE preprocessor variable. For example:

```
#ifdef CTRACE
```

```
        if (c == '!' && i > 1000)
                ctron();
#endif
```

You can also call these functions from sdb(1) if you compile with the -g flag option. For example, to trace all but lines 7 to 10 in the main function, enter:

```
sdb a.out
main:7b ctroff()
main:11b ctron()
r
```

You can also turn the trace off and on by setting the static variable tr_ct_ to 0 and 1, respectively. This is useful if you are using a debugger that cannot call these functions directly.

DIAGNOSTICS

This section contains diagnostic messages from both ctrace and cc(1), since the traced code often gets some cc warning messages. You can get cc error messages in some rare cases, all of which can be avoided.

ctrace Diagnostics

`Warning: some variables are not traced in this statement`

Only 10 variables are traced in a statement to prevent the C compiler "out of tree space; simplify expression" error. Use the -t flag option to increase this number.

`Warning: statement too long to trace`

This statement is over 400 characters long. Make sure that you are using tabs to indent your code, not spaces.

`Cannot handle preprocessor code, use -P option`

This is usually caused by #ifdef/#endif preprocessor statements in the middle of a C statement, or by a semicolon at the end of a #define preprocessor statement.

`''if ... else if'' sequence too long`

Split the sequence by removing an else from the middle.

`Possible syntax error, try -P option`

Use the -P flag option to preprocess the ctrace input, along with any appropriate -D, -I, and -U preprocessor flag options. If you still get the error message, check the

WARNINGS section, below.

cc Diagnostics

```
Warning: floating point not implemented
Warning: illegal combination of pointer and
integer
Warning: statement not reached
Warning: sizeof returns 0
```

Ignore these messages.

`Compiler takes size of function`
: See the `ctrace ''possible syntax error''` message above.

`yacc stack overflow`
: See the `ctrace 'if ... else if' sequence too long` message, above.

`Out of tree space; simplify expression`
: Use the `-t` flag option to reduce the number of traced variables per statement from the default of 10. Ignore the "`ctrace: too many variables to trace`" warnings you will now get.

`redeclaration of signal`
: Either correct this declaration of `signal`(3), or remove it and #include <signal.h>.

WARNINGS

You will get a `ctrace` syntax error if you omit the semicolon at the end of the last element declaration in a structure or union, just before the right brace (}). This is optional in some C compilers.

Defining a function with the same name as a system function may cause a syntax error if the number of arguments is changed. To fix this, just use a different name.

`ctrace` assumes that `BADMAG` is a preprocessor macro, and that `EOF` and `NULL` are `#define` d constants. Declaring any of these to be variables, e.g., `int EOF` will cause a syntax error.

BUGS

`ctrace` does not know about the components of aggregates like structures, unions, and arrays. It cannot choose a format to print all the components of an aggregate when an assignment is made to the entire aggregate. `ctrace` may choose to print the address of an aggregate or use the wrong format (e.g., `%e` for a structure with two integer members) when printing the value of an aggre-

gate.

Pointer values are always treated as pointers to character strings.

The loop trace output elimination is done separately for each file of a multi-file program. This can result in functions called from a loop still being traced, or the elimination of trace output from one function in a file until another in the same file is called.

FILES

```
/usr/bin/ctrace
/usr/lib/ctrace
```

SEE ALSO

sdb(1), ctype(3C), fflush(3S), longjmp(3C), printf(3S), setjmp(3C), signal(3), string(3C).

NAME

cu — call another system

SYNOPSIS

cu [-d] [-e] [-h] [-l*line*] [-m] [-n] [-o] [-s*speed*] [-t] *args*

DESCRIPTION

cu calls up another system and manages an interactive conversation with possible transfers of ASCII files.

cu accepts the following flag options and arguments.

-s*speed*
: Specifies the transmission *speed* (110, 150, 300, 600, 1200, 4800, 9600); 300 being the default value. Most modems are either 300 or 1200 baud. Directly connected lines may be set to a speed higher than 1200 baud.

-l*line*
: Specifies a device name to use as the communication *line*. This can be used to override the search for the first available line having the right speed. When the -l flag option is used without the -s flag option, the speed of a line is taken from the file /usr/lib/uucp/L-devices. When the -l and -s flag options are used simultaneously, cu will search the L-devices file to check if the requested speed for the requested line is available. If so, the connection will be made at the requested speed; otherwise, an error message will be printed and the call will not be made. The specified device is generally a directly-connected asynchronous line (for example, /dev/ttyab), in which case, a telephone number is not required; however, the string dir may be use to specify a null ACU. If the specified device is associated with an auto dialer, a telephone number must be provided.

-h Emulates local echo, supporting calls to other computer systems which expect terminals to be set to half-duplex mode.

-t Used when dialing an ASCII terminal which has been set to "auto answer." Appropriate mapping of return to return-line feed pairs is set.

-d Causes diagnostic traces to be printed.

-e Designates that even parity is to be generated for data sent to the remote.

`-o` Designates that odd parity is to be generated for data sent to the remote.

`-m` Designates a direct line which has modem control.

`-n` Will request the telephone number to be dialed from the user rather than taking it from the command line.

args may be one of the following:

telno
When using an automatic dialer, the argument becomes the telephone number with equal signs at appropriate places for secondary dial tone or with minus signs for delays.

systemname
A `uucp` system name may be used rather than a telephone number; in this case, `cu` will obtain an appropriate direct line or telephone number from `/usr/lib/uucp/L.sys` (the appropriate baud is also read along with telephone numbers). `cu` will try each telephone number or direct line for *systemname* in the `L.sys` file until a connection is made or all the entries are tried.

`dir`
Using `dir` ensures that `cu` will use the line specified by the `-l` flag option.

After making the connection, `cu` runs as two processes: the *transmit* process reads data from the standard input and, except for lines beginning with ~, passes it to the remote system; the *receive* process accepts data from the remote system and, except for lines beginning with ~, passes it to the standard output. Normally, an automatic DC3/DC1 protocol is used to control input from the remote so the buffer is not overrun. Lines beginning with ~ have special meanings.

The *transmit* process interprets the following:

~.	Terminate the conversation.
~!	Escape to an interactive shell on the local system.
~!*cmd*	Run *cmd* on the local system (via `sh -c`).
~$*cmd*	Run *cmd* locally and send its output to the remote system.

`~%cd` Change the directory on the local system. Note that `~!cd` will cause the command to be run by a sub-shell, which is probably not what was intended.

`~%take` *from* [*to*]
Copy file *from* (on the remote system) to the file *to* on the local system. If *to* is omitted, the *from* argument is used in both places.

`~%put` *from* [*to*]
Copy file *from* (on local system) to the file *to* on remote system. If *to* is omitted, the *from* argument is used in both places.

`~~`*cmd* Send the line `~`*cmd* to the remote system.

`~%break` Transmit a BREAK to the remote system.

`~%nostop` Toggles between DC3/DC1 input control protocol and no input control. This is useful in case the remote system is one which does not respond properly to the DC3 and DC1 characters.

The *receive* process normally copies data from the remote system to its standard output. A line from the remote that begins with `~>` initiates an output diversion to a file. The complete sequence is

`~>:`*file*
zero or more lines to be written to *file*
`~>`

Data from the remote are diverted (or appended, if >> is used) to *file*. The trailing `~>` terminates the diversion.

The use of `~%put` requires `stty`(1) and `cat`(1) on the remote side. It also requires that the current erase and kill characters on the remote system be identical to the current ones on the local system. Backslashes are inserted at appropriate places.

The use of `~%take` requires the existence of `echo`(1) and `cat`(1) on the remote system. Also, `stty tabs` mode should be set on the remote system if tabs are to be copied without expansion.

When `cu` is used on system X to connect to system Y and subsequently used on system Y to connect to system Z, commands on system Y can be executed by using `~~`. For example, `uname` may

be executed on Z, X, and Y as follows:

```
uname
Z
~!uname
X
~~!uname
Y
```

In general, ~ causes the command to be executed on the original machine, ~~ causes the command to be executed on the next machine in the chain.

EXAMPLES

In the following examples, assume that you wish to connect to a system named `plato` through the serial line `/dev/tty0`. To dial a system whose number is 9 201 555 9999 using 1200 baud

```
cu -s1200 9=2015559999
```

If the speed is not specified, 300 is the default value.

To login to a system connected by a direct line

```
cu -l /dev/tty0 dir
```

To dial a system with the specific line and a specific speed

```
cu -s1200 -l /dev/tty0 dir
```

To dial a system using a specific line

```
cu -l /dev/cul0 2015559999
```

To use a system name

```
cu plato
```

FILES

```
/usr/bin/cu
/usr/lib/uucp/L.sys
/usr/lib/uucp/L-devices
/usr/spool/uucp/LCK..tty-device
/dev/null
```

SEE ALSO

`cat`(1), `ct`(1C), `echo`(1), `ftp`(1N), `stty`(1), `telnet`(1N), `tip`(1C), `uname`(1), `uucp`(1C).
"Using `cu`" in *A/UX Communications User's Guide.*
"UNIX-to-UNIX Communcations Package: `uucp`" in *A/UX Local System Administration.*

DIAGNOSTICS

Exit code is zero for normal exit, nonzero (various values) otherwise.

BUGS

`cu` buffers input internally.

There is an artificial slowing of transmission by `cu` during the `~%put` operation so that loss of data is unlikely.

NOTES

Any input character after a ~ will be preceded by [*sysname*] (inserted by `cu`).

NAME

cut — cut out selected fields of each line of a file

SYNOPSIS

cut *-type* [-d *char*] [-s] [*file*]...

DESCRIPTION

Use cut to cut out columns from a table or fields from each line of a file; in database parlance, it implements the projection of a relation. cut may be used as a filter; if no files are given, the standard input is used.

type may be either c or f, followed by *list*. The fields as specified by *list* may be fixed length, i.e., character positions as on a punched card (-c flag option) or the length may vary from line to line and be marked with a field delimiter character like TAB (-f flag option). Note that *-type* (i.e., either the -c or -f flag option) must be specified.

The meanings of the flag options are:

list A comma-separated list of integer field numbers (in increasing order), with optional - to indicate ranges as in the -o flag option of nroff/troff for page ranges; e.g., 1,4,7; 1-3,8; -5,10 (short for 1-5,10); or 3- (short for third through last field).

-c*list* The *list* following -c (no space) specifies character positions (e.g., -c1-72 would pass the first 72 characters of each line).

-f*list* The *list* following -f is a list of fields assumed to be separated in the file by a delimiter character (see -d); e.g., -f1,7 copies the first and seventh field only. Lines with no field delimiters will be passed through intact (useful for table subheadings), unless -s is specified.

-d*char* The character following -d is the field delimiter (-f flag option only). Default is tab. Space or other characters with special meaning to the shell must be quoted.

-s Suppresses lines with no delimiter characters in case of -f flag option. Unless specified, lines with no delimiters will be passed through untouched.

HINTS

Use grep(1) to make horizontal cuts (by context) through a file, or paste(1) to put files together column-wise (i.e., horizontally). To reorder columns in a table, use cut and paste.

EXAMPLES

```
cut -d: -f1,5 /etc/passwd
```

mapping of user IDs to names.

```
name=`who am i | cut -f1 -d" "`
```

to set *name* to current login name.

DIAGNOSTICS

`Line too long`
: A line can have no more than 1023 characters or fields.

`Bad list for c/f flag option`
: Missing -c or -f flag option or incorrectly specified *list*. No error occurs if a line has fewer fields than the *list* calls for.

`No fields`
: The *list* is empty.

FILES

/usr/bin/cut

SEE ALSO

awk(1), colrm(1), grep(1), paste(1), sed(1).

NAME

cw, checkcw — prepare constant-width text for otroff

SYNOPSIS

cw [-d] [-f*n*] [-l*xx*] [-r*xx*] [-t] [+t] [*files...*]

checkcw [-l*xx*] [-r*xx*] *file...*

DESCRIPTION

cw is a preprocessor for otroff(1)) input files that contain text to be typeset in the constant-width (CW) font.

Text typeset with the CW font resembles the output of terminals and of line printers. This font is used to typeset examples of programs and of computer output in user manuals, programming texts, etc. (An earlier version of this font was used in typesetting *The C Programming Language* by B. W. Kernighan and D. M. Ritchie.) It has been designed to be quite distinctive (but not overly obtrusive) when used together with the Times Roman font.

Because the CW font contains a *nonstandard* set of characters and because text typeset with it requires different character and interword spacing than is used for *standard* fonts, documents that use the CW font must be preprocessed by cw.

The CW font contains the 94 printing ASCII characters:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
!$%&()'′*+@.,/:;=?[]|-_^~"<>{}#\
```

plus eight non-ASCII characters represented by four-character otroff(1) names (in some cases attaching these names to nonstandard graphics):

character	symbol	troff name
Cents sign	¢	\(ct
EBCDIC *not* sign	¬	\(no
Left arrow	←	\(<-
Right arrow	→	\(->
Down arrow	↓	\(da
Vertical single quote	′	\(fm
Control-shift indicator	†	\(dg
Visible space indicator	□	\(sq

Hyphen - \(hy

The hyphen is a synonym for the unadorned minus sign (−). Certain versions of cw recognize two additional names: \(ua for an up arrow and \(lh for a diagonal left-up (home) arrow.

cw recognizes five request lines, as well as user-defined delimiters. The request lines look like otroff(1) macro requests, and are copied in their entirety by cw onto its output; thus, they can be defined *by the user* as otroff(1) macros; in fact, the .CW and .CN macros *should* be so defined (see HINTS below). The five requests are:

.CW Start of text to be set in the CW font; .CW causes a break; it can take precisely the same flag options, in precisely the same format, as are available on the cw command line.

.CN End of text to be set in the CW font; .CN causes a break; it can take the same flag options as are available on the cw command line.

.CD Change delimiters and/or settings of other flag options; takes the same flag options as are available on the cw command line.

.CP *arg1 arg2 arg3* ... argn
All the arguments which are delimited like otroff(1) macro arguments) are concatenated, with the odd-numbered arguments set in the CW font and the even-numbered ones in the prevailing font.

.PC *arg1 arg2 arg3* ... argn
Same as .CP, except that the even-numbered arguments are set in the CW font and the odd-numbered ones in the prevailing font.

The .CW and .CN requests are meant to bracket text (e.g., a program fragment) that is to be typeset in the CW font as is. Normally, cw operates in the *transparent* mode. In that mode, except for the .CD request and the nine special four-character names listed in the table above, every character between .CW and .CN request lines stands for itself. In particular, cw arranges for periods (.) and apostrophes (') at the beginning of lines, and backslashes (\) everywhere to be hidden from otroff(1). The transparent mode can be turned off (see below), in which case normal otroff(1) rules apply; in particular, lines that begin with . and ' are passed through untouched (except if they contain delimiters–see below).

In either case, cw hides the effect of the font changes generated by the .CW and .CN requests; cw also defeats all ligatures (fi, ff, etc.) in the CW font.

The only purpose of the .CD request is to allow the changing of various flag options other than just at the beginning of a document.

The user can also define *delimiters*. The left and right delimiters perform the same function as the .CW/.CN requests; they are meant, however, to enclose CW words or phrases in running text (see example under BUGS below). cw treats text between delimiters in the same manner as text enclosed by .CW/.CN pairs, except that, for aesthetic reasons, spaces and backspaces inside .CW/.CN pairs have the same width as other CW characters, while spaces and backspaces between delimiters are half as wide, so they have the same width as spaces in the prevailing text (but are *not* adjustable). Font changes due to delimiters are *not* hidden.

Delimiters have no special meaning inside .CW/.CN pairs.

The flag options are:

-l*xx* The one- or two-character string *xx* becomes the left delimiter; if *xx* is omitted, the left delimiter becomes undefined, which it is initially.

-r*xx* Same for the right delimiter. The left and right delimiters may (but need not) be different.

-f*n* The CW font is mounted in font position *n*; acceptable values for *n* are 1, 2, and 3 (default is 3, replacing the bold font). This flag option is only useful at the beginning of a document.

-t Turn transparent mode *off*.

+t Turn transparent mode *on* (this is the initial default).

-d Print current flag option settings on file descriptor 2 in the form of otroff(1) comment lines. This flag option is meant for debugging.

cw reads the standard input when no *files* are specified (or when - is specified as the last argument), so it can be used as a filter. Typical usage is:

```
cw files | otroff ...
```

checkcw checks that left and right delimiters, as well as the .CW/.CN pairs, are properly balanced. It prints out all offending

lines.

HINTS

Typical definitions of the .CW and .CN macros meant to be used with the mm(1) macro package:

```
.de CW
.DS I
.ps 9
.vs 10.5p
.ta 16m/3u 32m/3u 48m/3u 64m/3u 80m/3u 96m/3u ...
..
.de CN
.ta 0.5i 1i 1.5i 2i 2.5i 3i 3.5i 4i 4.5i 5i 5.5i 6i
.vs
.ps
.DE
..
```

At the very least, the .CW macro should invoke the otroff(1) no-fill (.nf) mode.

When set in running text, the CW font is meant to be set in the same point size as the rest of the text. In displayed matter, on the other hand, it can often be profitably set one point *smaller* than the prevailing point size (the displayed definitions of .CW and .CN above are one point smaller than the running text on this page). The CW font is sized so that, when it is set in 9-point, there are 12 characters per inch.

Documents that contain CW text may also contain tables and/or equations. If this is the case, the order of preprocessing should be: cw, tbl, and eqn. Usually, the tables contained in such documents will not contain any CW text, although it is entirely possible to have *elements* of the table set in the CW font; of course, care must be taken that tbl(1) format information not be modified by cw. Attempts to set equations in the CW font are not likely to be either pleasing or successful.

In the CW font, overstriking is most easily accomplished with backspaces: letting ← represent a backspace, d←←† yields d‡. Because spaces (and, therefore backspaces) are half as wide between delimiters as inside .CW/.CN pairs (see above), two backspaces are required for each overstrike between delimiters.

EXAMPLES

```
cw text | tbl | otroff -mm
```

processes the text file `text`, sends the output to `tbl`(1) and then sends the output for final formatting to `otroff`(1) and `mm`(1).

FILES

`/bin/cw`
`/usr/lib/font/ftCW`

SEE ALSO

`eqn`(1), `mmt`(1), `tbl`(1), `troff`(1), `mm`(5), `mv`(5),
"Other Text Processing Tools" in *A/UX Programming Languages and Tools, Volume 2*.

WARNINGS

If text preprocessed by `cw` is to make any sense, it must be set on a typesetter equipped with the CW font or on a STARE facility; on the latter, the CW font appears as bold, but with the proper CW spacing.

Do not use periods (.), backslashes (\), or double quotes (") as delimiters, or as arguments to `.CP` and `.PC`.
Do not use `cw` with `nroff`, since `nroff` already makes everything constant-width.

BUGS

Certain CW characters don't concatenate gracefully with certain Roman characters, for example, a CW ampersand (`&`) followed by a Roman comma (`,`). in such cases, judicious use of `otroff`(1) half- and quarter-spaces (`\|` and `\^`) is most salutary, for example, one should use `_&_\^` (rather than just plain `_&_`) to obtain `&` (assuming that `_` is used for both delimiters).
The output of `cw` is hard to read. See also BUGS under `otroff`(1).

NAME

cxref — generate C program cross-reference

SYNOPSIS

cxref [-c] [-o *file*] [-s] [-t] [-w[*num*]] *file*...

DESCRIPTION

cxref analyzes a collection of C files and attempts to build a cross-reference table. cxref utilizes a special version of cpp to include information from #define statements in its symbol table. It produces a separate listing on standard output of all symbols (auto, static, and global) in each file, or with the -c flag option, of all symbols in combination. Each symbol contains an asterisk (*) before the declaring reference.

In addition to the -D, -I, and -U flag options (which are identical to the corresponding flag options in cc(1)), the following flag options are interpreted by cxref:

-c	Prints a combined cross-reference of all input files.
-w[*num*]	Width flag option which formats output no wider than *num* (decimal) columns. This flag option will default to 80 if *num* is not specified or is less than 51.
-o *file*	Directs output to named *file*.
-s	Operates silently; does not print input filenames.
-t	Formats listing for 80-column width.

FILES

/usr/bin/cxref	
/usr/lib/xpass	input file parser
/usr/lib/xcpp	special version of C-preprocessor.

SEE ALSO

cc(1).

DIAGNOSTICS

Error messages are unusually cryptic, but usually mean that you can't compile these files anyway.

BUGS

cxref considers a formal argument in a #define macro definition to be a declaration of that symbol. For example, a program that contains the line

```
#include <ctype.h>
```

will contain many declarations of the variable `c`.

When using the `-o` option, the space between the `-o` and the *file* argument is critical. If you omit the space, as in

```
cxref -otest test.c
```

then the input file `test.c` will be destroyed.

NAME

daiw — Apple ImageWriter II troff postprocessor filter

SYNOPSIS

daiw [-v] [-r*num*] *file*

DESCRIPTION

daiw translates a *file* created by troff(1) to a bitmap image suitable for printing on the Apple ImageWriter® II printer. If no *file* is mentioned, the standard input is used. The bitmap is sent to the standard output. The following options are understood.

-v	A *verbose* option. Warning messages (for example, font or point size substitutions) are printed on the standard error output. By default, warning messages are suppressed.
-r*num*	Sets printing resolution to either 72 dots-per-inch or 144 dots-per-inch.

When processing input files with pic or eqn, you should specify the -Tiw flag option; this option is not required with tbl.

EXAMPLES

The following command line will format the file ch.1 with the appropriate pre- and post-processors and spool the output for printing on an ImageWriter II:

```
pic -Tiw ch.1 | troff -Tiw | daiw | lp
```

FILES

/usr/bin/daiw	
/usr/lib/font/deviw/*	troff description files for the ImageWriter II.

SEE ALSO

troff(1).
Inside Macintosh, Addison Wesley, 1985.
Apple ImageWriter II Technical Reference Manual.
A/UX Text Processing Tools.

NAME

daps — Autologic APS-5 phototypesetter troff postprocessor

SYNOPSIS

daps [-b] [-h*string*] [-o*list*] [-r] [-s*n*] [-t] [-w] [*file...*]

DESCRIPTION

daps prints *files* created by troff(1) on an Autologic APS-5 phototypesetter. If you do not specify a *file*, the standard input is printed. daps understands the following options:

-b	Report whether the typesetter is busy; does not print output
-h*string*	Prints *string* in this job's header. The header appears on a page preceding the output.
-o*list*	Print pages whose numbers are given in the *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.
-r	Reports the number of 11-inch pages generated by this job.
-s*n*	Stop after every *n* pages of output. daps continues when you push the PROCEED button on the typesetter.
-t	Directs output to the standard output instead of the typesetter.
-w	Waits for typesetter to become free, then prints output.

The *files* that you submit to daps should be prepared under the -Taps flag option of troff.

FILES

/usr/bin/daps	
/dev/aps	APS-5 phototypesetter device
/usr/lib/font/devaps/*	description files for APS-5

SEE ALSO

grap(1), mmt(1), mvt(1), pic(1), tc(1), troff(1).

BUGS

Installations with an Autologic APS-5 phototypesetter should be aware that getting a good match to their Autologic fonts will almost certainly require hand-tuning of the distributed font descrip-

tion files (see FILES above).

NAME

`date` — display and set the date

SYNOPSIS

`date` [*mmddhhmm*[yy]] [+*format*]

DESCRIPTION

If no argument is given, or if the argument begins with +, the current date and time are displayed. Otherwise, the current date is set. The first *mm* is the month number; *dd* is the day number in the month; *hh* is the hour number (24 hour system); the second *mm* is the minute number; and *yy* is the last 2 digits of the year number and is optional. If a number less than 70 is given, the year that results is 1970. For example,

```
date 10080045
```

sets the date to Oct. 8, 12:45 AM. The current year is the default if no year is mentioned. The operating system operates in GMT. `date` takes care of the conversion to and from local standard and daylight time.

If the argument begins with +, the output of `date` is under the control of the user. The format for the output is similar to that of the first argument to `printf`(3S). All output fields are of fixed size (zero padded if necessary). Each field descriptor is preceded by `%` and will be replaced in the output by its corresponding value. A single `%` is encoded by `%%`. All other characters are copied to the output without change. The string is always terminated with a newline character.

Field Descriptors

`n`	insert a newline character
`t`	insert a tab character
`m`	month of year—01 to 12
`d`	day of month—01 to 31
`y`	last 2 digits of year—70 to 99
`D`	date as mm/dd/yy
`H`	hour—00 to 23
`M`	minute—00 to 59
`S`	second—00 to 59
`T`	time as HH:MM:SS
`j`	day of year—001 to 366
`w`	day of week—Sunday = 0
`a`	abbreviated weekday—Sun to Sat

h abbreviated month—Jan to Dec
r time in AM/PM notation

EXAMPLES

The command

```
date '+DATE: %m/%d/%y%nTIME: %H:%M:%S'
```

generates as output

```
DATE: 08/01/76
TIME: 14:45:05
```

DIAGNOSTICS

`No permission` if you are not the superuser and you try to change the date;

`bad conversion` if the date set is syntactically incorrect

`bad format character` if the field descriptor is not recognizable

FILES

`/bin/date`
`/etc/wtmp` to record time

SEE ALSO

`gettimeofday`(2), `stime`(2), `time`(2), `printf`(3S), `utmp`(4).

WARNINGS

It is bad practice to change the date while the system is running multiuser.

NAME

dc — desk calculator

SYNOPSIS

dc [*file*]

DESCRIPTION

dc is an arbitrary precision arithmetic package. Ordinarily it operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained. The overall structure of dc is a stacking (reverse Polish) calculator. If an argument is given, input is taken from that file until its end, then from the standard input. The following constructions are recognized:

number	Pushes the value of *number* on the stack. A number is an unbroken string of one or more digits in the range 0–9. It may be preceded by an underscore (_) to indicate a negative number. Numbers may contain decimal points.
+ – / * % ^	Operate on the top two values on the stack. These are added (+), subtracted (–), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. Any fractional part of an exponent is ignored.
s*x*	Pops the top of the stack and stores it into a register named *x*, where *x* may be any character.
S*x*	Pushes the value on *x*, which is treated as a stack.
l*x*	Pushes the value in register *x* on the stack. The register *x* is not altered. All registers start with zero value.
L*x*	Pops the top value of register *x*, which is treated as a stack, onto the main stack.
d	Duplicates the top value on the stack.
p	Prints the top value on the stack. The top value remains unchanged.
P	Interprets the top of the stack as an ASCII string, removes it, and prints it.

`f`	Prints all values on the stack.
`q`	Exits the program. If executing a string, the recursion level is popped by two. Alternately, CONTROL-d (EOF) will exit from `dc`.
`Q`	Pops the top value on the stack and pops the string execution level by that value. Alternately, CONTROL-d (EOF) will exit from `dc`.
`x`	Treats the top element of the stack as a character string and executes it as a string of `dc` commands.
`X`	Replaces the number on the top of the stack with its scale factor.
`[`*string*`]`	Puts the bracketed ASCII string onto the top of the stack.
<x *>x* *=x*	Pops the top two elements of the stack and compares them. Register *x* is evaluated if they obey the stated relation.
`v`	Replaces the top element on the stack by its square root. Any existing fractional part of the argument is taken into account, but otherwise the scale factor is ignored.
`!`	Interprets the rest of the line as a system command.
`c`	Pops all values on the stack.
`i`	Pops the top value on the stack and uses it as the number radix for further input.
`I`	Pushes the input base on the top of the stack.
`o`	Pops the top value on the stack and uses it as the number radix for further output.
`O`	Pushes the output base on the top of the stack.
`k`	Pops the top of the stack and uses that value as a non-negative scale factor: prints the appropriate number of places on output, and maintains them during multiplication, division, and exponentiation. The interaction of scale factor, input base, and output base will be reasonable if all are changed together.

`z`	Pushes the stack level onto the stack.
`Z`	Replaces the number on the top of the stack with its length.
`?`	Takes a line of input from the input source (usually the terminal) and executes it.
`; :`	Allow `bc` to perform array operations.

EXAMPLES

```
dc
24.2 56.2 + p
```

adds the two numbers and prints the result (top value in the stack).

```
[la1+dsa*pla10>y]sy
0sa1
lyx
```

prints the first ten values of *n*!.

FILES

`/usr/bin/dc`

SEE ALSO

`bc`(1).
"`dc` Reference" in *A/UX Programming Languages and Tools, Volume 2*.

DIAGNOSTICS

`x is unimplemented`	where *x* is an octal number.
`stack empty`	for not enough elements on the stack to do what was asked.
`Out of space`	when the free list is exhausted (too many digits).
`Out of headers`	for too many numbers being kept around.
`Out of pushdown`	for too many items on the stack.
`Nesting Depth`	for too many levels of nested execution.

NAME

dd — convert and copy a file

SYNOPSIS

dd [*option=value*]...

DESCRIPTION

dd copies the specified input file to the specified output with possible conversions. The standard input and output are used by default. The input and output block size may be specified to take advantage of raw physical I/O.

Option	Values
if=*file*	input filename; standard input is default
of=*file*	output filename; standard output is default
ibs=*n*	input block size *n* bytes (default 512)
obs=*n*	output block size (default 512)
bs=*n*	set both input and output block size, superseding ibs and obs; also, if no conversion is specified, it is particularly efficient since no incore copy needs to be generated
cbs=*n*	conversion buffer size
skip=*n*	skip *n* input blocks before starting copy
seek=*n*	seek *n* blocks from beginning of output file before copying; dd creates the specified output file (see creat(2)), which insures that the length of the file will be zero for regular files; seeking *n* blocks from the beginning of the output file will fill the skipped area with zeros (nulls)
count=*n*	copy only *n* input blocks
conv=ascii	convert EBCDIC to ASCII
ebcdic	convert ASCII to EBCDIC
ibm	slightly different map of ASCII to EBCDIC
lcase	map alphabetics to lowercase
ucase	map alphabetics to uppercase
swab	swap every pair of bytes
noerror	do not stop processing on an error

`sync` pad every input block to `ibs`

type`,`*type* several comma-separated conversions, where *type* is one of the conversions listed for `conv`

`multi=in` input file is multivolume

`out` output file is multivolume

`in,out` both the input file and output file are multivolume

Where sizes are specified, a number of bytes is expected. A number may end with `k`, `b`, or `w` to specify multiplication by 1024, 512, or 2, respectively; a pair of numbers may be separated by `x` to indicate a product.

`cbs` is used only if `ascii`, `ebcdic`, or `ibm` conversion is specified. In the first case, `cbs` characters are placed into the conversion buffer, converted to ASCII, and trailing blanks are trimmed and a newline added before sending the line to the output. In the next two cases, ASCII characters are read into the conversion buffer, converted to EBCDIC (or the IBM version of EBCDIC), and blanks are added to make up an output block of size `cbs`.

If multivolume input (output) is specified, a prompt is given at end-of-file to allow another volume to be mounted.

After completion, `dd` reports the number of whole and partial input and output blocks.

EXAMPLES

```
dd if=/dev/rmt/0m of=x ibs=800 cbs=80 conv=ascii,lcase
```

will read an EBCDIC tape blocked at ten 80-byte EBCDIC card images per block into the ASCII file `x`.

Note the use of raw magnetic tape. `dd` is especially suited to I/O on the raw physical devices because it allows reading and writing in arbitrary block sizes.

FILES

`/bin/dd`

SEE ALSO

`cp`(1), `cpio`(1), `tar`(1), `tr`(1).

DIAGNOSTICS

f+p `blocks in(out)`

are numbers of full and partial blocks read (written).

BUGS

The ASCII/EBCDIC conversion tables are taken from the 256-character standard in the *CACM*, November, 1968. The `ibm` conversion, while less blessed as a standard, corresponds better to certain IBM print-train conventions. There is no universal solution.

Newlines are inserted only on conversion to ASCII; padding is done only on conversion to EBCDIC. These should be separate options.

When using `dd` to transfer data over an Ethernet connection, you should specify a block size of 1 kilobyte.

NAME

delta — make a delta (change) to an SCCS file

SYNOPSIS

delta [-g*list*] [-m[*mrlist*]] [-n] [-p] [-r*SID*] [-s] [-y[*comment*]] *file* ...

DESCRIPTION

delta is used to permanently introduce into the named SCCS file changes that were made to the file retrieved by get(1) (called the g-file, or generated file).

delta makes a delta to each named SCCS file. If a directory is named, delta behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with s.) and unreadable files are silently ignored. If a name of – is given, the standard input is read (see WARNINGS); each line of the standard input is taken to be the name of an SCCS file to be processed.

Delta may issue prompts on the standard output depending upon certain keyletters specified and flags (see admin(1)) that may be present in the SCCS file (see -m and -y keyletters below).

Keyletter arguments apply independently to each named file.

-r*SID* Uniquely identifies which delta is to be made to the SCCS file. The use of this keyletter is necessary only if two or more outstanding gets for editing (get -e) on the same SCCS file were done by the same person (login name). The *SID* value specified with the -r keyletter can be either the *SID* specified on the get command line or the *SID* to be made as reported by the get command (see get(1)). A diagnostic results if the specified *SID* is ambiguous, or, if necessary and omitted on the command line.

-s Suppresses the issue on the standard output of the created delta's *SID*, as well as the number of lines inserted, deleted and unchanged in the SCCS file.

-n	Specifies retention of the edited g-file (normally removed at completion of delta processing).
-g*list*	Specifies a *list* (see get(1) for the definition of *list*) of deltas which are to be *ignored* when the file is accessed at the change level (*SID*) created by this delta.
-m[*mrlist*]	If the SCCS file has the v flag set (see admin(1)) then a Modification Request (MR) number *must* be supplied as the reason for creating the new delta. If -m is not used and the standard input is a terminal, the prompt MRs? is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. The MRs? prompt always precedes the comments? prompt (see -y keyletter). MRs in a list are separated by blanks and/or tab characters. An unescaped newline character terminates the MR list. Note that if the v flag has a value (see admin(1)), it is taken to be the name of a program (or shell procedure) which will validate the correctness of the MR numbers. If a nonzero exit status is returned from MR number validation program, delta terminates (it is assumed that the MR numbers were not all valid).
-y[*comment*]	Arbitrary text used to describe the reason for making the delta. A null string is considered a valid *comment*. If the comment includes spaces, you must enclose the entire string in double quotes. If -y is not specified and the standard input is a terminal, the prompt comments? is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued.

An unescaped newline character terminates the comment text.

`-p` Causes `delta` to print (on the standard output) the SCCS file differences before and after the delta is applied in a `diff(1)` format.

EXAMPLES

```
% delta s.test1.c
comments?  second version
1.2
1 inserted
0 deleted
12 unchanged
```

does a `delta` on file `test1.c`.

WARNINGS

Lines beginning with an SOH ASCII character (binary 001) cannot be placed in the SCCS file unless the SOH is escaped. This character has special meaning to SCCS (see `sccsfile(5)`) and will cause an error.

A `get` of many SCCS files, followed by a `delta` of those files, should be avoided when the `get` generates a large amount of data. Instead, multiple `get/delta` sequences should be used.

If the standard input (–) is specified on the `delta` command line, the `-m` (if necessary) and `-y` keyletters *must* also be present. Omission of these keyletters causes an error to occur.

Comments are limited to text strings of at most 512 characters.

FILES

`/usr/bin/delta`

g-file Existed before the execution of `delta`; removed after completion of `delta`.

p-file Existed before the execution of `delta`; may exist after completion of `delta`.

q-file Created during the execution of `delta`; removed after completion of `delta`.

x-file Created during the execution of `delta`; renamed to SCCS file after completion of `delta`.

z-file Created during the execution of `delta`; removed during the execution of `delta`.

d-file	Created during the execution of `delta`; removed after completion of `delta`.
`/usr/bin/bdiff`	Program to compute differences between the "gotten" file and the *g-file*.

Note: All files of the form *?*`-file` are explained in the "SCCS Reference" in *A/UX Programming Languages and Tools, Volume 2*. The naming convention for these files is also described there.

SEE ALSO

`admin`(1), `bdiff`(1), `cdc`(1), `get`(1), `help`(1), `prs`(1), `rmdel`(1), `sccs`(1), `sccsfile`(4).
"SCCS Reference" in *A/UX Programming Languages and Tools, Volume 2*.

DIAGNOSTICS

Use `help`(1) for explanations.

NAME

derez — decompile a resource file

SYNOPSIS

derez [*option*]... *resource-file* [*resource-description-file*]...

DESCRIPTION

derez creates a text representation (*resource description*) of a compiled resource file according to the resource type declarations in the *resource-description-files*.

The *resource-file* parameter specifies the name of the file containing the compiled resources. You must specify a resource file; derez never reads the standard input.

The *resource-description-file* parameter specifies one or more files containing the type declarations used by the resource file. derez can provide meaningful output only if you provide the type declarations.

The type declarations in the resource description file follow the same format as that used by the resource compiler, rez. The type declarations for standard Macintosh resources are contained in the files types.r and systypes.r in the directory /usr/libmac/rincludes. If you do not specify a resource description file, the output consists of data statements giving the resource data in hexadecimal form, without any additional format information.

The resource description is written to standard output. If the output of derez is used as input to rez with the same resource description files, it produces the same resource file that was originally input to derez. derez is not guaranteed to be able to run a declaration backward: if it can't, it produces a data statement instead of the appropriate resource statement.

derez ignores all include (but not #include), read, data, and resource statements found in the *resource-description-file*. (It still parses these statements for correct syntax.) Appendix C in the *A/UX Toolbox: Macintosh ROM Interface*, "Resource Compiler and Decompiler," describes the format of resource type declarations.

The resource description consists of resource and data statements that can be understood by rez.

If no errors or warnings are detected, `derez` runs silently. Errors and warnings are written to standard error (see `intro`(3S) in *A/UX Programmer's Reference*).

`derez` returns one of the following status values.

0 No errors

1 Error in parameters

2 Syntax error in file

3 I/O or program error

You may specify one or more of the following options.

`-c[ompatible]`
Generate output that is backward-compatible with `rez` 1.0.

`-d[efine]`*macro*`[=`*data*`]`
Define the macro variable *macro* to include the value *data*. If *data* is omitted, then *macro* is set to the null string (note that this still means that macro is defined). The `-d` flag option is the same as writing

```
#define macro [data]
```

at the beginning of the input. The `-d` flag option may be repeated any number of times.

`-e[scape]`
Do not escape characters that are normally escaped (such as: `\0xff`). Instead, print these characters as extended Macintosh characters. Note that not all fonts have defined all the characters.

Normally, characters with values between 0x20 and 0xD8 are printed as Macintosh characters. With the `-e` option, however, all characters (except null, newline, tab, backspace, form feed, vertical tab, and rubout) are printed as characters, not as escape sequences.

`-i`*pathname*(s)
Search the specified directories for `include` files. You may specify more than one pathname. The paths are searched in the order they appear on the command line.

To decompile an A/UX Toolbox resource file, use this pathname:

```
/usr/lib/mac/rincludes
```

`-m[axstringsize]` *n*
Set the maximum string size to *n*; *n* must be in the range 2–120. This setting controls how wide the strings can be in the output.

`-o[nly]` *type-expr*`[(`*ID1*`[:`*ID2*`])]`
`-o[nly]` *type-expr*`[`*resourceName*`]`
Read only resources of resource type *type-expr*. If an ID, range of IDs, or resource name is given, read only those resources for the given type. This option may be repeated.

Note: *type-expr* is an expression, so straight quotes (') might be needed. If an ID, range of IDs, or name is given, the entire option parameter must be quoted. For example,

```
derez -only "'MENU'(1:128)" ...
```

See also the EXAMPLES later in this section.

Note: The `-only` flag option cannot be specified together with the `-skip` flag option.

`-o[nly]` *type*
Read only resources of the specified type. This is a simpler version of the above option. No quotes are needed to specify a literal type as long as it starts with a letter. Do not use escape characters or other special characters. For example,

```
derez -only MENU ...
```

`-p` Display progress and version information.

`-rd`
Suppress warning messages if a resource type is redeclared.

`-s[kip]` *type-expr* `[(`*ID1*`[:`*ID2*`])]`
`-s[kip]` *type-expr* `[`*resourceName*`]`
Skip resources of type *type-expr* in the resource file. For example, you can save execution time by skipping `CODE` resources. The `-s` option may be repeated any number of times.

Note: *type-expr* is an expression, so straight quotes (') might be needed. If an ID, range of IDs, or name is given, the entire option parameter must be quoted. See the note under `-only` *type-expr* earlier in this section.

`-s[kip]` *type*
Skip resources of the specified type. This is a simpler version of the `-s` option. No quotes are needed to specify a literal as long as it starts with a letter.

`-u[ndef]` *macro*
Undefine the macro variable *macro*. This is the same as writing

```
#undef macro
```

at the beginning of the input file. (See Appendix C in *A/UX Toolbox: Macintosh ROM Interface* for a description of macro variables.) It is meaningful to undefine only the preset macro variables. This option may be repeated.

EXAMPLES

The command

```
derez -i /usr/lib/mac/rincludes sample types.r > sample.r
```

decompiles the resource file `%sample`, using the definitions in the file `/usr/lib/mac/rincludes/types.r` and putting the output into the file `sample.r`. If it has access to the type definitions, `derez` generates more meaningful output.

```
derez -o MENU -i /usr/lib/mac/rincludes sample types.r
```

displays all of the `MENU` resources in `%sample`. The type definition for `MENU` resources is in the file `types.r`.

FILES

`/mac/bin/derez`

SEE ALSO

A/UX Toolbox: Macintosh ROM Interface.

NAME

deroff — remove nroff/troff, tbl, and eqn constructs

SYNOPSIS

deroff [-mx] [-w] [*file*...]

DESCRIPTION

deroff reads each of the *files* in sequence and removes all troff(1) requests, macro calls, backslash constructs, eqn(1) constructs (between .EQ and .EN lines, and between delimiters), and tbl(1) descriptions, perhaps replacing them with white space (blanks and blank lines), and writes the remainder of the file on the standard output. deroff follows chains of included files (.so and .nx troff commands); if a file has already been included, a .so naming that file is ignored and a .nx naming that file terminates execution. If no input file is given, deroff reads the standard input.

The -m flag option may be followed by an m, s, or l. The -mm flag option causes the macros be interpreted so that only running text is output (that is, no text from macro lines.) The -ml flag option forces the -mm flag option and also causes deletion of lists associated with the mm macros.

If the -w flag option is given, the output is a word list, one *word* per line, with all other characters deleted. Otherwise, the output follows the original, with the deletions mentioned above. In text, a *word* is any string that contains at least two letters and is composed of letters, digits, ampersands (&), and apostrophes ('); in a macro call, however, a *word* is a string that *begins* with at least two letters and contains a total of at least three letters. Delimiters are any characters other than letters, digits, apostrophes, and ampersands. Trailing apostrophes and ampersands are removed from words.

EXAMPLES

The command

```
deroff textfile
```

removes all nroff, troff, and macro definitions from textfile.

FILES

/usr/bin/deroff

SEE ALSO

eqn(1), nroff(1), tbl(1), troff(1).

BUGS

deroff is not a complete troff interpreter, so it can be confused by subtle constructs. Most such errors result in too much rather than too little output.

The -ml flag option does not handle nested lists correctly.

NAME

df — report number of free disk blocks

SYNOPSIS

df [-t] [-f] [-T] [*file...*]

DESCRIPTION

df prints out the number of free blocks and free inodes available for mounted file systems. File systems may be specified either by device name (for example, /dev/dsk/c0s0d0) or by filenames (for example, /usr). If the argument is a regular file or directory, df reports on the amount of free space for the file system on which that file resides. If no argument is specified, the amount of free space on all of the mounted file systems is printed. The count of free inodes on remotely mounted file systems is always zero.

The -t flag option causes the total allocated block and inode figures to be reported as well.

If the -f flag option is given, an actual count of the blocks in the free list is made (free inodes are not reported).

If the -T flag option is given, the next argument is the file system type. The accepted types are: 4.2, 5.2, nfs, and pc; see fstab(4) for a description of the legal file system types.

FILES

/bin/df	
/dev/dsk/*	disk partitioning
/etc/mtab	list of currently mounted file systems

SEE ALSO

mount(1M), fs(4), fstab(4), mtab(4).

BUGS

Since inodes are file system dependent, the number of inodes reported on remotely mounted file systems is always zero.

NAME

diction, explain — locate wordy sentences in a document

SYNOPSIS

diction [-ml] [-mm] [-n] [-f *pfile*] *file*...

explain

DESCRIPTION

diction finds all sentences in a document that contain phrases from a data base of bad or wordy diction. Each phrase is bracketed with []. Because diction runs deroff before looking at the text, formatting header files should be included as part of the input. The default macro package -ms may be overridden with the flag -mm. The flag -ml (which causes deroff to skip lists) should be used if the document contains many lists of nonsentences. The user may supply her/his own pattern file to be used in addition to the default file with -f *pfile*. A *pfile* is just a list of (wordy) phrases, with one phrase per line. The default *pfile* is /usr/lib/dict.d. If the flag -n is also supplied, the default *pfile* will be suppressed.

explain is an interactive thesaurus for the phrases found by diction. It prompts you with:

```
phrase?
```

to which you should respond by typing the *phrase* flagged by diction that you need explained. The explanation tells what to use instead of *phrase*. To get out of explain, press DELETE.

FILES

/usr/ucb/diction
/usr/ucb/explain

SEE ALSO

deroff(1), style(1), spell(1).

BUGS

Use of nonstandard formatting macros may cause incorrect sentence breaks. In particular, diction does not recognize -me.

NAME

diff — differential file and directory comparator

SYNOPSIS

diff [-l] [-r] [-s] [-S*name*] [-cefh] [-b] *dir1 dir2*

diff [-cefh] [-b] *file1 file2*

diff [-D*string*] [-b] *file1 file2*

DESCRIPTION

If both arguments are directories, diff sorts the contents of the directories by name, and then runs the regular file diff algorithm (described below) on text files which are different. Binary files which differ, common subdirectories, and files which appear in only one directory are listed. Flag options when comparing directories are:

-l long output format; each text file diff is piped through pr(1) to paginate it, other differences are remembered and summarized after all text file differences are reported.

-r causes application of diff recursively to common subdirectories encountered.

-s causes diff to report files which are the same, which are otherwise not mentioned.

-S*name*
starts a directory diff in the middle beginning with file *name*.

When run on regular files, and when comparing text files which differ during directory comparison, diff tells what lines must be changed in the files to bring them into agreement. Except in rare circumstances, diff finds a smallest sufficient set of file differences. If neither *file1* nor *file2* is a directory, then either may be given as "-", in which case the standard input is used. If *file1* is a directory, then a file in that directory whose filename is the same as the filename of *file2* is used (and vice versa).

There are several flag options for output format; the default output format contains lines of these forms:

n1 a *n3,n4*
n1,n2 d *n3*
n1,n2 c *n3,n4*

These lines resemble ed commands to convert *file1* into *file2*. The numbers after the letters pertain to *file2*. In fact, by exchanging a for d and reading backward, one may ascertain equally how to convert *file2* into *file1*. As in ed, identical pairs where *n1* = *n2* or *n3* = *n4* are abbreviated as a single number.

Following each of these lines come all the lines that are affected in the first file flagged by "<", then all the lines that are affected in the second file flagged by ">".

Except for -b, which may be given with any of the others, the following flag options are mutually exclusive:

-c produces a diff with lines of context. The default is to present 3 lines of context and may be changed, e.g., to 10, by -c10. With -c the output format is modified slightly: the output beginning with identification of the files involved and their creation dates and then each change is separated by a line with a dozen *'s. The lines removed from *file1* are marked with "-"; those added to *file2* are marked "+". Lines which are changed from one file to the other are marked in both files with "!".

-e producing a script of *a*, *c*, and *d* commands for the editor ed, which will recreate *file2* from *file1*. In connection with -e, the following shell program may help maintain multiple versions of a file. Only an ancestral file ($1) and a chain of version-to-version ed scripts ($2,$3,. . .) made by diff need be on hand. A "latest version" appears on the standard output.

```
(shift; cat $*; echo '1,$p') | ed - $1
```

Extra commands are added to the output when comparing directories with -e, so that the result is a sh(1) script for converting text files which are common to the two directories from their state in *dir1* to their state in *dir2*. Since such a shell script is useful only in a file that you may run on other files, it is best to redirect the output of this command into a file.

-f produces a script similar to that of -e, not useful with ed, and in the opposite order.

`-h` does a fast, half-hearted job. It works only when changed stretches are short and well-separated, but does work on files of unlimited length.

`-Dstring`
causes `diff` to create a merged version of *file1* and *file2* on the standard output, with C preprocessor controls included so that a compilation of the result without defining *string* is equivalent to compiling *file1*, while defining *string* will yield *file2*.

`-b` causes trailing blanks (spaces and tabs) to be ignored, and other strings of blanks to compare equal.

FILES

```
/usr/bin/diff
/tmp/d?????
/usr/lib/diffh
/bin/pr
```

SEE ALSO

`bdiff`(1), `cmp`(1), `cpp`(1), `comm`(1), `diff3`(1), `ed`(1). "Other Tools" in *AIUX Programming Languages and Tools, Volume 2*.

DIAGNOSTICS

Exit status is 0 for no differences, 1 for some, 2 for trouble.

BUGS

Editing scripts produced under the `-e` or `-f` flag option are naive about creating lines consisting of a single ".".

When comparing directories with the `-b` flag option specified, `diff` first compares the files as with `cmp`, and then decides to run the `diff` algorithm if they are not equal. This may cause a small amount of spurious output if the files then turn out to be identical, because the only differences are insignificant blank string differences.

If an unrecognized flag option is specified, `diff` performs the default operation anyway.

`diff` may not work if files contain a very long line, or if files are very long.

NAME

diff3 — 3-way differential file comparison

SYNOPSIS

diff3 [-3] [-e] [-x] *file1 file2 file3*

DESCRIPTION

diff3 compares three versions of a file, and publishes disagreeing ranges of text flagged with these codes:

====	all three files differ
====1	*file1* is different
====2	*file2* is different
====3	*file3* is different

The type of change suffered in converting a given range of a given file to some other is indicated in one of these ways:

f : *n1* a — Text is to be appended after line number *n1* in file *f*, where *f* = 1, 2, or 3.

f : *n1* , *n2* c — Text is to be changed in the range line *n1* to line *n2*. If *n1* = *n2*, the range may be abbreviated to *n1*.

The original contents of the range follows immediately after a c indication. When the contents of two files are identical, the contents of the lower-numbered file is suppressed.

Under the -e flag option, diff3 publishes a script for the editor ed which results in all changes from *file2* and *file3* being implemented into file 1. i.e., the changes that normally would be flagged ==== and ====3. Flag option -x (-3) produces a script to incorporate only changes flagged ==== (====3). The following command will apply the resulting script to *file1*.

```
(cat script; echo '1,$p') | ed - file1
```

EXAMPLES

If file f1 contains the following text:

```
This is a file.
This is the first of three files.
This is not the last file.
```

and file f2 contains:

```
This is a file.
This is the second of three files.
This is not the last file.
```

and file f3 contains:

```
This is a file.
This is the third of three files.
This is the last file.
```

then

```
diff3 f1 f2 f3
```

will return

```
====
1:2,3c
   This is the first of three files.
   This is not the last file.
2:2,3c
   This is the second of three files.
   This is not the last file.
3:2,3c
   This is the third of three files.
   This is the last file
```

FILES

/usr/bin/diff3/
/tmp/d3*
/usr/lib/diff3prog

SEE ALSO

bdiff(1), cmp(1), diff(1).

BUGS

Text lines that consist of a single . will defeat -e.
diff3 won't work on files longer than 64K bytes.

NAME

diffmk — mark differences between files

SYNOPSIS

diffmk [–] *file1 file2 file3*

DESCRIPTION

diffmk compares two versions of a file and creates a third file that includes "change mark" requests (.mc) for nroff(1) or troff(1). The placeholders *file1* and *file2* are the old and new versions of the file. If – is given, *file1* is read from standard input. diffmk generates *file3*, which contains the lines of *file2* plus inserted formatter "change mark" requests. When *file3* is formatted, changed or inserted text is shown by | at the right margin of each line. The position of deleted text is shown by *, a single asterisk.

If anyone is so inclined, diffmk can be used to produce listings of C (or other) programs with changes marked. A typical command line for such use is

```
diffmk old.c new.c tmp; nroff macs tmp | lp
```

where the file macs contains

```
.pl 1
.ll 77
.nf
.eo
.nc
```

The .ll request might specify a different line length, depending on the nature of the program being printed. The .eo and .nc requests are probably needed only for C programs.

FILES

/usr/bin/diffmk

SEE ALSO

bdiff(1), cmp(1), diff(1), diff3(1), nroff(1), troff(1).

BUGS

Aesthetic considerations may dictate manual adjustment of some output. File differences involving only formatting requests may produce undesirable output. For example, replacing .sp by .sp 2 produces a "change mark" on the preceding or following line of output.

NAME

dircmp — directory comparison

SYNOPSIS

dircmp [-d] [-s] [-*wn*] *dir1 dir2*

DESCRIPTION

dircmp examines *dir1* and *dir2* and generates various tabulated information about the contents of the directories. Listings of files that are unique to each directory are generated for all the flag options. If no flag option is entered, a list is output indicating whether the filenames common to both directories have the same contents.

-d Compare the contents of files with the same name in both directories and output a list telling what must be changed in the two files to bring them into agreement. The list format is described in diff(1).

-s Suppress messages about identical files.

-*wn*
Change the width of the output line to *n* characters. The default width is 72.

EXAMPLES

```
dircmp d1 d2
```

will show the differences between the directories d1 and d2.

FILES

/bin/dircmp

SEE ALSO

cmp(1), bdiff(1), diff(1), diff3(1), diffmk(1).

See basename(1)

NAME

dis — disassembler

SYNOPSIS

dis [-d *sec*] [-da *sec*] [-F *function*] [-l *string*] [-L] [-o] [-t *sec*] [-V] *file* ...

DESCRIPTION

The dis command produces an assembly language listing of each of its object *file* arguments. The listing includes assembly statements and the binary that produced those statements.

The following flag options are interpreted by the disassembler and may be specified in any order:

-o	Print numbers in octal. Default is hexadecimal.
-V	Write the version number of the disassembler to standard error.
-L	Invoke a lookup of C source labels in the symbol table for subsequent printing.
-d *sec*	Disassemble the named section as .data, printing the offset of the data from the beginning of the section.
-da *sec*	Disassemble the named section as .data, printing the actual address of the data.
-t *sec*	Disassemble the named section as .text.
-l *string*	Disassemble the library file specified as *string*. For example, one would issue the command `dis -l x -l z` to disassemble libx.a and libz.a. All libraries are assumed to be in /lib.

If the -d, -da, or -t flag options are specified, only those named sections from each user-supplied filename are disassembled. Otherwise, all sections containing text are disassembled.

If the -F flag option is specified, only those named functions from each user-supplied filename are disassembled.

On output, a number enclosed in brackets at the beginning of a line, such as [5], means that dis has reached the point in the assembly code where a C language line (numbered as stated) begins. If a breakpoint is placed there using sdb/adb, the debugger used

will stop on a C line. An expression such as <40> in the operand field, following a relative displacement for control transfer instructions, is the computed address within the section to which control will be transferred. A C function name will appear in the first column, followed by ().

FILES

/bin/dis

SEE ALSO

as(1), cc(1), ld(1), strings(1).

DIAGNOSTICS

The self-explanatory diagnostics indicate errors in the command line or problems encountered with the specified files.

See enable(1)

NAME

domainname — set or display name of current domain system

SYNOPSIS

domainname [*name-of-domain*]

DESCRIPTION

Without an argument, domainname displays the name of the current domain. Only the superuser can set the domain name by giving an argument; this is usually done in the startup script /etc/sysinitrc. Currently, domains are only used by the yellow pages, to refer collectively to a group of hosts.

To make a permanent change to the domain name, edit the second field of /etc/HOSTNAME and then reboot.

FILES

/bin/domainname

SEE ALSO

ypinit(1M).
A/UX Network System Administration.

NAME

du — summarize disk usage

SYNOPSIS

du [-a] [-r] [-s] [*names*]

DESCRIPTION

du gives the number of blocks contained in all files and (recursively) directories within each directory and file specified by the *names* argument. The default system size for physical blocks is 512 bytes. The block count includes the indirect blocks of the file. If *names* is missing, . is used.

The optional argument -s causes only the grand total (for each of the specified *names*) to be given. The optional argument -a causes an entry to be generated for each file. Absence of either causes an entry to be generated for each directory only.

du is normally silent about directories that cannot be read, files that cannot be opened, etc. The -r flag option will cause du to generate messages in such instances.

A file with two or more links is only counted once.

EXAMPLES

```
du dir1 dir2
```

produces a count of the number of (512-byte) blocks in each of the directories. In order to see how many blocks are in each file, the -a flag option must be used.

FILES

/bin/du

SEE ALSO

df(1).

BUGS

If the -a flag option is not used, nondirectories given as arguments are not listed.

If there are too many distinct linked files, du will count the excess files more than once.

Files with holes in them will get an incorrect block count.

NAME

dump — dump selected parts of an object file

SYNOPSIS

dump [[-a] [-c] [-f] [-g] [-h] [-l] [-o] [-r] [-s] [-t] [-z *name*]] [[-d *number*] [+d *number*] [-n *name*] [-p] [-t *index*] [+t *index*] [-u] [-v] [-z *name*,*number*] [+z *name*]] *file* ...

DESCRIPTION

The dump command dumps selected parts of each of its object *file* arguments.

This command accepts both object files and archives of object files. It processes each file argument according to one or more of the following flag options:

-a	Dump the archive header of each member of each archive file argument.
-f	Dump each file header.
-g	Dump the global symbols in the symbol table of a version 6.0 archive.
-o	Dump each optional header.
-h	Dump section headers.
-s	Dump section contents.
-r	Dump relocation information.
-l	Dump line number information.
-t	Dump symbol table entries.
-z *name*	Dump line number entries for the named function.
-c	Dump the string table.

The following modifiers are used in conjunction with the flag options listed above to modify their capabilities.

-d *number*	Dump the section number or range of sections starting at *number* and ending either at the last section number or *number* specified by +d.
+d *number*	Dump sections in the range either beginning with first section or beginning with section specified by -d.

`-n` *name* Dump information pertaining only to the named entity. This *modifier* applies to `-h`, `-s`, `-r`, `-l`, and `-t`.

`-p` Suppress printing of the headers.

`-t` *index* Dump only the indexed symbol table entry. When the `-t` is used in conjunction with `+t`, it specifies a range of symbol table entries.

`+t` *index* Dump the symbol table entries in the range ending with the indexed entry. The range begins at the first symbol table entry or at the entry specified by the `-t` flag option.

`-u` Underline the name of the file for emphasis.

`-v` Dump information in symbolic representation rather than numeric (e.g., `C_STATIC` instead of `0X02`). This *modifier* can be used with all the above flag options except the `-s` and `-o` flag options of `dump`.

`-z` *name,number*
Dump line number entry or range of line numbers starting at *number* for the named function.

`+z` *number* Dump line numbers starting at either function *name* or *number* specified by `-z`, up to *number* specified by `+z`.

Blanks separating a flag option and its modifier are optional. The comma separating the name from the number modifying the `-z` flag option may be replaced by a blank.

The `dump` command attempts to format the information it dumps in a meaningful way, printing certain information in character, hex, octal, or decimal representation, as appropriate.

FILES

`/bin/dump`

SEE ALSO

`as`(1), `dis`(1), `od`(1), `nm`(1), `strings`(1), `dumpfs`(1M), `restore`(1M), `a.out`(4), `ar`(4).

See ex(1)

NAME

echo — echo arguments

SYNOPSIS

echo [*arg*] ...

DESCRIPTION

echo writes its arguments separated by blanks and terminated by a newline on the standard output. It also understands C-like escape conventions; beware of conflicts with the shell's use of \:

- \b backspace
- \c print line without newline
- \f form-feed
- \n newline
- \r carriage return
- \t tab
- \v vertical tab
- \\ backslash
- *n* the 8-bit character whose ASCII code is the 1-, 2- or 3-digit octal number *n*, which must start with a zero.

echo is useful for producing diagnostics in command files and for sending known data into a pipe. A version of echo is built into the Bourne shell (sh(1)). Similar versions are also built into ksh(1) and csh(1).

EXAMPLES

```
echo curmudgeon
```

simply responds

```
curmudgeon
```

on the standard output.

FILES

/bin/echo

SEE ALSO

csh(1), ksh(1), sh(1).

NAME

ed, red — text editor

SYNOPSIS

ed [-] [-p *string*] [-x] [*file*]

red [-] [-p *string*] [-x] [*file*]

DESCRIPTION

ed is the standard text editor. If the *file* argument is given, ed simulates an *e* command (see below) on the named file; that is to say, the file is read into ed's buffer so that it can be edited. The optional – suppresses the printing of character counts by e, r, and w commands, of diagnostics from e and q commands, and of the ! prompt after a !*shell command*. The -p flag option allows the user to specify a prompt string. The string must be enclosed in double quotes. If -x is present, an x command is simulated first to handle an encrypted file. ed operates on a copy of the file it is editing; changes made to the copy have no effect on the file until a w (write) command is given. The copy of the text being edited resides in a temporary file called the *buffer*. There is only one buffer.

red is a restricted version of ed. It will allow editing of files only in the current directory. It prohibits executing shell commands via !*shell command*. Attempts to bypass these restrictions result in the error message:

```
restricted shell
```

Both ed and red support the fspec(4) formatting capability. After including a format specification as the first line of *file* and invoking ed with your terminal in stty -tabs or stty tab3 mode (see stty(1), the specified tab stops will be used automatically when scanning *file*. For example, if the first line of a file contained:

```
<:t5,10,15 s72:>
```

tab stops would be set at columns 5, 10, and 15, and a maximum line length of 72 would be imposed.

Note: While entering text, tab characters, when typed, are expanded to every eighth column, as is the default.

Commands to ed have a simple and regular structure: zero, one, or two *addresses* followed by a single-character *command*, followed by any applicable parameters to that command. These addresses specify one or more lines in the buffer. Every command that requires addresses has default addresses, so that the addresses very often can be omitted.

In general, only one command may appear on a line. Certain commands allow the input of text. This text is placed in the appropriate place in the buffer. While ed is accepting text, it is said to be in "input mode." In this mode, *no* commands are recognized; all input is merely collected. Input mode is left by typing a period (.) alone at the beginning of a line.

ed supports a limited form of "regular expression" (RE) notation; regular expressions are used in addresses to specify lines and in some commands (e.g., *s*) to specify portions of a line that are to be substituted. A regular expression specifies a set of character strings. A member of this set of strings is said to be *matched* by the RE. The REs allowed by ed are constructed as follows:

The following *one-character* REs match a *single* character:

1.1 An ordinary character (*not* one of those discussed in 1.2 below) is a one-character RE that matches itself.

1.2 A backslash (\ ) followed by any special character is a one-character RE that matches the special character itself. The special characters are:

a. ., *, [, and \ (period, asterisk, left square bracket, and backslash, respectively), which are always special, *except* when they appear within square brackets ([ ]; see 1.4 below).

b. ^ (circumflex), which is special at the *beginning* of an *entire* RE (see 3.1 and 3.2 below), or when it immediately follows the left of a pair of square brackets ([ ]) (see 1.4 below).

c. $ (currency symbol), which is special at the *end* of an entire RE (see 3.2 below).

d. The character used to bound (i.e., delimit) an entire RE, which is special for that RE (for example, see how slash (/) is used in the g command, below.)

1.3 A period (`.`) is a one-character RE that matches any character except newline.

1.4 A nonempty string of characters enclosed in square brackets (`[ ]`) is a one-character RE that matches *any one* character in that string. If, however, the first character of the string is a circumflex (`^`), the one-character RE matches any character *except* newline and the remaining characters in the string. The `^` has this special meaning *only* if it occurs first in the string. The minus (–) may be used to indicate a range of consecutive ASCII characters; for example, `[0-9]` is equivalent to `[0123456789]`. The – loses this special meaning if it occurs first (after an initial `^`, if any) or last in the string. The right square bracket (`]`) does not terminate such a string when it is the first character within it (after an initial `^`, if any); e.g., `[ ]a-f]` matches either a right square bracket (`]`) or one of the letters `a` through `f`, inclusive. The four characters listed in 1.2.a (above) stand for themselves within such a string of characters.

The following rules may be used to construct REs from one-character REs:

2.1 A one-character RE is a RE that matches whatever the one-character RE matches.

2.2 A one-character RE followed by an asterisk (`*`) is a RE that matches *zero* or more occurrences of the one-character RE. If there is any choice, the longest leftmost string that permits a match is chosen.

2.3 A one-character RE followed by `\{`*m*`\}`, `\{`*m*`,\}`, or `\{`*m*`,`*n*`\}` is a RE that matches a *range* of occurrences of the one-character RE. The values of *m* and *n* must be nonnegative integers less than 256:

`\{`*m*`\}` matches *exactly m* occurrences;

`\{`*m*`,\}` matches *at least m* occurrences;

`\{`*m*`,`*n*`\}` matches *any number* of occurrences *between m* and *n* inclusive.

Whenever a choice exists, the RE matches as many occurrences as possible.

2.4 The concatenation of REs is a RE that matches the concatenation of the strings matched by each component of the RE.

2.5 A RE enclosed between the character sequences \(and \) is a RE that matches whatever the unadorned RE matches.

2.6 The expression *n* matches the same string of characters as was matched by an expression enclosed between \(and \) *earlier* in the same RE. Here *n* is a digit; the sub-expression specified is that beginning with the *n*-th occurrence of \(counting from the left. For example, the expression ^\(.*\)\1$ matches a line consisting of two repeated appearances of the same string.

Finally, an *entire* RE may be constrained to match only an initial segment or final segment of a line (or both).

3.1 A caret (^) at the beginning of an entire RE constrains that RE to match an *initial* segment of a line.

3.2 A currency symbol ($) at the end of an entire RE constrains that RE to match a *final* segment of a line.

The construction ^*entire RE*$ constrains the entire RE to match the entire line.

The null RE (e.g., //) is equivalent to the last RE encountered. See the paragraph before FILES, below.

To understand addressing in ed, it is necessary to know that at any time there is a *current line*. Generally speaking, the current line is the last line affected by a command; the exact effect on the current line is discussed under the description of each command. *addresses* are constructed as follows:

1. The character . addresses the current line.
2. The character $ addresses the last line of the buffer.
3. A decimal number *n* addresses the *n*-th line of the buffer.
4. 'x addresses the line marked with the mark name character *x*, which must be a lowercase letter. Lines are marked with the *k* command (described below). If *x* was not used to mark a line, 'x addresses line 0.
5. A RE enclosed by slashes (/) addresses the first line found by searching *forward* from the line *following* the current line toward the end of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the beginning of the buffer and continues up to and including the current line, so that the entire buffer is

searched. See the paragraph before FILES, below.

6. A RE enclosed in question marks (?) addresses the first line found by searching *backward* from the line *preceding* the current line toward the beginning of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the end of the buffer and continues up to and including the current line. See also the last paragraph before FILES, below.
7. An address followed by a plus sign (+) or a minus sign (–) followed by a decimal number specifies that address plus (respectively minus) the indicated number of lines. The plus sign may be omitted.
8. If an address begins with + or –, the addition or subtraction is taken with respect to the current line; e.g, `-5` is understood to mean `.-5`.
9. If an address ends with + or –, then 1 is added to or subtracted from the address, respectively. As a consequence of this rule and of rule 8 immediately above, the address – refers to the line preceding the current line. (To maintain compatibility with earlier versions of the editor, the character ^ in addresses is entirely equivalent to –.) Moreover, trailing + and – characters have a cumulative effect, so -- refers to the current line less 2.
10. For convenience, a comma (`,`) stands for the address pair `1,$`, while a semicolon (`;`) stands for the pair `.,$`.

Commands may require zero, one, or two addresses. Commands that require no addresses regard the presence of an address as an error. Commands that accept one or two addresses assume default addresses when an insufficient number of addresses is given; if more addresses are given than such a command requires, the last one(s) are used.

Typically, addresses are separated from each other by a comma (`,`). They may also be separated by a semicolon (`;`). In the latter case, the current line (`.`) is set to the first address, and only then is the second address calculated. This feature can be used to determine the starting line for forward and backward searches (see rules 5. and 6. above). The second address of any two-address sequence must correspond to a line that follows, in the buffer, the line corresponding to the first address.

In the following list of ed commands, the default addresses are shown in parentheses. The parentheses are *not* part of the address; they show that the given addresses are the default.

It is generally illegal for more than one command to appear on a line. Any command (except e, f, r, or w) may be suffixed by l, n or p, however, in which case the current line is either listed, numbered or printed, respectively, as discussed below under the l, n and p commands.

(.)a
text
.

The append command reads the given text and appends it after the addressed line; . is left at the last inserted line, or, if there were none, at the addressed line. Address 0 is legal for this command: it causes the appended text to be placed at the beginning of the buffer. The maximum number of characters that may be entered from a terminal is 256 per line (including the newline character).

(.)c
text
.

The change command deletes the addressed lines, then accepts input text that replaces these lines; . is left at the last line input, or, if there were none, at the first line that was not deleted.

(.,.)d

The delete command deletes the addressed lines from the buffer. The line after the last line deleted becomes the current line; if the lines deleted were originally at the end of the buffer, the new last line becomes the current line.

e *file*

The edit command causes the entire contents of the buffer to be deleted, and then the named file to be read in; . is set to the last line of the buffer. If no file name is given, the currently-remembered filename, if any, is used (see the f command). The number of characters read is typed; *file* is remembered for possible use as a default filename in subsequent e, r, and w commands. If *file* is replaced by !, the rest of the line is taken to be a shell (sh(1)) command whose output is to be read. Such a shell command is *not* remembered as the current filename. See also DIAGNOSTICS, below.

E *file* The E command is like e, except that the editor does not check to see if any changes have been made to the buffer since the last w command.

f *file* If *file* is given, this command changes the currently-remembered filename to *file*; otherwise, it prints the currently-remembered filename.

(1,$)g/*RE*/*command list*
In the global command, the first step is to mark every line that matches the given RE. Then, for every such line, the given *command list* is executed with . initially set to that line. A single command or the first of a list of commands appears on the same line as the global command. All lines of a multi-line list except the last line must be ended with a \; a, i, and c commands and associated input are permitted. The . terminating input mode may be omitted if it would be the last line of the *command list*. An empty *command list* is equivalent to the p command.

(1,$)G/*RE*/
In the interactive global command, the first step is to mark every line that matches the given RE. Then, for every such line, that line is printed, . is changed to that line, and any *one* command (other than one of the a, c, i, g, G, v, and V commands) may be input and is executed. After the execution of that command, the next marked line is printed, and so on; a newline acts as a null command; an & causes the re-execution of the most recent command executed within the current invocation of G. Note that the commands input as part of the execution of the G command may address and affect *any* lines in the buffer. The G command can be terminated by an interrupt signal (ASCII DELETE or BREAK). A command that causes an error terminates the G command.

h The help command gives a short error message that explains the reason for the most recent ? diagnostic.

H The Help command causes ed to enter a mode in which error messages are printed for all subsequent ? diagnostics. It will also explain the previous ? if there was one. The H command alternately turns this mode on and off;

it is initially off.

`(.)i`
text
`.`

The insert command inserts the given text before the addressed line; `.` is left at the last inserted line, or, if there were none, at the addressed line. This command differs from the `a` command only in the placement of the input text. Address 0 is not legal for this command. The maximum number of characters that may be entered from a terminal is 256 per line (including the newline character).

`(.,.+1)j`
Join contiguous lines by removing the appropriate newline characters. If exactly one address is given, this command does nothing.

`(.)k`*x* Mark the addressed line with name *x*, which must be a lowercase letter. The address *'x* then addresses this line; `.` is unchanged.

`(.,.)l`
List the addressed lines in an unambiguous way: a few nonprinting characters (e.g., *tab, backspace*) are represented by mnemonic overstrikes. All other nonprinting characters are printed in octal, and long lines are folded. An `l` command may be appended to any other command other than `e`, `f`, `r`, or `w`.

`(.,.)m`*a*
Move addressed line(s) to after the line addressed by `a`. Address 0 is legal for `a` and causes the addressed line(s) to be moved to the beginning of the file. It is an error if address `a` falls within the range of moved lines; `.` is left at the last line moved.

`(.,.)n`
Prints the addressed lines, preceding each line by its line number and a tab character; `.` is left at the last line printed. The `n` command may be appended to any other command other than `e`, `f`, `r`, or `w`.

`(.,.)p`
Print the addressed lines; `.` is left at the last line printed. The `p` command may be appended to any other command other than `e`, `f`, `r`, or `w`. For example, `dp` deletes

the current line and prints the new current line.

`P` The editor will prompt with a * for all subsequent commands. The `P` command alternately turns this mode on and off; it is initially off.

`q` Exit. `ed` No automatic write of a file is done (but see DIAGNOSTICS, below).

`Q` Exit `ed` without checking if changes have been made in the buffer since the last `w` command.

`( $ ) r` *file*
The read command reads in the given file after the addressed line. If no filename is given, the currently-remembered filename, if any, is used (see `e` and `f` commands). The currently-remembered filename is *not* changed unless *file* is the very first filename mentioned since `ed` was invoked. Address 0 is legal for `r` and causes the file to be read at the beginning of the buffer. If the read is successful, the number of characters read is typed; `.` is set to the last line read in. If *file* is replaced by `!`, the rest of the line is taken to be a shell (`sh`(1)) command whose output is to be read. For example, `$r !ls` appends current directory to the end of the file being edited. Such a shell command is *not* remembered as the current filename.

`( . , . ) s`/*RE*/*replacement*/ or
`( . , . ) s`/*RE*/*replacement*/`g` or
`( . , . ) s`/*RE*/*replacement*/*n*
Search each addressed line for an occurrence of the specified RE. In each line in which a match is found, all (nonoverlapped) matched strings are replaced by the *replacement* if the global replacement indicator `g` appears after the command. If the global indicator does not appear, only the first occurrence of the matched string is replaced. Sometimes substitution of an RE results in the last (or only) affected line being printed out. This occurs only when substitution is not global or of an *n*th occurrence. If a number *n* appears after the command, only the *n*th occurrence of the matched string on each addressed line is replaced. It is an error for the substitution to fail on *all* addressed lines. Any character other than space or newline may be used instead of / to del-

imit the RE and the *replacement*; . is left at the last line on which a substitution occurred. See the paragraph before FILES, below.

An ampersand (&) appearing in the *replacement* is replaced by the string matching the RE on the current line. The special meaning of & in this context may be suppressed by preceding it by \. As a more general feature, the characters *n*, where *n* is a digit, are replaced by the text matched by the *n*-th regular subexpression of the specified RE enclosed between \(and \). When nested parenthesized subexpressions are present, *n* is determined by counting occurrences of \(starting from the left. When the character % is the only character in the *replacement*, the *replacement* used in the most recent substitute command is used as the *replacement* in the current substitute command. The % loses its special meaning when it is in a replacement string of more than one character or is preceded by a \.

A line may be split by substituting a newline character into it. The newline in the *replacement* must be escaped by preceding it by \. Such substitution cannot be done as part of a g or v command list.

(.,.)t*a*

Similar to the move (m) command, except that a copy of the addressed lines is placed after address *a* (which may be 0); . is left at the last line of the copy.

u

Undo the most recent command that modified anything in the buffer, namely the most recent a, c, d, g, i, j, m, r, s, t, v, G, or V command.

(1,$)v/*RE*/*command list*

This command is the same as the global command g, except that the *command list* is executed with . initially set to every line that does *not* match the RE.

(1,$)V/*RE*/

This command is the same as the interactive global command G except that the lines that are marked during the first step are those that do *not* match the RE.

(1,$)w *file*

Write the addressed lines into the named file. If the file

does not exist, it is created with mode 666 (readable and writable by everyone), unless your *umask* setting (see sh(1)) dictates otherwise. The currently-remembered filename is *not* changed unless *file* is the very first filename mentioned since ed was invoked. If no filename is given, the currently-remembered filename, if any, is used (see e and f commands); . is unchanged. If the command is successful, the number of characters written is typed. If *file* is replaced by !, the rest of the line is taken to be a shell (sh(1)) command whose standard input is the addressed lines. Such a shell command is *not* remembered as the current filename.

x A key string is demanded from the standard input. Subsequent e, r, and w commands will encrypt and decrypt the text with this key by the algorithm of crypt(1). An explicitly empty key turns off encryption. The encryption scheme used here is not secure.

($)= The line number of the addressed line is typed; address 0 is legal for this command. . is unchanged by this command.

!*shell command*
The remainder of the line after the ! is sent to the system shell (sh(1)) to be interpreted as a command. Within the text of that command, the unescaped character % is replaced with the remembered filename; if a ! appears as the first character of the shell command, it is replaced with the text of the previous shell command. Thus, ! ! will repeat the last shell command. If any expansion is performed, the expanded line is echoed; . is unchanged.

(.+1) newline
An address alone on a line causes the addressed line to be printed. A newline alone is equivalent to .+1p; it is useful for stepping forward through the buffer.

If an interrupt signal (ASCII or CONTROL-c is sent, ed prints a ? and returns to *its* command level.

Some size limitations: 512 characters per line, 256 characters per global command list, 64 characters per file name, and 128K characters in the buffer. The limit on the number of lines depends on the amount of user memory: each line takes 1 word.

When reading a file, ed discards ASCII NUL characters and all characters after the last newline. Files (e.g., a.out) that contain characters not in the ASCII set (bit 8 on) cannot be edited by ed.

If the closing delimiter of a RE or of a replacement string (e.g., /) would be the last character before a newline, that delimiter may be omitted, in which case the addressed line is printed. The following pairs of commands are equivalent:

```
s/s1/s2    s/s1/s2/p
g/s1       g/s1/p
?s1        ?s1?
```

EXAMPLES

```
ed text
```

invokes the editor with the file named text. For further examples, see "Using ed" in *A/UX Text Editing Tools.*

FILES

/bin/ed	
/bin red	
/tmp/*e#*	temporary; # is the process number.
ed.hup	work is saved here if the terminal is hung up.

DIAGNOSTICS

?	for command errors.
?*file*	for an inaccessible file. (use the h and H commands for detailed explanations).

If changes have been made in the buffer since the last w command that wrote the entire buffer, ed warns the user if an attempt is made to destroy ed's buffer via the e or q commands. It prints ? and allows one to continue editing. A second e or q command will take effect at any time, provided no further changes have been made to the file. The – command-line flag option inhibits this feature.

SEE ALSO

crypt(1), ex(1), grep(1), sed(1), sh(1), stty(1), vi(1), fspec(4), regexp(5).
"Using ed" in *A/UX Text Editing Tools.*

NOTES

The ! command and the ! escape from the e, r, and w commands cannot be used if the the editor is invoked from a restricted shell (see sh(1)).
The sequence \n in a RE does not match a newline character.
The l command mishandles interrupts.
Files encrypted directly with the crypt(1) command with the null key cannot be edited.
Characters are masked to 7 bits on input.

The -x flag option and the editor command X are not implemented in the international distribution.
If the editor input is coming from a command file (i.e., ed *file* < *ed-cmd-file*), the editor will exit at the first failure of a command that is in the command file.

edit(1)

See ex(1)

NAME

efl — Extended Fortran Language

SYNOPSIS

efl [–#] [–C] [–w] [*file* ...]

DESCRIPTION

efl compiles a program written in the efl language into clean Fortran on the standard output. efl provides the C-like control constructs similar to Ratfor:

statement grouping with braces.

decision-making:

if, if-else, and select-case (also known as switch-case);
while, for, Fortran do, repeat, and repeat ... until loops;
multi-level break and next.

efl has C-like data structures, e.g.:

```
struct
    {
    integer flags(3)
    character(8) name
    long real coords(2)
    } table(100)
```

The language offers generic functions, assignment operators (+=, &=, etc.), and sequentially evaluated logical operators (&& and ||). There is a uniform input/output syntax:

```
write(6,x,y:f(7,2),     do     i=1,10     {
a(i,j),z.b(i)  } )
```

efl also provides some syntactic *sugar*:

free-form input:
multiple statements per line; automatic continuation; statement label names (not just numbers).

comments:

```
# this is a comment.
```

translation of relational and logical operators:
>, >=, &, etc., become .GT., .GE., .AND., etc.

return expression to caller from function:

```
return (expression)
```

defines:

`define` *name replacement*

includes:

`include` *file*

`efl` understands several flag option arguments: `-w` suppresses warning messages, `-#` suppresses comments in the generated program, and the default flag option `-C` causes comments to be included in the generated program.

An argument with an embedded = (equal sign) sets an `efl` flag option as if it had appeared in an `option` statement at the start of the program. Many options are described in the reference manual. A set of defaults for a particular target machine may be selected by one of the choices: `system=unix`, `system=gcos`, or `system=cray`. The default setting of the `system` option is the same as the machine the compiler is running on. Other specific options determine the style of input/output, error handling, continuation conventions, the number of characters packed per word, and default formats.

`efl` is best used with `f77`(1).

EXAMPLES

The command sequence:

```
efl prog.for > prog.f
f77 prog.f -o prog
```

will process the program `prog.for` through `efl` and then run the `f77`(1) compiler on the output from `efl`, generating an executable file named `prog`.

FILES

`/usr/bin/efl`

SEE ALSO

`cc`(1), `f77`(1).

"`efl` Reference" in *A/UX Programming Languages and Tools, Volume 2*.

See grep(1)

NAME

eject — eject diskette from drive

SYNOPSIS

eject [0] [1] [/dev/rdsk/*name*]

DESCRIPTION

eject causes a floppy diskette drive (see fd(7)) to eject an inserted diskette. One of three arguments may be included on the command line. Arguments 0 or 1 specify diskette drive 0 or 1. Alternatively, /dev/rdsk/*name* specifies the full path name of the character special file for the device. If the device is not given, the device 0 is assumed.

FILES

/bin/eject
/dev/rdsk/c8d?s0

SEE ALSO

fd(7).

NAME

enable, disable — enable or disable LP printers

SYNOPSIS

enable *printers*
disable [-c] [-r[*reason*]] *printers*

DESCRIPTION

enable activates the named *printers*, enabling them to print requests taken by lp(1). Use lpstat(1) to find the status of printers.

disable deactivates the named *printers*, disabling them from printing requests taken by lp(1). By default, any requests that are currently printing on the designated printers will be reprinted in their entirety either on the same printer or on another member of the same class. Use lpstat(1) to find the status of printers. Flag options useful with disable are:

-c Cancel any requests that are currently printing on any of the designated printers.

-r [*reason*]
Associates a *reason* with the deactivation of the printers. This reason applies to all printers mentioned up to the next -r flag option. If the -r flag option is not present or the -r flag option is given without a reason, then a default reason will be used. *reason* is reported by lpstat(1).

FILES

/usr/bin/enable
/usr/bin/disable
/usr/spool/lp/*

SEE ALSO

lp(1), lpstat(1).
A/UX Local System Administration

NAME

enscript — convert text files to POSTSCRIPT format for printing

SYNOPSIS

enscript [-12BGghKklmoqRr] [-L*lines*] [-f*font*] [-F*hfont*] [-b*header*] [-p*out*] [*spoolopts*] [*files*]

DESCRIPTION

enscript reads plain text files, converts them to POSTSCRIPT format, and spools them for printing on a POSTSCRIPT printer. Fonts, headings, and limited formatting options may be specified.

For example:

```
enscript -paleph boring.txt
```

processes the file called boring.txt for POSTSCRIPT printing, writing the output to the file aleph.

```
enscript -2r boring.c
```

prints a two-up landscape listing of the file called boring.c on the default printer (see below).

Font specifications have two parts: A font name as known to POSTSCRIPT (for example, Times-Roman, Times-Roman BoldItalic , Helvetica, Courier), and a point size (1 point=1/72 inch). So, Courier-Bold8 is 8 point Courier Bold, Helvetica12 is 12 point Helvetica.

The environment variable ENSCRIPT may be used to specify defaults. The value of ENSCRIPT is parsed as a string of arguments before the arguments that appear on the command line. For example

```
ENSCRIPT='-fTimes-Roman8'
```

sets your default body font to 8 point Times Roman.

The possible options are:

-2 set in two columns.

-1 set in one column (the default).

-r rotate the output 90 degrees (landscape mode). This is good for output that requires a wide page or for program listings when used in conjunction with –2.

```
enscript -2r files
```

is a nice way to get program listings.

`-R` don't rotate, also known as portrait mode (the default).

`-G` print in gaudy mode: causes page headings, dates, page numbers to be printed in a flashy style, at some slight performance expense.

`-l` simulate a line printer: make pages 66 lines long and omit headers.

`-B` omit page headings.

`-b`*header*
sets the string to be used for page headings to *header*. The default header is constructed from the file name, its last modification date, and a page number.

`-L`*lines* set the maximum number of lines to output on a page. `enscript` usually computes how many to put on a page based on point size, and may put fewer per page than requested by *lines*.

`-f`*font* sets the font to be used for the body of each page. Defaults to `Courier10`, unless two column rotated mode is used, in which case it defaults to `Courier7`.

`-F`*hfont* sets the font to be used for page headings. Defaults to `Courier-Bold10`.

`-p`*out* causes the POSTSCRIPT file to be written to the named file rather than being spooled for printing. As a special case, `-p -` will send the POSTSCRIPT to the standard output.

`-g` enables the printing of files containing non-printing characters. Any file with more than a small number of non-printing characters is suspected of being garbage and is not printed unless this option is used.

`-o` If `enscript` cannot find characters in a font, the missing characters are listed.

`-q` causes `enscript` to be quiet about what it is doing. `enscript` won't report about pages, destination, omitted characters, and so forth. Fatal errors are still reported to the standard error output.

`-k` enables page prefeed (if the printer supports it). This allows simple documents (e.g., program listings in one font) to print somewhat faster by keeping the printer run-

ning between pages.

`-K` disable page prefeed (the default).

`-h` suppress printing of job burst page.

ENVIRONMENT

`ENSCRIPT`	string of options to be used by `enscript`.
`PSLIBDIR`	path name of a directory to use instead of `/usr/lib/ps` for `enscript` prologue and font metric files.
`PSTEMPDIR`	path name of temporary directory to use instead of `XPSTEMDIRX` of spooled temporary files.
`LPDEST`	the name of a printer for `lp` to use. If `LPDEST` is not set, `enscript` will spool to a printer class named `PostScript`.

FILES

`/usr/bin/enscript`	
`/usr/lib/ps/*.afm`	font metrics files.
`/usr/lib/ps/enscript.pro`	prologue for `enscript` files.

SEE ALSO

`cancel`(1), `lp`(1), `lpr`(1), `lprm`(1), `lpstat`(1), `pr`(1), `ps630`(1), `getopt`(3).

FEATURES

Options and the `ENSCRIPT` environment string are parsed in `getopt`(3) fashion.

BUGS

Long lines are truncated. Line truncation may be off by a little bit as printer margins vary. There should be a "wrap" option and multiple (truncated or wrapped) columns.

NAME

env — set environment for command execution

SYNOPSIS

env [-] [*name=value*]... [*command args*]

DESCRIPTION

env obtains the current *environment*, modifies it according to its arguments, then executes *command* with the modified environment. Arguments of the form *name=value* are merged into the inherited environment before the command is executed. The – flag option causes the inherited environment to be ignored completely, so that the command is executed with exactly the environment specified by the arguments.

If no command is specified, the resulting environment is printed, one name-value pair per line.

EXAMPLES

```
env XYZ=pdq sh
```

XYZ to the value pdq for the duration of the command, which here is a new shell, sh.

FILES

/bin/env

SEE ALSO

csh(1), ksh(1), sh(1), exec(2), profile(4), environ(5).

NAME

eqn, checkeq — format mathematical text for troff

SYNOPSIS

eqn [-d*xy*] [-p*n*] [-s*n*] [-f*n*] [-T*tty-type*] [-] [*file...*]

checkeq [*file...*]

DESCRIPTION

eqn is a troff(1) preprocessor for typesetting mathematical text on a phototypesetter. Normal usage is

eqn [*options*] *file* ... | troff [*options*] | [*typesetter*]

If you do not specify *file* (or if you specify - as the last argument), eqn reads the standard input. eqn prepares output for the output device you name in the -T option. Currently supported devices are -Taps (Autologic APS-5) and -Tpsc (POSTSCRIPT device). The default is -Tpsc.

A line beginning with .EQ marks the start of an equation; a line beginning with .EN marks the end of an equation; troff does not alter these lines, so they may be defined in macro packages to get centering, numbering, and so forth. You may also name two characters as delimiters; eqn treats subsequent text between delimiters as input. You may set delimiters to characters *x* and *y* with the command-line argument -d*xy* or (more commonly) with delim *xy* between .EQ and .EN. The left and right delimiters may be the same character, the dollar sign often being used as such a delimiter. Delimiters are turned off by delim off. All text that is neither between delimiters nor between .EQ and .EN is passed through untouched.

checkeq is a related program that reports missing or unbalanced delimiters and .EQ/.EN pairs.

Tokens within eqn are separated by spaces, tabs, newlines, braces, double quotes, tildes, and circumflexes. Braces ({ }) are used for grouping; generally, wherever a single character such as *x* may be used, then *x* enclosed in braces may be used instead. A tilde (~) represents a full space in the output; a circumflex (^) half as much.

For full details, see "eqn Reference" in *A/UX Text Processing Tools*.

FILES

/bin/checkeq
/bin/eqn

SEE ALSO

mm(1), mmt(1), nroff(1), tbl(1), troff(1), eqnchar(5), mm(5), mv(5).
"eqn Reference" in *A/UX Text Processing Tools.*

BUGS

To boldface digits, parentheses, and so forth, it is necessary to quote them, as in

```
bold "12.3"
```

When you use eqn with the mm macro package, displayed equations must appear only inside displays.
See also BUGS under troff(1).

NAME

e, ex, edit — text editor

SYNOPSIS

ex [–] [+*command*] [–r] [–R] [–t *tag*] [–v] [–x] *name...*

e *ex-arguments*

edit [–] [+*command*] [–r] [–R] [–t *tag*] [–v] [–x] *name...*

DESCRIPTION

ex is the root of a family of editors: edit, ex, and vi. The edit command set is a subset of the ex set, including just the basic commands, fewer magic characters, and line-based editing only. Display-based editing is the focus of vi.

If you have not used ed, or are a casual user, you will find that the editor edit is convenient for you. It avoids some of the complexities of ex, which is used mostly by systems programmers and those very familiar with ed.

e is synonymous with ex.

If you have a CRT terminal, you may wish to use a display-based editor; in this case see vi(1), which is a command that focuses on the display editing portion of ex.

The following flag options are recognized:

– Suppresses all interactive-user feedback, as when processing editor scripts in command files.

–v Equivalent to using vi rather than ex.

–t *tag*
Equivalent to an initial *tag* command, editing the file containing the *tag* and positioning the editor at its definition.

–r*file*
Used for recovery after an editor or system crash, retrieving the last saved version of the named file. If no file is specified, a list of saved files will be reported.

–R Read-only mode set, prevents accidentally overwriting the file.

+*command*
Indicates that the editor should begin by executing the specified command. If *command* is omitted, then it defaults to $, positioning the editor initially at the last line of the first file. Other useful commands here are scanning patterns of

the form */pat* or line numbers, for example, `+100` to start at line 100.

`-x` Encryption mode; a key is prompted for allowing creation or editing of an encrypted file. This encryption scheme is not secure.

name
Indicates files to be edited.

Modes

Command	Normal and initial state. Input prompted for by `:`. The kill character cancels partial command.
Insert	Entered by `a`, `i`, and `c`. Arbitrary text may be entered. Insert is normally terminated by line having only `.` on it, or abnormally with an interrupt.
Visual	Entered by `vi`, terminates with `Q` or `^\`.

Command Names and Abbreviations

abbrev	ab	next	n	undo	u
append	a	number	nu	unmap	unm
args	ar	preserve	pre	version	ve
change	c	print	p	visual	vi
copy	co	put	pu	write	w
delete	d	quit	q	xit	x
edit	e	read	re	yank	ya
file	f	recover	rec	window	z
global	g	rewind	rew	escape	!
insert	i	set	se	lshift	<
join	j	shell	sh	printnext	CR
list	l	source	so	resubst	&
map	map	stop	st	rshift	>
mark	ma	substitute	s	scroll	^D
move	m	unabbrev	una		

where CR=RETURN, and ^D=CONTROL-D.

Command Addresses

n	line *n*	*/pat*	next with *pat*
.	current	*?pat*	previous with *pat*
$	last	*x-n*	*n* before *x*

+	next	*x*,*y*	*x* through *y*
–	previous	´*x*	marked with *x*
+*n*	*n* forward	´´	previous context
%	1,$		

Initializing options

EXINIT	place `set`'s here in environment variable
$HOME/.exrc	editor initialization file
./.exrc	editor initialization file
set *x*	enable option
set no*x*	disable option
set *x*=*val*	give value *val*
set	show changed options
set all	show all options
set *x*?	show value of option *x*

Most useful options

autoindent	ai	supply indent
autowrite	aw	write before changing files
ignorecase	ic	in scanning
lisp	lisp	( ) { } are *s-exp*'s
list	list	print CONTROL-I for tab, $ at end
magic	magic	. [* special in patterns
number	nu	number lines
paragraphs	para	macro names which start . . .
redraw	redraw	simulate smart terminal
scroll	scroll	command mode lines
sections	sect	macro names
shiftwidth	sw	for < >, and input CONTROL-d
showmatch	sm	to) and } as typed
showmode	smd	show insert mode in vi
slowopen	slow	stop updates during insert
window	window	visual mode lines
wrapscan	ws	around end of buffer?
wrapmargin	wm	automatic line splitting

Scanning pattern formation

^	beginning of line
$	end of line
.	any character
\<	beginning of word
\>	end of word

[*str*]	any char in *str*
[↑*str*]	. . . not in *str*
[*x*–*y*]	. . . between *x* and *y*
*	any number of preceding

FILES

`/usr/bin/e`	
`/usr/bin/ex`	
`/usr/bin/edit`	
`/usr/lib/ex3.9strings`	error messages
`/usr/lib/ex3.9recover`	recover command
`/usr/lib/ex3.9preserve`	preserve command
`/usr/lib/*/*`	describes capabilities of terminals
`~/.exrc`	editor startup command file, user-created in home directory
`/tmp/EX`*nnnnn*	editor temporary
`/tmp/Rx`*nnnnn*	named buffer temporary
`/usr/preserve`	preservation directory
`/usr/lib/tags`	standard editor tag file

EXAMPLES

The command

```
ex text
```

would invoke the editor with the file named `text`.

SEE ALSO

`awk`(1), `ed`(1), `grep`(1), `sed`(1), `vi`(1), `curses`(3X), `term`(4), `terminfo`(4),
"Using `ex`" and "Using `vi`" in *A/UX Text Editing Tools*.

BUGS

The undo (`u`) command causes all marks to be lost on lines changed and then restored if the marked lines were changed.

The undo command never clears the "buffer modified" condition, that is, once the editor buffer has been modified, `ex` tells you that it is `[Modified]`, even if you undo the only modification.

The `z` command prints a number of logical rather than physical lines. More than a screen full of output may result if long lines are present.

File input/output errors don't print a name if the command line - option is used.

There is no easy way to do a single scan ignoring case.

The editor does not warn if text is placed in named buffers and not used before exiting the editor.

Null characters are discarded in input files, and cannot appear in resultant files.

NAME

expand, unexpand — expand tabs to spaces, and vice versa

SYNOPSIS

expand [-*tabstop*] [-*tab1*, *tab2*, ..., *tabn*] [*file*...]

unexpand [-a] [*file*...]

DESCRIPTION

expand reads the named files (or the standard input, if none are given) and writes on the the standard output with tabs changed into blanks. Backspace characters are preserved into the output and decrement the column count for tab calculations. expand is useful for preprocessing character files that contain tabs (before sorting, looking at specific columns, and so forth).

If a single numerical *tabstop* argument is given, then tabs are set *tabstop* spaces apart instead of the default 8. If multiple tabstops are given, then the tabs are set at those specific columns.

unexpand puts tabs back into the data from the standard input or the named files and writes the result on the standard output. By default, only leading blanks and tabs are reconverted to maximal strings of tabs. If the -a flag option is given, then tabs are inserted whenever they would compress the resultant file by replacing two or more characters.

FILES

/usr/ucb/expand
/usr/ucb/unexpand

See diction(1)

NAME

expr — evaluate arguments as an expression

SYNOPSIS

expr *arguments*

DESCRIPTION

expr evaluates its arguments as an expression and writes the result on the standard output. Terms of the expression must be separated by blanks. Characters special to the Bourne shell or Korn shell (sh (1) or ksh (1), respectively) must be escaped. (expr is replaced in the C shell (csh (1)) by @.) Note that 0 is returned to indicate a zero value, rather than the null string. Strings containing blanks or other special characters should be quoted. Integer-valued arguments may be preceded by a unary minus sign. Internally, integers are treated as 32-bit, 2's-complement numbers.

The operators and keywords are listed below. Characters that need to be escaped are preceded by \. The list is in order of increasing precedence, with equal precedence operators grouped within { } symbols.

expr \| *expr*
returns the first *expr* if it is neither null nor 0, otherwise returns the second *expr*.

expr \& *expr*
returns the first *expr* if neither *expr* is null or 0, otherwise returns 0.

expr { =, \>, \>=, \<, \<=, != } *expr*
returns the result of an integer comparison if both arguments are integers, otherwise returns the result of a lexical comparison.

expr { +, − } *expr*
addition or subtraction of integer-valued arguments.

expr { *, /, % } *expr*
multiplication, division, or remainder of the integer-valued arguments.

expr : *expr*
The matching operator : compares the first argument with the second argument which must be a regular expression; regular expression syntax is the same as that of ed(1), except that all patterns are *anchored* (i.e., begin with ^) and, there-

fore, ^ is not a special character, in that context. Normally, the matching operator returns the number of characters matched (0 on failure). Alternatively, the `\(...\)` pattern symbols can be used to return a portion of the first argument.

EXAMPLES

```
a=`expr $a + 1`
```

adds 1 to the shell variable a.

```
# 'For $a equal to either "/usr/abc/file"
# or just "file"'
expr $a : '.*/\(.*\)'  \ | $a
```

returns the last segment of a pathname (i.e., *file*). Watch out for / alone as an argument: *expr* will take it as the division operator (see BUGS below).

A better representation of above example:

```
expr //$a : '.*/\(.*\)'
```

the addition of the // characters eliminates any ambiguity about the division operator and simplifies the whole expression.

```
expr $VAR  :  '.*'
```

returns the number of characters in $VAR.

FILES

/bin/expr

SEE ALSO

csh(1), ed(1), ksh(1), sh(1).

EXIT CODE

As a side effect of expression evaluation, expr returns the following exit values:

0 if the expression is neither null nor 0
1 if the expression is null or 0
2 for invalid expressions

DIAGNOSTICS

`syntax error`	for operator/operand errors
`non-numeric argument`	if arithmetic is attempted on such a string

BUGS

After argument processing by the shell, `expr` cannot tell the difference between an operator and an operand except by the value. If `$a` is an =, the command:

```
expr  $a  =  '='
```

looks like:

```
expr  =  =  =
```

as the arguments are passed to `expr` (and they will all be taken as the = operator). The following works:

```
expr  X$a  =  X=
```

NAME

f77 — Fortran 77 compiler

SYNOPSIS

f77 [-l] [-66] [-c] [-C] [-E] [-f] [-F] [-g] [-I[24s]] [-m] [-o*output*] [-O] [-onetrip] [-p] [-R] [-S] [-u] [-U] [-w] *file* ...

DESCRIPTION

f77 is the Fortran 77 compiler; it accepts several types of *file* arguments:

Arguments whose names end with .f are taken to be Fortran 77 source programs; they are compiled and each object program is left in the current directory in a file whose name is that of the source, with .o substituted for .f.

Arguments whose names end with .r or .e are taken to be EFL source programs; these are first transformed by the EFL preprocessor, then compiled by f77, producing .o files.

In the same way, arguments whose names end with .c or .s are taken to be C or assembly source programs and are compiled or assembled, producing .o files.

The following flag options have the same meaning as in cc(1) (see ld(1) for link editor flag options):

-c	Suppress link editing and produce .o files for each source file.
-p	Prepare object files for profiling (see prof(1)).
-O	Invoke an object code optimizer.
-S	Compile the named programs and leave the assembler language output in corresponding files whose names are suffixed with .s. (No .o files are created.)
-o*output*	Name the final output file *output*, instead of a.out.
-f	In systems without floating-point hardware, use a version of f77 that handles floating-point constants and links the object program with the floating-point interpreter.
-g	Generate additional information needed for the use of sdb(1)

The following flag options are peculiar to f77:

-onetrip	Perform all DO loops at least once. (Fortran 77 DO loops are not performed at all if the upper limit is smaller than the lower limit.)
-1	Same as -onetrip.
-66	Suppress extensions which enhance Fortran 66 compatibility.
-C	Generate code for run-time subscript range-checking.
-I[24s]	Change the default size of integer variables (only valid on machines where the normal integer size is not equal to the size of a single precision real). -I2 causes all integers to be 2-byte quantities, -I4 (default) causes all integers to be 4-byte quantities, and -Is changes the default size of subscript expressions (only) from the size of an integer to 2 bytes.
-U	Do not "fold" cases. F77 is normally a no-case language (i.e., a is equal to A). The -U flag option causes f77 to treat upper and lower cases separately.
-u	Make the default type of a variable *undefined*, rather than using the default Fortran rules.
-w	Suppress all warning messages. If the flag option is -w66, only Fortran 66 compatibility warnings are suppressed.
-F	Apply EFL preprocessor to relevant files and put the result in files whose names have their suffix changed to .of. (No .o files are created.)
-m	Apply the M4 preprocessor to each EFL source file before transforming with the efl(1) processor.
-E	The remaining characters in the argument are used as an EFL flag argument whenever processing a .e file.

Other arguments are taken to be link editor flag option arguments, f77-compilable object programs (typically produced by an earlier run), or libraries of f77-compilable routines. These programs, together with the results of any compilations specified, are linked (in the order given) to produce an executable program with the default name a.out.

FILES

/usr/bin/f77	
file.[fresc]	input file
file.o	object file
a.out	linked output
./fort[*pid*].?	temporary
/usr/lib/f77comp	compiler
/lib/c2	optional optimizer
/usr/lib/libF77.a	intrinsic function library
/usr/lib/libI77.a	Fortran I/O library
/lib/libc.a	C library; see Section 3 in *A/UX Programmer's Reference.*

SEE ALSO

asa(1), cc(1), efl(1), fpr(1), fsplit(1), ld(1), m4(1), prof(1), sdb(1).
"f77 Reference" in *A/UX Programming Languages and Tools, Volume 1.*

DIAGNOSTICS

The diagnostics produced by f77 itself are self-explanatory. Occasional messages may be produced by the link editor ld(1).

NAME

`factor` — factor a number

SYNOPSIS

`factor` [*number*]

DESCRIPTION

`factor` prints the prime factors of its argument. When `factor` is invoked without an argument, it waits for a number to be typed in. If you type in a positive number less than `pow(2,56)`, it will factor the number and print its prime factors; each one is printed the proper number of times. Then it waits for another number. It exits if it encounters a zero or any non-numeric character.

If `factor` is invoked with an argument, it factors the number as above and then exits.

Maximum time to factor is proportional to $\sqrt{n}$ and occurs when n is prime or the square of a prime, where n is the *number* being factored. It takes 1 minute to factor a prime near 10^{11} on a 68020.

FILES

`/bin/factor`

DIAGNOSTICS

"`Ouch`" is echoed when input is out of range or is garbage.

See true(1)

NAME

fcnvt — convert a resource file to another format

SYNOPSIS

fcnvt [-i *input-format*] [-o *output-format*] [-f] *input-file output-file*
fcnvt [-i *input-format*] -s [-f] *input-file output-file*
fcnvt [-i *input-format*] -d [-f] *input-file output-file*
fcnvt [-i *input-format*] -t [-f] *input-file output-file*
fcnvt [-i *input-format*] -p [-f] *input-file output-file*
fcnvt [-i *input-format*] -b [-f] *input-file output-file*
fcnvt [-i *input-format*] -m [-f] *input-file output-file*

DESCRIPTION

fcnvt converts a file (*input-file*) from one file format to another (*output-file*). The command line options and their meanings are:

-i *input-format*
Specify the file format of the file to be converted. If an input-file format is not specified, the AppleSingle format is assumed. Supported formats are:

single
AppleSingle (see the -s flag option)

double
AppleDouble (see the -d flag option)

triple
Plain Triple (see the -t flag option)

pair
Plain Pair (see the -p flag option)

hex
BinHex 4.0 (see the -b flag option)

bin
MacBinary (see the -m flag option)

-o *output-format*
Specify the output-file format. Supported formats are listed above. If an output-file format is not specified, the input file is converted to AppleSingle format.

-s Create an AppleSingle-format output file. This format is the default. AppleSingle combines both the resource and data forks into a single A/UX® file and is most useful when the resource fork seldom or never changes or when a file has no

data fork. The use of AppleSingle format is very inefficient when both the resource and data forks are frequently expanded.

`-d` Create an AppleDouble-format output file. The AppleDouble format is the same as AppleSingle format except that the data fork is kept in a separate file. The resource file is prefixed with a `%` character.

`-t` Convert the input file into Plain Triple file format. This format is used by the `macget` and `macput` public-domain file-transfer programs. Three files are created with suffixes attached to help tell them apart. The files *output-file*`.info`, *output-file*`.data`, and *output-file*`.rsrc` contain identification information, the data fork, and the resource fork, respectively.

`-p` Convert the input file into Plain Pair file format. This option is the same as the `-t` option except that *output-file*`.info` is not created.

`-b` Create a BinHex 4.0-format output file. The input-file is encoded into ASCII characters, permitting ASCII transfer of a binary file.

`-m` Create a MacBinary-format output file. This format is commonly used when transferring files by using XMODEM, XMODEM7, Kermit, and CompuServe A or B protocols.

`-f` Allow `fcnvt` to overwrite an existing file with the same name as the new file. If you specify an output filename that is the same as an existing filename without specifying the `-f` flag option, `fcnvt` takes no action and returns an error message.

AppleSingle is particularly nice for executable Macintosh® object files. Directory listings look much cleaner because each Macintosh file maps to a single A/UX file with no `%` prefix.

If the initial transfer is made using a terminal emulator program, the file created is likely to be in a text-only format, BinHex 4.0 format, or MacBinary format, if not just a copy of the resource fork of the Macintosh file. In any of those cases, `fcnvt` can be used to convert the file to either AppleSingle or AppleDouble format, assuming you know their starting format.

Note that file transfers made using terminal emulators are likely to strip away the Macintosh type and creator attributes for the file. (Each of these attributes is one four-character string.) See settc(1) to restore those attributes, once you know what they are supposed to be.

FILES

/mac/bin/fcnvt

SEE ALSO

settc(1).

fgrep(1)

See grep(1)

NAME

file — determine file type

SYNOPSIS

file [-c] [-f *ffile*] [-m *mfile*] *arg* ...

DESCRIPTION

file performs a series of tests on each file named in its arguments in an attempt to classify it. If file appears to be ASCII, file examines the first 512 bytes and tries to guess its language. If a file is an executable a.out file, file will print the version stamp, provided it is greater than 0 (see ld(1)).

If the -f flag option is given, the next argument is taken to be a file containing the names of the files to be examined.

file uses the file /etc/magic to identify files that have some sort of "magic number," that is, any file containing a numeric or string constant that indicates its type. Commentary at the beginning of /etc/magic explains its format.

The -m flag option instructs file to use an alternate magic file.

The -c flag option causes file to check the magic file for format errors. This validation is not normally carried out for reasons of efficiency. No file typing is done under -c.

EXAMPLES

file *text-file program-file directory*

reports the file names and directory name, and whether the files are English text, nroff input, a C program, or whatever.

FILES

/bin/file
/etc/magic

SEE ALSO

ld(1), magic(4).

NAME

find — find files

SYNOPSIS

find *pathname-list expression*

DESCRIPTION

find recursively descends the directory hierarchy for each pathname in the *pathname-list* (that is, one or more pathnames) seeking files that match a boolean *expression* written in the primaries given below. find does not follow symbolic links. In the descriptions, the argument *n* is used as a decimal integer, where *+n* means more than *n*, *-n* means less than *n*, and *n* means exactly *n*.

-name *file* — True if *file* matches the current filename. Normal shell argument syntax may be used if escaped (watch out for [, ?, and *).

-perm *onum* — True if the file permission flags exactly match the octal number *onum* (see chmod(1)). If *onum* is prefixed by a minus sign, more flag bits (017777, see stat(2)) become significant and the flags are compared:

```
(flags & onum) == onum
```

-type *c* — True if the type of the file is *c*, where *c* is b, c, d, l, p, or f for block special file, character special file, directory, symbolic link, fifo (named pipe), or plain file, respectively.

-links *n* — True if the file has *n* links.

-user *uname* — True if the file belongs to the user *uname*. If *uname* is numeric and does not appear as a login name in the /etc/passwd file, it is taken as a user ID.

-group *gname* — True if the file belongs to the group *gname*. If *gname* is numeric and does not appear in the /etc/group file, it is taken as a group ID.

-size *n*[c] — True if the file is *n* blocks long (512 bytes per block). If *n* is followed by a c, the size is in characters.

`-atime` *n*	True if the file has been accessed in *n* days. The access time of directories in *pathname-list* is changed by `find` itself.
`-mtime` *n*	True if the file has been modified in *n* days.
`-ctime` *n*	True if the file has been changed in *n* days.
`-exec` *cmd*	True if the executed *cmd* returns a zero value as exit status. The end of *cmd* must be punctuated by an escaped semicolon. A command argument of the form `{}` is replaced by the current pathname.
`-ok` *cmd*	Like `-exec`, except that the generated command line is printed with a question mark first, and is executed only if the user responds by typing `y`.
`-print`	Always true; causes the current pathname to be printed.
`-cpio` *device*	Always true; write the current file on *device* in `cpio`(4) format (512-byte records).
`-newer` *file*	True if the current file has been modified more recently than the argument *file*.
`-depth`	Always true; causes descent of the directory hierarchy to be done so that all entries in a directory are acted on before the directory itself. This can be useful when `find` is used with `cpio`(1) to transfer files that are contained in directories without write permission.
(*expression*)	True if the parenthesized expression is true (parentheses are special to the shell and must be escaped).

The primaries may be combined using the following operators (in order of decreasing precedence):

1. The negation of a primary (`!` is the unary NOT operator).
2. Concatenation of primaries (the AND operation is implied by the juxtaposition of two primaries).
3. Alternation of primaries (`-o` is the OR operator).

EXAMPLES

The command

```
find / -perm 755 -exec ls "{}" ";"
```

will find all files, starting with the root directory, on which the permission levels have been set to 755 (see chmod(1)).

With -exec and a command such as ls, it is often necessary to escape the {} that stores the current pathname under investigation by putting it in double quotes. It is *always* necessary to escape the semicolon at the end of an -exec sequence.

Note again that it is also necessary to escape parentheses used for grouping primaries, by means of a backslash, as illustrated:

```
find \( -name a.out -o -name '*.o' \) -exec rm {} \;
```

removes all files named a.out or *.o.

FILES

/bin/find
/etc/passwd
/etc/group

SEE ALSO

chmod(1), cpio(1), csh(1), ksh(1), sh(1), xargs(1), ff(1M), stat(2), cpio(4), fs(4),
"Other Tools" in *A/UX Programming Languages and Tools, Volume 2*.

NAME

`finger` — user information lookup program

SYNOPSIS

`finger` [-b] [-f] [-h] [-i] [-l] [-m] [-p] [-q] [-s] [-w] [*name...*]

DESCRIPTION

By default, `finger` lists the login name, full name, terminal name and write status (as a "*" before the terminal name if write permission is denied), idle time, login time, and office location and phone number (if they are known) for each current A/UX® user. (Idle time is minutes if it is a single integer, hours and minutes if a ":" is present, or days and hours if a `d` is present.)

A longer format also exists and is used by `finger` whenever a list of people's names is given. Login names as well as actual user names (that is, first, last, or middle names) are accepted. This format is multiline and includes all the information described above as well as the user's home directory and login shell, any plan that the user has placed in the file `.plan` in his or her home directory, the project on which he or she is working and has placed in the file `.project`, in the home directory, and the user's home phone number.

`finger` may be used to look up users on a remote machine. To do so, specify *name* as *user@host*. If *user* is not supplied, a listing in standard format is provided.

FLAG OPTIONS

`finger` flag options include:

- `-b` In the short form, the option is the same as no option. In long form, suppress the printing of the directory and login shell.
- `-f` In the long form, same as no option. In short form, suppress printing of column headings.
- `-h` In the short form, same as no option. In long form, suppress the printing of the project.
- `-i` Always print the short form. This option matches *name* only on login and suppresses the name and office column.
- `-l` Force the output to the long format.
- `-m` Match arguments only on login name.

-p Suppress the printing of the `.plan` files.

-q Always print the short form. This option matches *name* only on login.

-s Force short output format.

-w In long form, same as no option. In short form, supress printing the name field.

FILES

`/usr/ucb/finger`	
`/etc/utmp`	who file
`/etc/passwd`	for users names, offices, ...
`/usr/adm/lastlog`	last login times
`~/.plan`	plans
`~/.project`	projects

SEE ALSO

`chfn`(1), `w`(1), `who`(1), `whoami`(1), `passwd`(4).

BUGS

Only the first line of the `.project` file is printed and case is ignored in the *name* argument.

NAME

fmt — simple text formatter

SYNOPSIS

fmt [*name...*]

DESCRIPTION

fmt is a simple text formatter which reads the concatenation of input files (or standard input if none are given) and produces on standard output a version of its input with lines as close to 72 characters long as possible. The spacing at the beginning of the input lines is preserved in the output, as are blank lines and interword spacing.

fmt is meant to format mail messages prior to sending, but may also be useful for other simple tasks. For instance, within visual mode of the ex editor (e.g. vi) the command

```
!}fmt
```

will reformat a paragraph, evening the lines.

FILES

/usr/ucb/fmt

SEE ALSO

nroff(1), mail(1), pr(1), troff(1), vi(1).

BUGS

The program was designed to be simple and fast; for more complex operations, the standard text processors are likely to be more appropriate.

NAME

fold — fold long lines for finite-width output device

SYNOPSIS

fold [–*width*] [*file...*]

DESCRIPTION

fold is a filter which will fold the contents of the specified files, or the standard input if no files are specified, breaking the lines to have maximum width *width*. The default for *width* is 80. *width* should be a multiple of 8 if tabs are present, or the tabs should be expanded using expand(1) before coming to fold.

FILES

/usr/ucb/fold

SEE ALSO

expand(1)

BUGS

If underlining is present it may be messed up by folding.

NAME

fpr — filter the output of Fortran programs for line printing

SYNOPSIS

```
fpr
```

DESCRIPTION

fpr is a filter that transforms files formatted according to Fortran's carriage control conventions into files formatted according to UNIX line printer conventions.

fpr copies its input onto its output, replacing the carriage control characters with characters that will produce the intended effects when printed using lpr(1). The first character of each line determines the vertical spacing as follows:

Character	Vertical Space Before Printing
Blank	One line
0	Two lines
1	To first line of next page
+	No advance

A blank line is treated as if its first character is a blank. A "blank" that appears as a carriage control character is deleted. A "0" is changed to a newline. A "1" is changed to a form feed. The effects of a "+" are simulated using backspaces.

EXAMPLES

```
a.out | fpr | lpr
fpr < f77.output | lpr
```

FILES

```
/usr/ucb/fpr
```

SEE ALSO

asa(1), f77(1).

BUGS

Results are undefined for input lines longer than 170 characters.

NAME

freq — report on character frequencies in a file

SYNOPSIS

freq [*file*...]

DESCRIPTION

freq counts occurrences of characters in the list of files specified on the command line. If no files are specified, the standard input is read.

EXAMPLES

```
freq filea
```

will list a count of each character that appears in filea.

FILES

/bin/freq

NAME

from — who is my mail from?

SYNOPSIS

from [-s *sender*] [*user*]

DESCRIPTION

from prints out the mail header lines in your mailbox file to show you who your mail is from. If *user* is specified, then *user*'s mailbox is examined instead of your own. If the -s flag option is given, then only headers for mail sent by *sender* are printed.

from works with mail and mailx.

FILES

/usr/ucb/from
/usr/bin/mailx
/usr/mail/*

SEE ALSO

biff(1), mail(1), mailx(1).

NAME

fsplit — split f77 or efl files

SYNOPSIS

fsplit [-e] [-f] [-s] *file* ...

DESCRIPTION

fsplit splits the named *files* into separate files, with one procedure per file. A procedure includes blockdata, function, main, program, and subroutine program segments. Procedure *X* is put in file *X*.f, *X*.r, or *X*.e depending on the language flag option chosen, with the following exceptions: main is put in the file MAIN.[efr] and unnamed blockdata segments in the files blockdata*N*.[efr] where *N* is a unique integer value for each file.

The following flag options are available:

- -e Input files are EFL.
- -f (default) Input files are F77.
- -s Strip f77 input lines to 72 or fewer characters with trailing blanks removed.

FILES

/bin/fsplit

SEE ALSO

csplit(1), efl(1), f77(1), split(1).

NAME

fstyp — report file-system type

SYNOPSIS

fstyp *file*

DESCRIPTION

fstyp reports the type of the file system on which *file* resides. If *file* is a device file, fstyp attempts to read a file-system superblock from the device. The file system must be one of the supported types listed in fstypes(4).

DIAGNOSTICS

If successful, fstyp prints to the standard output a message that indicates the file-system type.

FILES

/etc/fstyp
/etc/fs/*/fstyp

SEE ALSO

statfs(2), fstyp(3), fs(4), fstypes(4).

NAME

ftp — ARPANET file transfer program

SYNOPSIS

ftp [-v] [-d] [-i] [-n] [-g] [*host*]

DESCRIPTION

ftp is the user interface to the ARPANET standard File Transfer Protocol. The program allows a user to transfer files to and from a remote network site.

The client host with which ftp is to communicate may be specified on the command line. If this is done, ftp will immediately attempt to establish a connection to an FTP server on that host; otherwise, ftp will enter its command interpreter and await instructions from the user. When ftp is awaiting commands from the user, the prompt ftp> is provided to the user. The following commands are recognized by ftp:

! [*command* [*args*]]
: Invoke an interactive shell on the local machine. If there are arguments, the first is taken to be a command to execute directly, with the rest of the arguments as its arguments.

$ *macro-name* [*args*]
: Execute the macro *macro-name* that was defined with the macdef command. Arguments are passed to the macro unglobbed.

account [*passwd*]
: Supply a supplemental password required by a remote system for access to resources once a login has been completed successfully. If no argument is included, the user will be prompted for an account password in a nonechoing input mode.

append *local-file* [*remote-file*]
: Append a local file to a file on the remote machine. If *remote-file* is left unspecified, the local filename is used in naming the remote file, after being altered by any ntrans or nmap setting. File transfer uses the current settings for type, form (format), mode, and struct (structure).

ascii
: Set the file transfer type to network ASCII. This is the default type.

`bell`
Arrange that a bell be sounded after each file transfer command is completed.

`binary`
Set the file transfer `type` to support binary image transfer.

`bye`
Terminate the FTP session with the remote server and exit `ftp`. An end-of-file will also terminate the session and exit.

`case`
Toggle remote computer filename case mapping during `mget` commands. When `case` is on (default is off), remote computer filenames with all letters in upper case are written in the local directory with the letters mapped to lower case.

`cd` *remote-directory*
Change the working directory on the remote machine to *remote-directory*.

`cdup`
Change the remote machine working directory to the parent of the current remote machine working directory.

`close`
Terminate the FTP session with the remote server, and return to the command interpreter. Any defined macros are erased.

`cr` Toggle carriage return stripping during ASCII-type file retrieval. Records are denoted by a carriage return-linefeed sequence during ASCII-type file transfer. When `cr` is on (the default), carriage returns are stripped from this sequence to conform with the UNIX single-linefeed record delimiter. Records on non-UNIX remote systems may contain single linefeeds; when an ASCII-type transfer is made, these linefeeds may be distinguished from a record delimiter only when `cr` is off.

`delete` *remote-file*
Delete the file *remote-file* on the remote machine.

`debug` [*debug-value*]
Toggle debugging mode. If an optional *debug-value* is specified, it is used to set the debugging level. When debugging is on, `ftp` prints each command sent to the remote machine, preceded by the string `-->`.

`dir` [*remote-directory*] [*local-file*]
Print a listing of the directory contents in the directory, *remote-directory*, and, optionally, placing the output in *local-file*. If no directory is specified, the current working directory on the remote machine is used. If no local file is specified, or *local-file* is -, output comes to the terminal.

`disconnect`
A synonym for `close`.

`form` *format*
Set the file transfer `form` to *format*. The default format is `file`.

`get` *remote-file* [*local-file*]
Retrieve the *remote-file* and store it on the local machine. If the local filename is not specified, it is given the same name it has on the remote machine, subject to alteration by the current `case`, `ntrans`, and `nmap` settings. The current settings for `type`, `form`, `mode`, and `struct` (structure) are used while transferring the file.

`glob`
Toggle filename expansion for `mdelete`, `mget`, and `mput`. If globbing is turned off with `glob`, the filename arguments are taken literally and not expanded. Globbing for `mput` is done as in `csh`(1). For `mdelete` and `mget`, each remote filename is expanded separately on the remote machine and the lists are not merged. Expansion of a directory name is likely to be different from expansion of the name of an ordinary file: the exact result depends on the foreign operating system and ftp server, and can be previewed by issuing:

`mls` *remote-files* -

Note: `mget` and `mput` are not meant to transfer entire directory subtrees of files. That can be done by transferring a `tar`(1) archive of the subtree (in binary mode).

`hash`
Toggle hash-sign (#) printing for each data block transferred. The size of a data block is 1024 bytes.

`help` [*command*]
Print an informative message about the meaning of *com-*

mand. If no argument is given, `ftp` prints a list of the known commands.

`lcd` [*directory*]
Change the working directory on the local machine. If no *directory* is specified, the user's home directory is used.

`ls` [*remote-directory*] [*local-file*]
Print an abbreviated listing of the contents of a directory on the remote machine. If *remote-directory* is left unspecified, the current working directory is used. If no local file is specified, or if *local-file* is -, the output is sent to the terminal.

`macdef` *macro-name*
Define a macro. Subsequent lines are stored as the macro *macro-name*; a null line (consecutive newline characters in a file or carriage returns from the terminal) terminates macro input mode. There is a limit of 16 macros and 4096 total characters in all defined macros. Macros remain defined until a `close` command is executed. The macro processor interprets `$` and \ as special characters. A `$` followed by a number (or numbers) is replaced by the corresponding argument on the macro invocation command line. A `$` followed by an `i` signals that macro processor that the executing macro is to be looped. On the first pass, `$i` is replaced by the first argument on the macro invocation command line, on the second pass, it is replaced by the second argument, and so on. A \ followed by any character is replaced by that character. Use the \ to prevent special treatment of the `$`.

`mdelete` [*remote-files*]
Delete the *remote-files* on the remote machine.

`mdir` *remote-files local-file*
Like `dir`, except multiple remote files may be specified. If interactive prompting is on, `ftp` will prompt the user to verify that the last argument is indeed the target local file for receiving `mdir` output.

`mget` *remote-files*
Expand the *remote-files* on the remote machine and do a `get` for each filename thus produced. See `glob` for details on the filename expansion. Resulting filenames will then be processed according to `case`, `ntrans`, and `nmap` settings. Files are transferred into the local working directory, which

can be changed with the command

`lcd` *directory*

and new local directories can be created with

`! mkdir` *directory*

`mkdir` *directory-name*
Make a directory on the remote machine.

`mls` *remote-files local-file*
Like `ls`, except multiple remote files may be specified. If interactive prompting is on, `ftp` will prompt the user to verify that the last argument is indeed the target local file for receiving `mls` output.

`mode` [*mode-name*]
Set the file transfer *mode* to *mode-name*. The default mode is "stream" mode.

`mput` *local-files*
Expand wild cards in the list of local files given as arguments and do a `put` for each file in the resulting list. See `glob` for details of filename expansion. Resulting filenames will then be processed according to `ntrans` and `nmap` settings.

`nmap` [*inpattern outpattern*]
Set or unset the filename mapping mechanism. If no arguments are specified, the filename mapping mechanism is unset. If arguments are specified, remote filenames are mapped during `mput` commands and `put` commands issued without a specified remote target filename. If arguments are specified, local filenames are mapped during `mget` commands and `get` commands issued without a specified local target filename. This command is useful when connecting to a non-UNIX remote computer with different file naming conventions or practices. The mapping follows the pattern set by *inpattern* and *outpattern*. *inpattern* is a template for incoming filenames (which may have already been processed according to the `ntrans` and `case` settings). Variable templating is accomplished by including the sequences `$1`, `$2`,...,`$9` in *inpattern*. Use `\` to prevent this special treatment of the `$` character. All other characters are treated literally, and are used to determine the `nmap` *inpattern* variable values. For example, given *inpattern* `$1.$2` and the remote filename `mydata.data`, `$1` would have the value

mydata, and $2 would have the value data. The *outpattern* determines the resulting mapped filename. The sequences $1, $2,...,$9 are replaced by any value resulting from the *inpattern* template. The sequence $0 is replaced by the original filename. Additionally, the sequence [*seq1*,*seq2*] is replaced by *seq1* if *seq1* is not a null string; otherwise, it is replaced by *seq2*. For example, the command:

```
nmap $1.$2.$3 [$1,$2].[$2,file]
```

would yield the output filename myfile.data for input filenames myfile.data and myfile.data.old, myfile.file for the input filename myfile, and myfile.myfile for the input filename .myfile. Spaces may be included in *outpattern*, as in the example:

```
nmap $1 | sed "s/  *$//" > $1
```

Use the \ character to prevent special treatment of the $, [,], and , characters.

ntrans [*inchars* [*outchars*]]
: Set or unset the filename character translation mechanism. If no arguments are specified, the filename character translation mechanism is unset. If arguments are specified, characters in remote filenames are translated during mput commands and put commands issued without a specified remote target filename. If arguments are specified, characters in local filenames are translated during mget commands and get commands issued without a specified local target filename. This command is useful when connecting to a non-UNIX remote computer with different file naming conventions or practices. Characters in a filename matching a character in *inchars* are replaced with the corresponding character in *outchars*. If the character's position in *inchars* is longer than the length of *outchars*, the character is deleted from the filename.

open *host* [*port*]
: Establish a connection to the specified *host* FTP server. An optional port number may be supplied, in which case, ftp will attempt to contact an FTP server at that port. If the auto-login option is on (default), ftp will also attempt automatically to log the user in to the FTP server (see below).

`prompt`
Toggle interactive prompting. Interactive prompting occurs during multiple file transfers to allow the user selectively to retrieve or store files. If prompting is turned off (default is on), any `mget` or `mput` will transfer all files, and any `mdelete` will delete all files.

`proxy` *ftp-command*
Execute an ftp command on a secondary control connection. This command allows simultaneous connection to two remote ftp servers for transferring files between the two servers. The first `proxy` command should be an `open`, to establish the secondary control connection. Enter the command `proxy ?` to see other ftp commands executable on the secondary connection. The following commands behave differently when prefaced by `proxy`: `open` will not define new macros during the autologin process, `close` will not erase existing macro definitions, `get` and `mget` transfer files from the host on the primary control connection to the host on the secondary control connection, and `put`, `mput`, and `append` transfer files from the host on the secondary control connection to the host on the primary control connection. Third party file transfers depend upon support of the ftp protocol PASV command by the server on the secondary control connection.

`put` *local-file* [*remote-file*]
Store a local file on the remote machine. If *remote-file* is left unspecified, the local filename is used, after processing according to any `ntrans` or `nmap` settings in naming the remote file. File transfer uses the current settings for `type`, `form` (format), `mode`, and `struct` (structure).

`pwd`
Print the name of the current working directory on the remote machine.

`quit`
A synonym for `bye`.

`quote` *arg1 arg2 ...*
The arguments specified are sent, verbatim, to the remote FTP server.

`recv` *remote-file* [*local-file*]
A synonym for `get`.

`remotehelp` [*command-name*]
Request help from the remote FTP server. If a *command-name* is specified, it is supplied to the server as well.

`rename` [*from*] [*to*]
Rename the file *from* on the remote machine, to the file *to*.

`reset`
Clear reply queue. This command resynchronizes command/reply sequencing with the remote ftp server. Resynchronization may be neccesary following a violation of the ftp protocol by the remote server.

`rmdir` *directory-name*
Delete a directory on the remote machine.

`runique`
Toggle storing of files on the local system with unique filenames. If a file already exists with a name equal to the target local filename for a `get` or `mget` command, a `.1` is appended to the name. If the resulting name matches another existing file, a `.2` is appended to the original name. If this process continues up to `.99`, an error message is printed, and the transfer does not take place. The generated unique filename will be reported. Note that `runique` will not affect local files generated from a shell command (see below). The default value is off.

`send` *local-file* [*remote-file*]
A synonym for `put`.

`sendport`
Toggle the use of PORT commands. By default, `ftp` will attempt to use a PORT command when establishing a connection for each data transfer. The use of PORT commands can prevent delays when performing multiple file transfers. If the PORT command fails, `ftp` will use the default data port. When the use of PORT commands is disabled, no attempt will be made to use PORT commands for each data transfer. This is useful for certain FTP implementations which do ignore PORT commands but, incorrectly, indicate they've been accepted.

`status`
Show the current status of `ftp`.

`struct` [*struct-name*]
Set the file transfer *structure* to *struct-name*. By default, "stream" structure is used.

`sunique`
Toggle storing of files on remote machine under unique filenames. Remote ftp server must support ftp protocol STOU command for successful completion. The remote server will report unique name. Default value is off.

`tenex`
Set the file transfer `type` to that needed to talk to TENEX machines.

`trace`
Toggle packet tracing.

`type` [*type-name*]
Set the file transfer `type` to *type-name*. If no type is specified, the current type is printed. The default type is network ASCII.

`user` *user-name* [*password*] [*account*]
Identify yourself to the remote FTP server. If the password is not specified, and the server requires it, `ftp` will prompt the user for it (after disabling local echo). If an account field is not specified, and the FTP server requires it, the user will be prompted for it. If an account field is specified, an account command will be relayed to the remote server after the login sequence is completed if the remote server did not require it for logging in. Unless `ftp` is invoked with "autologin" disabled, this process is done automatically on initial connection to the FTP server.

`verbose`
Toggle verbose mode. In verbose mode, all responses from the FTP server are displayed to the user. In addition, if verbose is on, when a file transfer completes, statistics regarding the efficiency of the transfer are reported. By default, verbose is on.

`?` [*command*]
A synonym for `help`.

Command arguments which have embedded spaces may be quoted with quote (") marks.

ABORTING A FILE TRANSFER

To abort a file transfer, use the terminal interrupt key (usually CONTROL-C). Sending transfers will be halted immediately. Receiving transfers will be halted by sending an ftp protocol ABOR command to the remote server, and discarding any further data received. The speed at which this is accomplished depends upon the remote server's support for ABOR processing. If the remote server does not support the ABOR command, an `ftp>` prompt will not appear until the remote server has completed sending the requested file.

The terminal interrupt key sequence will be ignored when `ftp` has completed any local processing and is awaiting a reply from the remote server. A long delay in this mode may result from the ABOR processing described above, or from unexpected behavior by the remote server, including violations of the ftp protocol. If the delay results from unexpected remote server behavior, the local `ftp` program must be killed by hand.

FILE NAMING CONVENTIONS

Files specified as arguments to `ftp` commands are processed according to the following rules:

1) If the filename - is specified, the standard input (for reading) or standard output (for writing) is used.

2) If the first character of the filename is |, the remainder of the argument is interpreted as a shell command. `ftp` then forks a shell, using `popen(3)` with the argument supplied, and reads (writes) from the stdout (stdin). If the shell command includes spaces, the argument must be quoted; for example,

```
"| ls -lt"
```

A particularly useful example of this mechanism is:

```
dir . | more
```

3) Failing the above checks, if "globbing" is enabled, local filenames are expanded according to the rules used in the `csh(1)`; compare the `glob` command. If the `ftp` command expects a single local file (for example, `put`), only the first filename generated by the "globbing" operation is used.

4) For `mget` commands and `get` commands with unspecified local filenames, the local filename is the remote filename, which may be altered by a `case`, `ntrans`, or `nmap` setting.

The resulting filename may then be altered if `runique` is on.

5) For `mput` commands and `put` commands with unspecified remote filenames, the remote filename is the local filename, which may be altered by a `ntrans` or `nmap` setting. The resulting filename may then be altered by the remote server if `sunique` is on.

FILE TRANSFER PARAMETERS

The FTP specification specifies many parameters which may affect a file transfer. The *type* may be one of `ascii`, `image` (binary), `ebcdic`, and `local byte size` (for PDP-10's and PDP-20's, mostly). `ftp` supports the `ascii` and `image` types of file transfer, plus local byte size 8 for `tenex` mode transfers.

`ftp` supports only the default values for the remaining file transfer parameters: `mode`, `form`, and `struct`.

FLAG OPTIONS

Flag options may be specified at the command line, or to the command interpreter.

The `-v` (verbose on) flag option forces `ftp` to show all responses from the remote server, as well as report on data transfer statistics.

The `-n` flag option restrains `ftp` from attempting "autologin" upon initial connection. If autologin is enabled, `ftp` will check the `.netrc` (see below) file in the user's home directory for an entry describing an account on the remote machine. If no entry exists, `ftp` will prompt for the remote machine login name (default is the user identity on the local machine), and, if necessary, prompt for a password and an account with which to login.

The `-i` flag option turns off interactive prompting during multiple file transfers.

The `-d` flag option enables debugging.

The `-g` flag option disables filename globbing.

THE `.netrc` FILE

The `.netrc` file contains login and initialization information used by the autologin process. It resides in the user's home directory. The following tokens are recognized; they may be separated by spaces, tabs, or newlines:

`machine` *name*

Identify a remote machine name. The autologin process

searches the `.netrc` file for a `machine` token that matches the remote machine specified on the `ftp` command line or as an `open` command argument. Once a match is made, the subsequent `.netrc` tokens are processed, stopping when the end-of-file is reached or another `machine` token is encountered.

`login` *name*
Identify a user on the remote machine. If this token is present, the autologin process will initiate a login using the specified name.

`password` *string*
Supply a password. If this token is present, the autologin process will supply the specified string if the remote server requires a password as part of the login process. Note that if this token is present in the `.netrc` file, `ftp` will abort the autologin process if the `.netrc` is readable by anyone besides the user.

`account` *string*
Supply an additional account password. If this token is present, the autologin process will supply the specified string if the remote server requires an additional account password, or the autologin process will initiate an `acct` command, if it does not.

`macdef` *name*
Define a macro. This token functions like the `ftp macdef` command functions. A macro is defined with the specified name; its contents begin with the next `.netrc` line and continue until a null line (consecutive newline characters) is encountered. If a macro named `init` is defined, it is automatically executed as the last step in the autologin process.

EXAMPLES

The first example illustrates a simple `ftp` connection and file transfer from a remote machine to the local machine. (Long output lines have been folded for the sake of readability.)

```
[2]% ftp printms
Connected to printms.
220 printms FTP server (Version 4.109 Fri Nov 20
                    07:43:57 PST 1987) ready.
Name (printms:tim):
331 Password required for tim.
```

```
Password:
230 User tim logged in.
ftp> get tmac.an
200 PORT command successful.
150 Opening data connection for tmac.an
                    (89.0.0.33,1205) (13366 bytes).
226 Transfer complete.
local: tmac.an remote: tmac.an
13922 bytes received in 0.69 seconds (20 Kbytes/s)
ftp> quit
221 Goodbye.
[3]%
```

The second example illustrates an `ftp` connection and a file transfer from the local machine to the remote.

```
[4]% ftp printms
Connected to printms.
220 printms FTP server (Version 4.109 Fri Nov 20
                    07:43:57 PST 1987) ready.
Name (printms:tim):
331 Password required for tim.
Password:
230 User tim logged in.
ftp> put tmac.an
200 PORT command successful.
150 Opening data connection for tmac.an
                    (89.0.0.33,1209).
226 Transfer complete.
local: tmac.an remote: tmac.an
13922 bytes sent in 0.83 seconds (16 Kbytes/s)
ftp> quit
221 Goodbye.
[5]%
```

The third example illustrates moving around in the remote file system and listing the contents of several directories.

```
[8]% ftp printms
Connected to printms.
220 printms FTP server (Version 4.109 Fri Nov 20
                    07:43:57 PST 1987) ready.
Name (printms:tim):
331 Password required for tim.
Password:
230 User tim logged in.
ftp> ls
200 PORT command successful.
```

```
150 Opening data connection for /bin/ls
                          (89.0.0.33,1212) (0 bytes).
OUT
cutmks
tmac.ap
tmac.ptx
tmac.syn
tmac.toc
226 Transfer complete.
76 bytes received in 1.2 seconds (0.13 Kbytes/s)
ftp> cd OUT
250 CWD command successful.
ftp> ls
200 PORT command successful.
150 Opening data connection for /bin/ls
                       (89.0.0.33,1213) (0 bytes).
junk
226 Transfer complete.
4 bytes received in 0.058 seconds (3.9 Kbytes/s)
ftp> close
221 Goodbye.
ftp> quit
```

FILES

`/usr/spool/ftp`

SEE ALSO

cu(1C), tip(1C).
"Using B-NET" in *A/UX Communications User's Guide*.

BUGS

Correct execution of many commands depends upon proper behavior by the remote server.

An error in the treatment of returns in the 4.2 BSD UNIX ASCII-mode transfer code has been corrected. This correction may result in incorrect transfers of binary files to and from 4.2 BSD servers using the `ascii` type. You may avoid this problem by using the `image` (binary) type.

When the verbose mode (`-v` flag option) is turned off, `ftp` does not echo responses from the remote server. This includes the response to the request `pwd`. Beware of this.

NAME

get — get 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*...

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).

Each of the keyletter arguments is explained in the following as though only one SCCS file is to be processed, but the effects of any keyletter argument apply independently to each named file.

-r*SID* 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.

-c*cutoff* 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). The -e keyletter used in a get for a particular version (SID) of the SCCS file 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 keyletter is accidentally ruined in the process of editing it, it may be regenerated by re-executing the get command with the -k keyletter in place of the -e keyletter.

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.

-b Used with the -e keyletter to indicate that the new delta should have an SID in a new branch as shown in Table 1. This keyletter is ignored if the b flag 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.

-i*list* 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.

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

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

`-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*.

`-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` keyletter is used, in which case it disappears.

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

`-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` keyletters are used, the format is: `%M%` value, followed by a horizontal tab, followed by the `-m` keyletter generated format.

`-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.

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

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

-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 keyletters are specified, the -a keyletter is used. Care should be taken when using the -a keyletter in conjunction with the -e keyletter, as the SID of the delta to be created may not be what one expects. The -r keyletter can be used with the -a and -e keyletters to control the naming of the SID of the delta to be created.

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 keyletter 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 keyletter is used included deltas are listed following the notation Included; if the -x keyletter 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 keyletter is effective only if the b flag (see `admin(1)`) is present in the file. An entry of – means *ir-*

relevant.

‡ 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

```
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` keyletter 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` keyletter 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 keyletter).

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` keyletter along to `delta`. Its contents are also used to prevent a subsequent execution of `get` with an `-e` keyletter 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` keyletter argument if it was present, followed by a blank and the `-x` keyletter 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.

FILES

`/usr/bin/get`

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).

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

DIAGNOSTICS

Use `help`(1) for explanations.

BUGS

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` keyletter is used.

NAME

getopt — parse command options

SYNOPSIS

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

DESCRIPTION

getopt returns *input-string* with additional separators to help distinguish any flag options, any arguments associated with the flag options, and any arguments not associated with the flag 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 flag option is twofold or threefold:

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

Each *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 ...
```

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 flags and flag 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 flag option in this example, the output of `getopt` includes hyphens in front of both `a` and `o`.

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 `$*` in order to request the parsing of command-line arguments, any quotes that may be present in the command line are automatically stripped by the shell. In such cases you need to use a reference to the unstripped version of the command-line arguments, `$@`. 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 evalu-

ate 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
fi
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"
```

FILES

`/bin/getopt`

SEE ALSO

`csh`(1), `ksh`(1), `sh`(1), `getopt`(3C).

DIAGNOSTICS

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

NAME

grap — pic preprocessor for drawing graphs

SYNOPSIS

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

DESCRIPTION

grap is a language for typesetting graphs. It is also the name of a preprocessor that feeds input to pic(1). 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.

FLAG OPTIONS

-T 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.

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

FILES

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

SEE ALSO

pic(1).

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

NAME

graph — draw 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*]]

DESCRIPTION

graph with no flag 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(1G) 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.

The following flag options are recognized, each as a separate argument:

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

-b
: Break (disconnect) the graph after each label in the input.

-c *label*
: Character string given by *label* is the default label for each point.

-g [*style*]
: *style* is grid style, where 0=no grid, 1=frame with ticks, 2=full grid (default).

-h *hspace*
: *hspace* is the fraction of the space for height.

-l *title*
: *title* is the label for the graph.

-m [*mode*]
: *mode* is 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*
: *rspace* is the fraction of the space to move right before plotting.

`-s` Save screen, don't erase before plotting.

`-t` Transpose horizontal and vertical axes.

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

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

`-x [l] [`*a*`] [`*b*`] [`*c*`]`
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*`]`
Similarly for the yaxis.

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

FILES

`/usr/bin/graph`

SEE ALSO

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

BUGS

`graph` 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.

NAME

greek — filter text for vintage display devices

SYNOPSIS

greek [-T*terminal*]

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)). The following *terminals* are recognized currently:

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.

EXAMPLES

The command

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

reinterprets the extended character set on a Tektronix 4014 terminal.

FILES

/usr/bin/greek
/usr/bin/300
/usr/bin/300s
/usr/bin/4014
/usr/bin/450

SEE ALSO

300(1), 4014(1), 450(1), eqn(1), mm(1), nroff(1), tplot(1G), term(4), environ(5), greek(5).

NAME

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

SYNOPSIS

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

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

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

DESCRIPTION

Commands of the grep family search the input *files* (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output. grep patterns are limited regular expressions in the style of ed(1); it uses a compact nondeterministic algorithm. egrep patterns are full regular expressions; it uses a fast deterministic algorithm that sometimes needs exponential space. fgrep patterns are fixed *strings*; it is fast and compact. The following flag options are recognized:

-v	All lines but those matching are printed.
-x	(Exact) only lines matched in their entirety are printed (fgrep only).
-c	Only a count of matching lines is printed.
-i	Ignore upper/lowercase distinction during comparisons.
-l	Only the names of files with matching lines are listed (once), separated by newlines.
-n	Each line is preceded by its relative line number in the file.
-b	Each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.
-s	The error messages produced for nonexistent or unreadable files are suppressed (grep only).
-e *expression*	Same as a simple *expression* argument, but useful when the *expression* begins with a - (does not work with grep).

`-f` *file* The regular *expression* (`egrep`) or *strings* list (`fgrep`) is taken from the *file*.

In all cases, the file name 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 quotes `'...'`.

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

`egrep` 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

```
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`.

FILES

```
/bin/grep
/bin/egrep
/bin/fgrep
```

SEE ALSO

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

DIAGNOSTICS

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

BUGS

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`.)

`egrep` 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.

NAME

groups — show group memberships

SYNOPSIS

groups [*user*]

DESCRIPTION

The groups command 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.

SEE ALSO

setgroups(2).

FILES

/usr/bin/groups
/etc/passwd
/etc/group

BUGS

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

See spell(1)

hashmake(1)

See spell(1)

NAME

head — give first few lines

SYNOPSIS

head [*–count*] [*file...*]

DESCRIPTION

This filter gives the first *count* lines of each of the specified files, or of the standard input. If *count* is omitted it defaults to 10.

EXAMPLES

```
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 head lines listed, if more than one file has been specified.

FILES

/bin/head

SEE ALSO

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

NAME

help — ask for help in using SCCS

SYNOPSIS

help [*args*]

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. If no arguments are given, help will prompt for one.

The arguments may be either message numbers (which normally appear in parentheses following messages) or command names, 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 (e.g., 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 (e.g., 26)

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

When all else fails, try help stuck.

EXAMPLES

```
help he2
```

prints the message for error number he2.

FILES

/usr/bin/help	
/usr/lib/help	directory containing files of message text.
/usr/lib/help/helploc	file containing locations of help files not in /usr/lib/help. This file is created by the user for user-defined help messages.
/usr/lib/help/lib/help2	

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).

DIAGNOSTICS

Use help(1) for explanations.

NAME

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

SYNOPSIS

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

DESCRIPTION

hex translates executable object files into ASCII formats suitable for Motorola S-record downloading. The following flag options determine locations:

-f	The file specified is to be shipped as is without treating it as an object file.
-l	Output "Loading at" message.
-n#	Number of characters to output per record. # is a decimal number.
-r	Output a carriage return at the end of each S-record (instead of a newline).
-s0	Output a leading s0 record.
-s2	S2 records only (no s1 records are produced).
-ns8	Do not output a trailing s8 (s9) record.
+*saddr*	Starting load address (in hex).
ifile	File to be downloaded. The file's starting address is used if *saddr* is not present.

EXAMPLES

```
hex objfile
```

where objfile is the object file to be downloaded.

FILES

/usr/bin/hex

SEE ALSO

as(1), od(1), rcvhex(1), a.out(4).

NAME

hostid — set or display the identifier of the current host system

SYNOPSIS

hostid [*identifier*]

DESCRIPTION

The hostid command 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 the 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

SEE ALSO

gethostid(2N).

NAME

hostname — set or display the name of the current host system

SYNOPSIS

hostname [*nameofhost*]

DESCRIPTION

The hostname command 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

SEE ALSO

gethostname(2N).

NAME

hyphen — find hyphenated words

SYNOPSIS

hyphen [*file*...]

DESCRIPTION

hyphen finds all the hyphenated words ending lines in *file*s and prints them on the standard output. If no arguments are given or if hyphen encounters –, 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
```

FILES

/usr/bin/hyphen

SEE ALSO

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

BUGS

hyphen can't cope with hyphenated italics (or underlined words); it frequently will either miss them altogether or mishandle them. hyphen occasionally gets confused but with no ill effects other than spurious extra output.

NAME

`id` — display 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 and `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
/etc/passwd
```

SEE ALSO

`logname`(1), `whoami`(1), `getuid`(2).

NAME

ident — display RCS keywords and their values

SYNOPSIS

ident *files*

DESCRIPTION

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

```
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.

ident 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 $
```

DISCLAIMER

This reference manual entry describes a utility that Apple understands to have been released into the public domain by its author or authors. Apple has included this public domain utility for your convenience. Use it at your own discretion. Often the source code can be obtained if additional requirements are met, such as the purchase of a site license from an author or institution.

IDENTIFICATION

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).

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` — indent and format C program source

SYNOPSIS

`indent` *input* [*output*] [*flags*]

DESCRIPTION

`indent` is intended primarily as a C program formatter. Specifically, `indent` will:

indent code lines
align comments
insert spaces around operators where necessary
break up declaration lists as in `int a,b,c;`.

`indent` 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 *flags* which can be specified follow. They may appear before or after the file names. If the *output* file is omitted, the formatted file will be written back into *input* and a "backup" copy of *input* will be written in the current directory. If *input* is named `/blah/blah/file`, the backup file will be named `.Bfile`. If *output* is specified, `indent` checks to make sure it is different from *input*.

The following flag options may be used to control the formatting style imposed by `indent`.

`-l`*nnn* Maximum length of an output line. The default is 75.

`-c`*nnn* The column in which comments will start. The default is 33.

`-cd`*nnn* The column in which comments on declarations will start. The default is for these comments to start in the same column as other comments.

`-i`*nnn* The number of spaces for one indentation level. The default is 4.

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

-v,-nv -v turns on "verbose" mode, -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.

-bc,-nbc If -bc is specified, then a newline will be forced after each comma in a declaration. -nbc will turn off this flag option. The default is -bc.

-d*nnn* This flag option 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 below.

-br,-bl Specifying -bl will cause complex statements to be lined up like this:

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

Specifying -br (the default) will make them look like this:

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

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

`indent` 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

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

Box comments. `indent` 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. These 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.

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

ed.

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 `-cnnn` command line parameter. Otherwise, the comment will be started at *nnn* indentation levels less than where code is currently being placed, where *nnn* is specified by the `-dnnn` command line parameter. (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.

DIAGNOSTICS

Diagnostic error messages, mostly to tell that a text line has been broken or is too long for the output line.

FILES

`/usr/ucb/indent`	
`~/.indent.pro`	profile file

SEE ALSO

`cb`(1).

BUGS

Does not know how to format "long" declarations.

NAME

indxbib — build inverted index for a bibliography

SYNOPSIS

indxbib *database...*

DESCRIPTION

indxbib makes an inverted index to the named *databases* (or files) for use by lookbib(1) and refer(1). 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 %.

indxbib 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.

FILES

/usr/ucb/indxbib

x.ia, *x*.ib, *x*.ic, where *x* is the first argument, or if these are not present, then *x*.ig.

SEE ALSO

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

BUGS

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

NAME

ipcrm — remove interprocess communications facilities

SYNOPSIS

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

DESCRIPTION

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

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

-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.

-s *semid* removes the semaphore identifier *semid* from the system and destroys the set of semaphores 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.

-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.

-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.

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

SEE ALSO

ipcs(1), msgctl(2), msgget(2), msgop(2), semctl(2), semget(2), semop(2), shmctl(2), shmget(2), shmop(2).

NAME

ipcs — report interprocess communication facilities status

SYNOPSIS

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

DESCRIPTION

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

-q Print information about active message queues.
-m Print information about active shared memory segments.
-s Print information about active semaphores.

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

-b Print 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 Print creator's login name and group name. See below.
-o Print 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 Print process number information (process ID of last process to send a message and 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) See below.
-t Print 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.) See below.
-a Use all print flag options. (This is a shorthand notation for -b, -c, -o, -p, and -t.)

–C *corefile*
Use the file *corefile* in place of /dev/kmem.

–N *namelist*
The argument will be taken as the name of an alternate *namelist* (/unix is the default).

The column headings and the meaning of the columns in an ipcs listing are given below; the letters in parentheses indicate the flag options that cause the corresponding heading to appear; *all* means that the heading always appears. Note that these flag 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 first two 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 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 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 group of the owner 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 group of the creator 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.

FILES

/bin/ipcs	
/unix	system namelist
/dev/kmem	memory
/etc/passwd	user names
/etc/group	group names

SEE ALSO

ipcrm(1), msgop(2), semop(2), shmop(2).

BUGS

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

See mactoiso(1)

NAME

iw2 — Apple ImageWriter print filter

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...*]

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 options set.

iw2 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. If no files are specified, the standard input is assumed. The various features of the Apple Imagewriter II may be specified by the following flag options.

-a *dotspace* Add 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 Print boldface text. Each dot of the character is printed twice with a small shift of position.

-c *color* Print 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` Actually selects the magenta color ribbon band. You can specify this color by `magenta` as well.
`blue` Actually selects the cyan color ribbon band. You can specify this color by `cyan` as well.
`orange` Orange will be printed by overprinting yellow and magenta.
`green` Green will be printed by overprinting yellow and cyan.
`purple` Purple will be printed by overprinting magenta and cyan.

`-d` Print double-width characters. Each character is printed with two dots for every one normally printed.

`-D` *udcfile* This works just like the `-U` flag option, except that the *udcfile* filename is prefixed with the directory pathname `/usr/lib/iw2/` first. (See the `-U` flag option later in this section).

`-f` An initial formfeed is output before any files are printed. Generally used with the Apple Imagewriter II sheetfeeder.

`-h` Print 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* Select 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* Select 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`	Select the American language font.
`italian`	Select the Italian language font.
`danish`	Select the Danish language font.
`british`	Select the British language font.
`german`	Select the German language font.
`swedish`	Select the Swedish language font.
`french`	Select the French language font.
`spanish`	Select the Spanish language font.

`-m` *margin* Specify 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* Specify page length. This is the page length in inches, integer values only. 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* Specify an output file. By default, `iw2` writes to the standard output, so this option will redirect the output to *file*.

`-p` *pitch* Specify 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` Print at 9 cpi, for 72 characters per line.
`10` Print at 10 cpi, for 80 characters per line.
`12` Print at 12 cpi, for 96 characters per line.
`13` Print at 13.4 cpi, for 107 characters per line. `13.4` may also be specified.
`15` Print at 15 cpi, for 120 characters per line.
`17` Print at at 17 cpi, for 136 characters per line.
`pica`
Print pica proportional font. Averages 10 cpi.
`elite`
Print elite proportional font. Averages 12 cpi.

`-q` *quality* Specify 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` Print in draft quality mode.
`better` Print in better, or correspondence quality mode.
`nlq` Print in best, near letter quality. You may also specify `best` for this mode.

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

`2` Set line spacing to 2 lines per inch.
`3` Set line spacing to 3 lines per inch.
`4` Set line spacing to 4 lines per inch.
`6` Set line spacing to 6 lines per inch.
`8` Set line spacing to 8 lines per inch.
`9` Set line spacing to 9 lines per inch.

Or the value can have a slash (/) affixed to it. This value, then, 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* Specify tab settings. Default tabs are set every 8 columns (9, 17, 25, ...). This flag 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` flag option for the margin range table.

`-u` This flag option 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* Set 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 flag 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 nine allow set dot spacing from 1 to 10.

`-x` This flag option 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 flag option, when encountered, resets the buffer holding the initialization sequences that were built by processing the default and environmental variable.

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

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 only, so 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.

NOTES

Using the -x flag 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 want 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
```

Or 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 both 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 flag 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.

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

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

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

FILES

/usr/bin/iw2

SEE ALSO

daiw(1), iwprep(1), lp(1), iwmap(4).

NAME

iwprep — prepare troff description files

SYNOPSIS

iwprep [*file*]

DESCRIPTION

iwprep processes a *file* describing a troff output device and creates the *device* and *font* files required by troff(1) as described in troff(5).

FILES

/usr/bin/iwprep	
/usr/lib/font/deviw/*.def	description files for the ImageWriter II.

SEE ALSO

troff(1), iwdesc(4), troff(5).

NAME

join — relational database operator

SYNOPSIS

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

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 below). The default output field separator is a blank.

Some of the below flag options use the argument *n*. This argument should be a 1 or a 2 referring to either *file1* or *file2*, respectively. The following flag options are recognized:

–a*n* In addition to the normal output, produce a line for each unpairable line in file *n*, where *n* is 1 or 2.

–e *s* Replace empty output fields by string *s*.

–j*n m* join 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.

–o *list* Each output line comprises 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* Use 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.

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

```
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
```

```
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.

FILES

`/usr/bin/join`

SEE ALSO

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

BUGS

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 flag option is used right before listing filenames.

NAME

kermit — Kermit file transfer

SYNOPSIS

kermit [*option...*] [*file...*]

DESCRIPTION

kermit is a file transfer program that allows files to be moved between machines of 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 will enter command mode. Otherwise, kermit will read the arguments off the command line and interpret them.

The following notation is used in command descriptions:

fn	An A/UX file specification, possibly containing either of the metacharacters "*", which matches all character strings, or "?", which matches any single character.
fn1	An A/UX file specification which may not contain * or ?.
rfn	A remote file specification in the remote system's own syntax, which may denote a single file or a group of files.
rfn1	A remote file specification which should denote only a single file.
n	A decimal number between 0 and 94.
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.
[ ]	Any field in brackets is optional.
{*x*,*y* ,*z*}	Alternatives are listed in braces.

kermit command line options may specify either actions or settings. If kermit is invoked with a command line that specifies no actions, then it will issue a prompt and begin interactive dialog. Action options specify either protocol transactions or terminal connection.

FLAG OPTIONS

-s *fn* Send the specified file or files. If *fn* contains metacharacters, the A/UX shell expands it into a list. If *fn* is -, then `kermit` sends from standard input, which must come from a file:

```
kermit -s - < foo.bar
```

or a parallel process:

```
ls -l | kermit -s -
```

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

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

-r Receive a file or files. Wait passively for files to arrive.

-k Receive (passively) a file or files, sending them to standard output. This option can be used in several ways:

```
kermit -k
```

displays the incoming files on your screen; to be used only in "local mode" (see below).

```
kermit -k > fnl
```

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

```
kermit -k | command
```

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

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

-a *fnl* If you have specified a file transfer option, you may specify an alternate name for a single file with the -a 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 the -a option:

```
kermit -ra baz
```

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

`-x` Begin server operation. May be used in either local or remote mode.

Before proceeding, a few words about remote and local operation are necessary. `kermit` is "local" if it is running on a personal computer or workstation that you are using directly, or if it is running on a multiuser system and transferring files over an external communication line, not your job's controlling terminal or console. `kermit` is remote if it is running on a multiuser system and transferring files over its own controlling terminal's communication 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 multiuser (timesharing) system, it is in remote mode unless you explicitly point it at an external line for file transfer or terminal connection. The following command sets `kermit`'s "mode":

`-l` *dev* Line; specify a terminal line to use for file transfer 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:

`-b` *n* Baud; specify the baud rate for the line given in the `-l` option, as in

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

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

`-p` *x* Parity; `e`, `o`, `m`, `s`, `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` agrees. The default parity is none.

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

The following commands may be used only with a `kermit` which is local, either by default or else because the `-l` option has been specified.

`-g` *rfn* Actively request 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\*.\?
```

`-f` Send a "finish" command to a remote server.

`-c` Establish 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`.

`-n` Like `-c`, but after a protocol transaction takes place; `-c` and `-n` may both be used in the same command. The use of `-n` and `-c` is illustrated below.

On a timesharing system, the `-l` and `-b` options will also have to be included with the `-r`, `-k`, or `-s` options if the other `kermit` 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 tranfser. 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 `%` for each retransmission. In addition, you may type (to standard input) certain "interrupt" commands during file transfer:

CONTROL-F	Interrupt the current file, and go on to the next (if any).
CONTROL-B	Interrupt the entire batch of files, terminate the transaction.
CONTROL-R	Resend the current packet
CONTROL-A	Display 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 the escape character (e.g. CONTROL-\).

Several other command-line options are provided:

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

`-w` Write-Protect; avoid filename collisions for incoming files.

`-q` Quiet; suppress screen update during file transfer, for instance to allow a file transfer to proceed in the background.

`-d` Debug; record 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.

`-h` Help; display a brief synopsis of the command line options.

The command line may contain no more than one protocol action option.

INTERACTIVE OPERATION

`kermit`'s interactive command prompt is

```
C-Kermit>
```

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

Commands begin with a keyword, normally an English verb, such as "`send`". You may 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`", "`connect`") have special nonunique abbreviations ("`s`"`for` send, ("`r`"`for` receive, ("`c`"`for` connect).

Certain characters have special functions in interactive commands:

`?` Question mark, typed at any point in a command, will produce 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.

`ESC` The ESCAPE or ALTMODE key; request completion of the current keyword or filename, or insertion of a default value. The result will be a beep if the requested operation fails.

`DEL` The DELETE or RUBOUT key; delete the previous character from the command. You may also use BS (, CONTROL-H) for this function.

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

`^U` CONTROL-U; erase the entire command.

`^R` CONTROL-R; redisplay the current command.

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

`CR` Carriage return; enters the command for execution. `LF` linefeed or `FF` (formfeed) may also be used for this purpose.

`\` Backslash; enter any of the above characters 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.

You may type the editing characters (`DEL`, `^W`, etc.) repeatedly, to delete all the way back to the prompt. No action will be performed until the 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`.

Interactive 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 it looks in the home directory, then 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 may be nested to any reasonable depth.

Here is a brief list of kermit interactive commands:

!	Execute an A/UX shell command.
bye	Terminate and logout a remote kermit server.
close	Close a log file.
connect	Establish a terminal connection to a remote system.
cwd	Change working directory.
dial	Dial a telephone number.
directory	Display a directory listing.
echo	Display arguments literally.
exit	Exit from the program, closing any open logs.
finish	Instruct a remote kermit server to exit, but not log out.
get	Get files from a remote kermit server.
help	Display a help message for a given command.
log	Open a log file — debugging, packet, session, transaction.
quit	Same as "exit".
receive	Passively wait for files to arrive.
remote	Issue file management commands to a remote kermit server.
script	Execute a login script with a remote system.

`send`	Send files.
`server`	Begin server operation.
`set`	Set various parameters.
`show`	Display values of "`set`" parameters.
`space`	Display current disk space usage.
`statistics`	Display statistics about most recent transaction.
`take`	Execute commands from a file.

The "`set`" parameters are:

`block-check`	Level of packet error detection.
`delay`	How long to wait before sending first packet.
`duplex`	Specify which side echoes during "connect".
`escape-character`	Character to prefix "escape commands" during `connect`.
`file`	Set various file parameters.
`flow-control`	Communication line full-duplex flow control.
`handshake`	Communication line half-duplex turnaround character.
`line`	Communication line device name.
`modem-dialer`	Type of modem-dialer on communication line.
`parity`	Communication line character parity.
`prompt`	Change the `kermit` program's prompt.
`receive`	Set various parameters for inbound packets.
`send`	Set various parameters for outbound packets.
`speed`	Communication line speed.

The "remote" commands are:

cwd	Change remote working directory.
delete	Delete remote files.
directory	Display a listing of remote file names.
help	Request help from a remote server.
host	Issue a command to the remote host in its own command language.
space	Display current disk space usage on remote system.
type	Display a remote file on your screen.
who	Display who's logged in, or get information about a user.

FILES

/usr/bin/kermit	
$HOME/.kermrc	kermit initialization commands
./.kermrc	more kermit initialization commands

SEE ALSO

cu(1C), uucp(1C).
Kermit User's Guide, Frank da Cruz and Bill Catchings, Columbia University, 6th Edition.

DIAGNOSTICS

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

BUGS

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

NAME

kill — terminate a process

SYNOPSIS

kill [*–sig*] *pid...*

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(1).

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.

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

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

EXAMPLES

```
kill 24068
```

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

FILES

/bin/kill

SEE ALSO

ps(1), sh(1), csh(1), ksh(1), kill(2), signal(3).

NAME

ksh — run the Korn shell, a command interpreter compatible with Bourne shell

SYNOPSIS

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

DESCRIPTION

ksh is a command programming language that executes commands that are read from a terminal or a file. See "Invocation" later in this section for the meaning of arguments to the shell.

Definitions

A *metacharacter* is one of the following characters:

; & ( ) | < > *newline space tab*

A *blank* is a tab or a space. An *identifier* is a sequence of letters, digits, or underscores starting with a letter or underscore. Identifiers are used as names for *aliases*, *functions*, and named *parameters*. A *word* is a sequence of characters separated by one or more nonquoted *metacharacters*.

Commands

A *simple-command* is a sequence of blank-separated words which may be preceded by a parameter assignment list (see "Environment" later in this section). The first word specifies the command name to be executed. With exceptions described later, the remaining words are passed as arguments to the invoked command. The command name is passed as argument 0 (see exec(2)). The *value* of a simple-command is its exit status if it terminates normally, or (octal) 200+*status* if it terminates abnormally (see signal(3) for a list of status values).

A *pipeline* is a sequence of one or more *commands* separated by a vertical bar (|). The standard output of all but the last command is connected by a pipe(2) to the standard input of the next command. Each command is run as a separate process; the shell waits for the final command to terminate. The exit status of a pipeline is the exit status of the last command.

A *list* is a sequence of one or more pipelines separated by ;, &, &&, or ||, and optionally terminated by ;, &, or |&. Of these five symbols, ;, &, and |& have equal precedence, which is lower than that of && and ||. The symbols && and || also have equal precedence. A semicolon (;) causes sequential execution of the

preceding pipeline; an ampersand (&) causes asynchronous execution of the preceding pipeline (that is, the shell does *not* wait for that pipeline to finish). The symbol |& causes asynchronous execution of the preceding command or pipeline with a two-way pipe established to the parent shell. The standard input and output of the spawned command can be written to and read from by the parent shell using the -p flag option of the special commands, `read` and `print` described later. Only one such command can be active at any given time. The symbol && (||) causes the *list* following it to be executed only if the preceding pipeline returns a zero (nonzero) value. An arbitrary number of newlines may appear in a *list*, instead of semicolons, to delimit commands.

Unless stated otherwise, the value returned is that of the last simple-command executed in the command. A *command* is either a simple-command or one of the following:

`for` *identifier* [`in` *word*...] `do` *list* `done`
: Each time a `for` command is executed, *identifier* is set to the next *word* taken from the `in` *word* list. If `in` *word* ... is omitted, then the `for` command executes the `do` *list* once for each positional parameter that is set (see "Parameter Substitution" later). Execution ends when there are no more words in the list.

`select` *identifier* [`in` *word*...] `do` *list* `done`
: A `select` command prints on standard error (file descriptor 2), the set of *words*, each preceded by a number. If `in` *word* ... is omitted, then the positional parameters are used instead (see "Parameter Substitution" later). The `PS3` prompt is printed and a line is read from the standard input. If this line consists of the number of one of the listed `words`, then the value of the parameter *identifier* is set to the *word* corresponding to this number. If this line is empty, the selection list is printed again. Otherwise the value of the parameter *identifier* is set to `null`. The contents of the line read from standard input is saved in the parameter `REPLY`. The *list* is executed for each selection until a break or end-of-file is encountered.

`case` *word* `in` [*pattern* [| *pattern*] ...) *list* `;;`]... `esac`
: A `case` command executes the *list* associated with the first *pattern* that matches *word*. The form of the patterns is the same as that used for filename generation (see "Filename Generation" later).

`if` *list* `then` *list* [`elif` *list* `then` *list*] ... [`else` *list*] `fi`
The *list* following `if` is executed and, if a zero exit status is returned, the *list* following the first `then` is executed. Otherwise, the *list* following `elif` is executed and, if its value is zero, the *list* following the next `then` is executed. Failing that, the `else` *list* is executed. If no `else` *list* or `then` *list* is executed, then the `if` command returns a zero exit status.

`while` *list* `do` *list* `done`
`until` *list* `do` *list* `done`
A `while` command repeatedly executes the `while` *list* and, if the exit status of the last command in the list is zero, executes the `do` *list*; otherwise the loop terminates. If no commands in the `do` *list* are executed, then the `while` command returns a zero exit status; `until` may be used in place of `while` to negate the loop termination test.

(*list*)
Executes *list* in a separate environment. Note that if two adjacent open parentheses are needed for nesting, a space must be inserted to avoid arithmetic evaluation as described later. A parenthesized list used as a command argument, denoting "process substitution," is also described.

{ *list;* }
list is simply executed. Note that { is a *keyword* and requires a blank in order to be recognized.

`function` *identifier* { *list;* }
identifier () { *list;* }
Defines a function which is referenced by *identifier*. The body of the function is the *list* of commands between { and }. (See "Functions" later in this section).

`time` *pipeline*
The *pipeline* is 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 occurring as the first word of a command and when not quoted.

```
if then else elif fi case esac for do
while until done { } function select time
```

Comments

A word beginning with a # causes that word and all the following characters prior to a newline to be ignored.

Aliasing

The first word of each command is replaced by the text of an `alias` if an `alias` for this word has been defined. The first character of an `alias` name can be any nonspecial printable character, but the rest of the characters must be the same as for a valid *identifier*. The replacement string can contain any valid shell script, including the metacharacters listed earlier. The first word of each command of the replaced text will not be tested for additional aliases. If the last character of the alias value is a blank, then the word following the alias will also be checked for alias substitution. Aliases can be used to redefine special built-in commands but cannot be used to redefine the keywords listed earlier. Aliases can be created, listed, and exported with the `alias` command and can be removed with the `unalias` command. Exported aliases remain in effect for subshells but must be reinitialized for separate invocations of the shell (see "Invocation" later in this section).

Aliasing is performed when scripts are read, not while they are executed. Therefore, for an alias to take effect, the `alias` command has to be executed before the command which references the alias is read.

Aliases are frequently used as a type of shorthand for full pathnames. A flag option to the aliasing facility allows the value of the alias to be automatically set to the full pathname of the corresponding command. These aliases are called "tracked" aliases. The value of a tracked alias is defined the first time the corresponding command is looked up and becomes undefined each time the `PATH` variable is reset. These aliases remain tracked so that the next subsequent reference will redefine the value. Several tracked aliases are compiled into the shell. The `-h` flag option of the `set` command makes each command name that is a valid alias name into a tracked alias.

The following "exported aliases" are compiled into the shell but can be unset or redefined:

```
false='let 0'
functions='typeset -f'
history='fc -l'
integer='typeset -i'
```

```
nohup='nohup '
r='fc -e -'
true=':'
type='whence -v'
hash='alias -t'
```

Tilde Substitution

After alias substitution is performed, each word is checked to see if it begins with an unquoted tilde (~). If it does, then the word prior to a / is checked to see if it matches a user name in the /etc/passwd file. If a match is found, the ~ and the matched login name is replaced by the login directory of the matched user. This is called a tilde substitution. If no match is found, the original text is left unchanged. A ~ by itself, or in front of a /, is replaced by the value of the HOME parameter. A ~ followed by a + or - is replaced by the value of the parameter PWD and OLDPWD respectively.

In addition, the value of each *keyword parameter* is checked to see if it begins with a ~ or if a ~ appears after a :. In either of these cases, a tilde substitution is attempted.

Command Substitution

The standard output from a command enclosed in parentheses preceded by a dollar sign $() or a pair of grave accents `` may be used as part or all of a word; trailing newlines are removed. In the second (archaic) form, the string between the quotes is processed for special quoting characters before the command is executed. (See "Quoting" later.) The command substitution $(cat *file*) can be replaced by the equivalent but faster $(<*file*). Command substitution of most special commands that do not perform input/output redirection are carried out without creating a separate process.

Process Substitution

This feature is only available on versions of the A/UX operating system that support the /dev/fd directory for naming open files. Each command argument of the form (*list*), <(*list*), or >(*list*) will run the process *list* asynchronously connected to some file in /dev/fd. The name of this file will become the argument to the command. If the form with > is selected then writing on this file will provide input for *list*. If < is used or omitted, then the file passed as an argument will contain the output of the *list* process. For example,

`paste (cut -f1 file1) (cut -fC file2) | tee >(proc1)>(proc2)`

`cut`s fields 1 and 3 from the files *file1* and *file2* respectively, `paste`s the results together, and sends it to *process1* and *process2*, as well as putting it onto the standard output. Note that the file, which is passed as an argument to the command, is an A/UX `pipe`(2) so programs that expect to `lseek`(2) on the file will not work.

Parameter Substitution

A *parameter* is an *identifier*, one or more digits, or any of the characters `*`, `@`, `#`, `?`, `-`, `$`, and `!`. A *named parameter* (a parameter denoted by an identifier) has a *value* and has zero or more *attributes*. By using the `typeset` special command, *named parameters* can be assigned *values* and *attributes*. The attributes supported by the shell are described later with the `typeset` special command. Exported parameters pass values and attributes to subshells but values only to the environment.

The shell supports a limited one-dimensional array facility. An element of an array parameter is referenced by a *subscript*. A *subscript* is denoted by a `[`, followed by an *arithmetic expression* (see "Arithmetic Evaluation" later in this section), followed by a `]`. The value of all subscripts must be in the range of 0 through 511. Arrays need not be declared. Any reference to a named parameter with a valid subscript is legal and an array will be created if necessary. Referencing an array without a subscript is equivalent to referencing the first element.

The *value* of a *named parameter* may also be assigned by writing:

name=*value* [*name*=*value*] ...

If the integer attribute, `-i`, is set for *name* the *value* is subject to arithmetic evaluation as described later.

Positional parameters denoted by a number are assigned values with the `set` special command. Parameter `$0` is set from argument zero when the shell is invoked.

The character `$` is used to introduce substitutable *parameters*.

`${`*parameter*`}`

The value, if any, of the parameter is substituted. The braces are required when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name or when a named parameter is subscripted. If *parameter* is one or more digits, then it is a positional parameter. A positional

parameter of more than one digit must be enclosed in braces. If *parameter* is * or @, then all the positional parameters, starting with $1, are substituted (separated by a field separator character). If an array *identifier* with subscript * or @ is used, then the value for each of the elements is substituted (separated by a field separator character).

${#*parameter*}
If *parameter* is * or @, the number of positional parameters is substituted. Otherwise, the length of the value of the *parameter* is substituted.

${#*identifier*[*]}
The number of elements in the array *identifier* is substituted.

${*parameter*:-*word*}
If *parameter* is set and is not null, then substitute its value; otherwise substitute *word*.

${*parameter*:=*word*}
If *parameter* is not set or is null, then set it to *word*; the value of the parameter is then substituted. Positional parameters may not be assigned in this way.

${*parameter*:?*word*}
If *parameter* is set and is not null, then substitute its value; otherwise print *word* and exit from the shell. If *word* is omitted then a standard message is printed.

${*parameter*:+*word*}
If *parameter* is set and is not null, then substitute *word*; otherwise substitute nothing.

${*parameter*#*pattern*}
${*parameter*##*pattern*}
If the shell *pattern* matches the beginning of the value of *parameter*, then the value of this substitution is the value of the *parameter* with the matched portion deleted; otherwise the value of this *parameter* is substituted. In the first form, the smallest matching pattern is deleted and in the latter form, the largest matching pattern is deleted.

${*parameter*%*pattern*}
${*parameter*%%*pattern*}
If the shell *pattern* matches the end of the value of *parameter*, then the value of this substitution is the value of *parameter* with the matched part deleted; otherwise substitute the

value of *parameter*. In the first form, the smallest matching pattern is deleted and in the latter form, the largest matching pattern is deleted.

In the preceding, *word* is not evaluated unless it is to be used as the substituted string, so that, in the following example, `pwd` is executed only if `d` is not set or is null.

```
echo ${d:- $(pwd)}
```

If the colon (`:`) is omitted from the above expression, then the shell only checks as to whether *parameter* is set or not.

The following parameters are automatically set by the shell.

#	The number of positional parameters in decimal.
-	Flags supplied to the shell on invocation or by the `set` command.
?	The decimal value returned by the last executed command.
$	The process number of this shell.
_	The last argument of the previous command. This parameter is not set for commands which are asynchronous. This parameter is also used to hold the name of the matching `MAIL` file when 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.
!	The process number of the last background command invoked.
`PPID`	The process number of the parent of the shell.
`PWD`	The present working directory set by the `cd` command.
`OLDPWD`	The previous working directory set by the `cd` command.
`RANDOM`	Each time this parameter is referenced, a random integer is generated. The sequence of random numbers can be initialized by assigning a numeric value to `RANDOM`.

REPLY — This parameter is set by the `select` statement and by the `read` special command when no arguments are supplied.

SECONDS — Each time this parameter is referenced, the number of seconds since shell invocation is returned. If this parameter is assigned a value, then the value returned upon reference will be the value that was assigned plus the number of seconds since the assignment.

The following parameters are used by the shell.

CDPATH — The search path for the `cd` command.

COLUMNS — If this variable is set, the value is used to define the width of the edit window for the shell edit modes and for printing `select` lists.

EDITOR — If the value of this variable ends in `vi` and the `VISUAL` variable is not set, then the corresponding flag option (see "Special Commands") will be turned on.

ENV — If this parameter is set, then parameter substitution is performed on the value to generate the pathname of the script that will be executed when the shell is invoked (see "Invocation"). This file is typically used for `alias` and `function` definitions.

FCEDIT — The default editor name for the `fc` command.

IFS — Internal field separators, normally space, tab, and newline that are used to separate command words which result from command or parameter substitution and for separating words with the special command `read`. The first character of the `IFS` parameter is used to separate arguments for the `$*` substitution (see "Quoting").

HISTFILE — If this parameter is set when the shell is invoked, then the value is the pathname of the file that will be used to store the command history (see "Command Re-entry").

HISTSIZE — If this parameter is set when the shell is invoked, then the number of previously entered com-

mands that are accessible by this shell will be greater than or equal to this number. The default is 128.

HOME — The default argument (home directory) for the `cd` command.

LINES — If this variable is set, the value is used to determine the column length for printing `select` lists. Select lists will print vertically until about two-thirds of `LINES` lines are filled.

MAIL — If this parameter is set to the name of a mail file *and* the `MAILPATH` parameter is not set, then the shell informs the user of arrival of mail in the specified file.

MAILCHECK — This variable specifies how often (in seconds) the shell will check for changes in the modification time of any of the files specified by the `MAILPATH` or `MAIL` parameters. The default value is 600 seconds. When the time has elapsed, the shell will check before issuing the next prompt.

MAILPATH — A colon (`:`) separated list of filenames. If this parameter is set, then 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 will undergo parameter and command substitution with the parameter, `$_`, defined as the name of the file that has changed. The default message is `You have mail in $_`.

PATH — The search path for commands (see "Execution").

PS1 — The value of this parameter is expanded for parameter substitution to define the primary prompt string which by default is `$`. The character `!` in the primary prompt string is replaced by the *command* number (see "Command Reentry").

PS2	Secondary prompt string, by default >.
PS3	Selection prompt string used within a `select` loop, by default `#?`.
SHELL	The pathname of the shell is kept in the environment. At invocation, if the value of this variable contains an `r` in the basename, then the shell becomes restricted.
TMOUT	If set to a value greater than zero, the shell will terminate if a command is not entered within the prescribed number of seconds after issuing the `PS1` prompt. (Note that the shell can be compiled with a maximum bound for this value which cannot be exceeded.)
VISUAL	If the value of this variable ends in `vi` then the corresponding option (see "Special Commands") will be turned on.

The shell gives default values to `PATH`, `PS1`, `PS2`, `MAILCHECK`, `TMOUT`, and `IFS`, while `HOME`, `SHELL`, `ENV`, and `MAIL` are not set by the shell (although `HOME` is set by `login`(1)). On some systems `MAIL` and `SHELL` are also set by `login`(1)).

Blank Interpretation

After parameter and command substitution, the results of substitutions are scanned for the field separator characters (those found in `IFS`) and split into distinct arguments where such characters are found. Explicit null arguments `""` or `''` are retained. Implicit null arguments (those resulting from *parameters* that have no values) are removed.

Filename Generation

Following substitution, each command *word* is scanned for the characters `*`, `?`, and `[` unless the `-f` option has been set. If one of these characters appears, then the word is regarded as a *pattern*. The word is replaced with alphabetically sorted filenames that match the pattern. If no filename is found that matches the pattern, then the word is left unchanged. When a *pattern* is used for filename generation, the character `.` at the start of a filename or immediately following a `/`, as well as the `/` itself, must be matched explicitly. In other instances of pattern matching the `/` and `.` are not treated specially.

*	Matches any string, including the null string.
?	Matches any single character.
[...]	Matches any one of the enclosed characters. A pair of characters separated by - matches any character lexically between the pair, inclusive. If the first character following the opening [is a ! then any character not enclosed is matched. A - can be included in the character set by putting it as the first or last character.

Quoting

Each of the *metacharacters* listed above (see "Definitions" earlier) has a special meaning to the shell and causes termination of a word unless quoted. A character may be "quoted" (that is, made to stand for itself) by preceding it with a backslash (\). The pair \newline is ignored. All characters enclosed between a pair of single quote marks '' are quoted. A single quote cannot appear within single quotes. Inside double quote marks "" parameter and command substitution occurs and \ quotes the characters \, `, ", and $. The meaning of $* and $@ is identical when not quoted or when used as a parameter assignment value or as a filename. However, when used as a command argument, $* is equivalent to $1*d* $2*d*..., where *d* is the first character of the IFS parameter, whereas $@ is equivalent to $1 $2.... Inside grave accent marks (``), \ quotes the characters \, `, and $. If the grave accents occur within double quotes, then \ also quotes the character ".

The special meaning of keywords or aliases can be removed by quoting any character of the keyword. The recognition of function names or special command names listed later in this section cannot be altered by quoting them.

Arithmetic Evaluation

An ability to perform integer arithmetic is provided with the special command let. Evaluations are performed using *long* arithmetic. Constants are of the form [*base*#]*n* where *base* is a decimal number between 2 and 36 representing the arithmetic base and *n* is a number in that base. If *base* is omitted, then base 10 is used.

An internal integer representation of a *named parameter* can be specified with the -i option of the typeset special command. When this attribute is selected, the first assignment to the parame-

ter determines the arithmetic base to be used when parameter substitution occurs.

Since many of the arithmetic operators require quoting, an alternative form of the `let` command is provided. For any command which begins with a `((`, all the characters until a matching `))` are treated as a quoted expression. More precisely, `((...))` is equivalent to `let "..."`.

Prompting

When used interactively, the shell prompts with the value of `PS1` before reading a command. If at any time a newline is typed and further input is needed to complete a command, then the secondary prompt (that is, the value of `PS2`) is issued.

Input/Output

Before a command is executed, its input and output may be redirected using a special notation interpreted by the shell. The following may appear anywhere in a simple-command or may precede or follow a *command* and are *not* passed on to the invoked command. Command and parameter substitution occurs before *word* or *digit* is used except as noted. Filename generation occurs only if the pattern matches a single file and blank interpretation is not performed.

<*word*	Use file *word* as standard input (file descriptor 0).
>*word*	Use file *word* as standard output (file descriptor 1). If the file does not exist then it is created; otherwise, it is truncated to zero length.
>>*word*	Use file *word* as standard output. If the file exists then output is appended to it (by first seeking to the end-of-file); otherwise, the file is created.
<<[-]*word*	The shell input is read up to a line that is the same as *word*, or to an end-of-file. No parameter substitution, command substitution, or filename generation is performed on *word*. The resulting document, called a "here-document," becomes the standard input. If any character of *word* is quoted, then no interpretation is placed upon the characters of the document; otherwise, parameter and command substitution occurs, the \newline command is ignored, and \ must be used to

quote the characters \, $, ', and the first character of *word*. If - is appended to <<, then all leading tabs are stripped from *word* and from the document.

<&*digit* The standard input is duplicated from the file descriptor *digit* (see dup(2)). Similarly for the standard output using >& *digit*.

<&- The standard input is closed. Similarly for the standard output using >&-.

If one of the above is preceded by a digit, then the file descriptor number referred to is that specified by the digit (instead of the default 0 or 1). For example,

```
...2>&1
```

means file descriptor 2 is to be opened for writing as a duplicate of file descriptor 1.

The order in which redirections are specified is significant. The shell evaluates each redirection in terms of the (*file descriptor*, *file*) association at the time of evaluation. For example,

```
...1>fname2>&1
```

first associates file descriptor 1 with file *fname*. It then associates file descriptor 2 with the file associated with file descriptor 1 (that is, *fname*). If the order of redirections were reversed, file descriptor 2 would be associated with the terminal (assuming previous association by file descriptor 1) and then file descriptor 1 would be associated with file *fname*.

If a command is followed by & and job control is not active, then the default standard input for the command is the empty file /dev/null. Otherwise, the environment for the execution of a command contains the file descriptors of the invoking shell as modified by input/output specifications.

Environment

The environment (see environ(7)) is a list of name-value pairs that is passed to an executed program in the same way as a normal argument list. The names must be *identifiers* and the values are character strings. The shell interacts with the environment in several ways. On invocation, the shell scans the environment and creates a parameter for each name found, giving it the corresponding value and marking it export. Executed commands inherit

the environment. If the user modifies the values of these parameters or creates new ones, using the `export` or `typeset -x` commands, they become part of the environment. The environment seen by any executed command is thus composed of any name-value pairs originally inherited by the shell, whose values may be modified by the current shell, plus any additions which must be noted in `export` or `typeset -x` commands.

The environment for any *simple-command* or function may be augmented by prefixing it with one or more parameter assignments. A parameter assignment argument is a word of the form *identifier=value*. Thus, the following two commands are equivalent (as far as the above execution of *cmd* is concerned).

```
TERM=450 cmd args
(export TERM; TERM=450; cmd args)
```

If the `-k` flag is set, *all* parameter assignment arguments are placed in the environment, even if they occur after the command name. The command that follows first prints `a=b c` and then `c`.

```
echo a=b c
set -k
echo a=b c
```

Functions

The `function` keyword, described in the "Commands" section earlier, is used to define shell functions. Shell functions are read in and stored internally. Alias names are resolved when the function is read. Functions are executed like commands with the arguments passed as positional parameters. (See "Execution" later in this section.)

Functions execute in the same process as the caller and share all files, traps (other than `EXIT` and `ERR`), and present working directories with the caller. A trap set on `EXIT` inside a function is executed after the function completes. Ordinarily, variables are shared between the calling program and the function. However, the `typeset` special command used within a function defines local variables whose scope includes the current function and all functions it calls.

The special command `return` is used to return from function calls. Errors within functions return control to the caller.

Function identifiers can be listed with the `-f` option of the `typeset` special command. The text of functions will also be listed. Function can be undefined with the `-f` option of the `unset` special command.

Ordinarily, functions are unset when the shell executes a shell script. The `-xf` option of the `typeset` command allows a function to be exported to scripts that are executed without a separate invocation of the shell. Functions that need to be defined across separate invocations of the shell should be placed in the ENV file.

Jobs

If the `monitor` option of the `set` command is turned on, an interactive shell associates a job with each pipeline. It keeps a table of current jobs, printed by the `jobs` command, and assigns them small integer numbers. When a job is started asynchronously with `&`, the shell prints a line which looks like

```
[1] 1234
```

indicating that the job which was started asynchronously was job number 1 and had one (top-level) process, whose process ID was 1234.

If you are running a job and wish to do something else you may hit CONTROL-Z which sends a `STOP` signal to the current job. The shell will then normally indicate that the job has been "Stopped," and print another prompt. You can then manipulate the state of this job, putting it in the background with the `bg` command, or run some other commands and then eventually bring the job back into the foreground with the foreground command `fg`. A CONTROL-Z takes effect immediately and is similar to an interrupt in that pending output and unread input are discarded when it is typed.

A job being run in the background will stop if it tries to read from the terminal. Background jobs are normally allowed to produce output, but this can be disabled by giving the command `stty tostop`. If you set this tty option, then background jobs will stop when they try to produce output as they do when they try to read input.

There are several ways to refer to jobs in the shell. The character `%` introduces a job name. If you wish to refer to job number 1, you can give it the name `%1`. Jobs can also be named with prefixes of the string typed in to start them. Thus, on systems that support job control, `fg %ed` would normally restart a suspended `ed`(1) job,

provided a suspended job whose name began with the string `ed` was present.

The shell maintains a notion of current and previous jobs. In output pertaining to jobs, the current job is marked with a + and the previous job with a -. The abbreviation `%+` refers to the current job and `%-` refers to the previous job. `%%` is also a synonym for the current job.

This shell knows immediately when the state of a process changes and will generally inform you when a job becomes blocked and no further progress is possible, This information is offered just before the shell prints a prompt, so as to not be interrupted.

When you try to leave the shell while jobs are running or stopped, you will be warned that "`You have stopped (running) jobs`". You may use the `jobs` command to see what they are. If you use this command or immediately try to exit again, the shell will not warn you a second time, and the stopped jobs will be terminated.

Signals

The `INT` and `QUIT` signals for an invoked command are ignored if the command is followed by an `&` and the job `monitor` option is not active. Otherwise, signals have the values inherited by the shell from its parent (see also the `trap` command later in this section).

Execution

Each time a command is executed, the substitutions described are carried out. If the command name matches one of the "Special Commands" listed later, it is executed within the current shell process. Next, the command name is checked to see if it matches one of the user defined functions. If it does, the positional parameters are saved and then reset to the arguments of the *function* call. When the *function* completes or issues a RETURN, the positional parameter list is restored and any trap set on `EXIT` within the function is executed. The value of a *function* is the value of the last command executed. A function is also executed in the current shell process. If a command name is not a special command or a user defined *function*, a process is created and an attempt is made to execute the command via `exec`(2).

The shell parameter `PATH` defines the search path for the directory containing the command. Alternative directory names are separated by a colon (`:`). The default path is `/bin:/usr/bin:` (speci-

fying /bin, /usr/bin, and the current directory in that order). The current directory can be specified by two or more adjacent colons, or by a colon at the beginning or end of the path list. If the command name contains a / then the search path is not used. Otherwise, each directory in the path is searched for an executable file. If the file has execute permission but is not a directory or an a.out file, it is assumed to be a file containing shell commands and a subshell is spawned to read it. All nonexported aliases, functions, and named parameters are removed in this case. If the shell command file doesn't have read permission, or if the setuid and/or setgid bits are set on the file, then the shell executes an agent whose job it is to set up the permissions and execute the shell with the shell command file passed down as an open file. A parenthesized command is also executed in a subshell without removing nonexported quantities.

Command Re-entry

The text of the last HISTSIZE (default 128) commands entered from a terminal device is saved in a history file. The file $HOME/sh_history is used if the HISTFILE variable is not set or is not writable. A shell can access the commands of all interactive shells which use the same named HISTFILE. The special command fc is used to list or edit a portion of this file. The portion of the file to be edited or listed can be selected by number or by typing the first character or characters of the command. A single command or range of commands can be specified. If you do not specify an editor program as an argument to fc then the value of the parameter FCEDIT is used. If FCEDIT is not defined then /bin/ed is used. The edited command(s) is printed and reexecuted upon leaving the editor. The editor name - is used to skip the editing phase and to re-execute the command. In this case a substitution parameter of the form *old=new* can be used to modify the command before execution. For example, if r is aliased to 'fc -e -' then typing r bad=good c will reexecute the most recent command which starts with the letter c, replacing the first occurrence of the string bad with the string good.

Inline Editing Options

Normally, each command line entered from a terminal device is simply typed and followed by a newline (RETURN or LINEFEED). If the vi or emacs option is active, the user can edit the command line. To be in either of these edit modes, set the correspond-

ing option. An editing option is automatically selected each time the VISUAL or EDITOR variable is assigned a value ending in either of these option names.

The editing features require that the user's terminal accept RETURN as a carriage return without a linefeed and that a space ( ) must overwrite the current character on the screen.

> *Note:* ADM terminal users should set the "space-advance" switch to "space". Hewlett-Packard series 2621 terminal users should set the straps to "bcGHxZ etX".

The editing modes implement a concept where the user is looking through a window at the current line. The window width is the value of COLUMNS if it is defined, otherwise 80. If the line is longer than the window width minus two, a mark is displayed at the end of the window to notify the user. As the cursor moves and reaches the window boundaries the window will be centered around the cursor. The mark is a > (<, *) if the line extends on the right (left, both) side(s) of the window.

The emacs Editing Mode

This mode is entered by enabling either the emacs or gmacs option. The only difference between these two modes is the way they handle CONTROL-T. To edit, the user moves the cursor to the point needing correction and then inserts or deletes characters or words as needed. All the editing commands are control characters or escape sequences. The notation for control characters is caret (^) followed by the character. For example, ^F is the notation for CONTROL-F. The SHIFT key is *not* depressed. (The notation ^? indicates the DELETE.)

The notation for escape sequences is M- followed by a character. For example, M-f (pronounced Meta f) is entered by depressing ESCAPE (ASCII 033) followed by f. (M-F would be the notation for ESCAPE followed by SHIFT (uppercase) F.)

All edit commands operate from any place on the line (not just at the beginning). Neither RETURN nor LINEFEED is entered after edit commands except when noted.

^F	Move cursor forward (right) one character.
M-f	Move cursor forward one word. (The editor's idea of a word is a string of characters consisting of only letters, digits and underscores.)

^B	Move cursor backward (left) one character.
M-b	Move cursor backward one word.
^A	Move cursor to start of line.
^E	Move cursor to end of line.
^]*char*	Move cursor to character *char* on current line.
^X^X	Interchange the cursor and mark.
erase	Delete previous character. (User-defined erase character as defined by the stty command, usually CONTROL-H or #).
^D	Delete current character.
M-d	Delete current word.
M-^H	Delete previous word. (Meta-backspace)
M-h	Delete previous word.
M-^?	Delete previous word (Meta-delete) If your interrupt character is ^? (DELETE, the default), then this command will not work.
^T	Transpose current character with next character in emacs mode. Transpose two previous characters in gmacs mode.
^C	Capitalize current character.
M-c	Capitalize current word.
M-l	Change the current word to lowercase.
^K	Kill from the cursor to the end of the line. If given a parameter of zero then kill from the start of the line to the cursor.
^W	Kill from the cursor to the mark.
M-p	Push the region from the cursor to the mark on the stack.
kill	Kill the entire current line. (User-defined kill character as defined by the stty command, usually CONTROL-G or @.) If two *kill* characters are entered in succession, all kill characters following will cause a linefeed (useful when using paper terminals).
^Y	Restore last item removed from the line. (Yank item back to the line.)
^L	Line feed and print current line.
^@	Set mark. (Null character)
M-	Set mark. (Meta space)
^J	Execute the current line. (Newline)
^M	Execute the current line. (RETURN)

eof	End-of-file character, normally CONTROL-D, will terminate the shell if the current line is null.
^P	Fetch previous command. Each time CONTROL-P is entered, the previous command, backward in time, is accessed.
M-<	Fetch the least recent (oldest) history line.
M->	Fetch the most recent (youngest) history line.
^N	Fetch next command. Each time CONTROL-n is entered, the next command, forward in time, is accessed.
^R*string*	Reverse search history for a previous command line containing *string*. If a parameter of zero is given, the search is forward. The *string* is terminated by a RETURN or newline character. If *string* is omitted, then the next command line containing the most recent *string* is accessed. In this case, a parameter of zero reverses the direction of the search.
^O	Operate–execute the current line and fetch the next line relative to the current line from the history file.
M-*digits*	Define numeric parameter, the digits are taken as a parameter to the next command. (ESCAPE) The commands that accept a parameter are ., CONTROL-F, CONTROL-B, *erase*, CONTROL-D, CONTROL-K, CONTROL-R, CONTROL-P, CONTROL-n, M-., M-_, M-b, M-c, M-d, M-f, M-h, and M-^H.
M-*letter*	Soft-key searches your alias list for an alias by the name _*letter* and if an alias of this name is defined, its value will be inserted on the input queue. The *letter* must not be one of the above named meta-functions.
M-.	The last word of the previous command is inserted on the line. If preceded by a numeric parameter, the value of this parameter determines which word to insert other than the last word.
M-_	Same as M-..
M-*	Attempts filename generation on the current word. An asterisk is appended if the word doesn't contain any special pattern characters.
M-ESCAPE	Same as M-*.
M-=	Lists all files matching current word pattern if an asterisk is appended.

^U	Multiplies parameter of next command by 4.
\	Escape next character. Editing characters, the user's erase, kill, and interrupt (normally ^?) characters, may be entered in a command line or in a search string if preceded by a \. The \ removes the next character's editing features (if any).
^V	Display version of the shell.

The vi Editing Mode

There are two typing modes. Initially, when you enter a command you are in the *input* mode. To edit, the user enters *control* mode by typing ESCAPE (033), moves the cursor to the point needing correction, and then inserts or deletes characters or words as needed. Most control commands accept an optional repeat *count* prior to the command.

When in vi mode on most systems, canonical processing is initially enabled and the command will be echoed again if the speed is 1200 baud or greater, if it contains any control characters, or if less than one second has elapsed since the prompt was printed. The escape character terminates canonical processing for the remainder of the command and the user can then modify the command line. This scheme has the advantages of canonical processing with the type-ahead echoing of raw mode.

If the option viraw is also set, the terminal will always have canonical processing disabled. This mode is implicit for systems that do not support two alternate end-of-line delimiters, and may be helpful for certain terminals.

Input Edit Commands

By default the editor is in input mode.

Erase	Delete previous character. (User-defined erase character as defined by the stty command, usually CONTROL-H or #.)
^W	Delete the previous blank separated word.
^D	Terminate the shell.
^V	Escape next character. Editing characters, the user's erase or kill characters, may be entered in a command line or in a search string if preceded by a CONTROL-V. The CONTROL-V removes the next character's editing features (if any).
\	Escape the next erase or kill character.

Motion Edit Commands

These commands will move the cursor.

[*count*]`l`	Cursor forward (right) one character.
[*count*]`w`	Cursor forward one alpha-numeric word.
[*count*]`W`	Cursor to the beginning of the next word that follows a blank.
[*count*]`e`	Cursor to the end of the current word.
[*count*]`E`	Cursor to the end of the current blank delimited word.
[*count*]`h`	Cursor backward (left) one character.
[*count*]`b`	Cursor backward one word.
[*count*]`B`	Cursor to the preceding blank separated word.
[*count*]`f`*c*	Find the next character *c* in the current line.
[*count*]`F`*c*	Find the previous character *c* in the current line.
[*count*]`t`*c*	Equivalent to `f` followed by `h`.
[*count*]`T`*c*	Equivalent to `F` followed by `l`.
`;`	Repeats the last single character find command, `f`, `F`, `t`, or `T`.
`,`	Reverses the last single character find command.
`0`	Cursor to the start of the line.
`^`	Cursor to the first nonblank character in the line.
`$`	Cursor to the end of the line.

Search Edit Commands

These commands access your command history.

[*count*]`k`	Fetch previous command. Each time `k` is entered the previous command in time is accessed.
[*count*]`-`	Equivalent to `k`.
[*count*]`j`	Fetch next command. Each time `j` is entered the next command in time is accessed.
[*count*]`+`	Equivalent to `j`.
[*count*]`G`	The command number *count* is fetched. The default is the least recent history command.
`/`*string*	Search backward through history for a previous command containing *string*. The *string* is terminated by a RETURN or newline. If *string* is null the previous string will be used.
`?`*string*	Same as `/` except that the search will be in the forward direction.
`n`	Search for the next match of the last pattern to `/` or `?` commands.

`N` Search for the next match of the last pattern to / or ?, but in reverse direction. Search history for the *string* insert by the previous / command.

Text Modification Edit Commands

These commands will modify the line.

`a` Enter input mode and enter text after the current character.

`A` Append text to the end of the line. Equivalent to `$a`.

[*count*]`c`*motion*
`c`[*count*]*motion*
Delete from the current character through the character preceding the cursor which was moved by *motion*. If *motion* is `c`, the entire line will be deleted and the input mode entered.

`C` Delete from the current character through the end of the line and enter input mode. Equivalent to `c$`.

`S` Equivalent to `cc`.

`D` Delete from the current character through the end of the line. Equivalent to `d$`.

[*count*]`d`*motion*
`d`[*count*]*motion*
Delete from the current character through the character that *motion* would move to. If *motion* is `d`, the entire line will be deleted.

`i` Enter input mode and insert text before the current character.

`I` Insert text before the beginning of the line. Equivalent to the two character sequence `^i`.

[*count*]`P` Place the previous text modification before the cursor.

[*count*]`p` Place the previous text modification after the cursor.

`R` Enter input mode and replace the characters on the screen with characters you type in overlay fashion.

`r`*c* Replace the current character with *c*.

[*count*]x Delete current character.

[*count*]X Delete preceding character.

[*count*]. Repeat the previous text modification command.

~ Invert the case of the current character and advance the cursor.

[*count*]_ Causes the *count* word of the previous command to be appended and input mode entered. The last word is used if *count* is omitted.

* Causes an * to be appended to the current word and filename generation is attempted. If no match is found, it rings the bell. Otherwise, the word is replaced by the matching pattern and input mode is entered.

Other Edit Commands

[*count*]y*motion*
y[*count*]*motion*
Yank from the current character through the character that *motion* would move the cursor to and puts them into the delete buffer. The text and cursor are unchanged.

Y Yanks from the current position to the end of the line. Equivalent to y$.

u Undo the last text modifying command.

U Undo all the text modifying commands performed on the line.

[*count*]v Returns the command
`fc -e ${VISUAL:-${EDITOR:-vi}}` *count*
in the input buffer. If *count* is omitted, then the current line is used.

^L Line feed and print current line. Has effect only in the control mode.

^J Execute the current line, regardless of mode. (Newline)

^M Execute the current line, regardless of mode. (RETURN)

Inserts a # before the line and after each newline prior to sending it. Useful for causing the current line to be inserted in the history without being executed.

= Lists the filenames that match the current word as if an asterisk were appended to it.

@*letter* Search your `alias` list for an `alias` by the name _*letter* and if an alias of this name is defined, its value will be inserted on the input queue for processing.

Special Commands

The following simple commands are executed in the shell process. Input/output redirection is permitted. Unless otherwise indicated, the output is written on file descriptor 1. Commands that are preceded by one or two † are treated specially in the following ways:

- Parameter assignment lists preceding the command remain in effect when the command completes.
- Commands are executed in a separate process when used within command substitution.
- Errors in commands preceded by †† cause the script that contains them to abort.

† : [*arg* ...]
The command only expands parameters. A zero exit code is returned.

†† . *file* [*arg* ...]
Read and execute commands from *file* and return. 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 *arg* are given, they become the positional parameters. Otherwise the positional parameters are unchanged.

`alias` [`-tx`] [*name* [=*value*] ...]
`alias` with no arguments prints the list of aliases in the form *name*=*value* on standard output. An `alias` is defined for each name whose *value* is given. A trailing space in *value* causes the next word to be checked for alias substitution. The `-t` flag is used 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. Without the `-t` flag, assigned to each *name* in the argument list for which no *value* is given, the name and value of the alias is printed. The `-x` flag is used to set or print exported

aliases. An exported alias is defined across subshell environments. Alias returns true unless a *name* is given for which no alias has been defined.

`bg` [`%`*job*]
Puts the specified *job* into 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 *n* is specified, then break *n* levels.

`continue` [*n*]
Resumes the next iteration of the enclosing `for`, `while`, `until`, or `select` loop. If *n* is specified then resume at the *n*th enclosing loop.

† `cd` [*arg*]
† `cd` *old new*
This command can be in either of two forms. In the first form it changes the current directory to *arg*. If *arg* is - the directory is changed to the previous directory. The shell parameter `HOME` is the default *arg*. The parameter `PWD` is set to the current directory. The shell parameter `CDPATH` defines the search path for the directory containing *arg*. Alternative directory names are separated by a colon (`:`). The default path is `<null>` (specifying the current directory). Note that the current directory is specified by a null pathname, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If *arg* begins with a /, then the search path is not used. Otherwise, each directory in the path is searched for *arg*.

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.

`echo` [`-n`] [*arg*...]
The built-in `echo` command writes its arguments (separated by blanks and terminated by a RETURN) on the standard output. If the `-n` flag is used, no newline is added to the output. `echo` is useful for producing diagnostics in shell programs and for writing constant data on pipes. To send diagnostics to the standard error file, do

```
echo ... 1>&2
```

†† `eval` [*arg* ...]
The arguments are read as input to the shell and the resulting command(s) are executed.

†† `exec` [*arg* ...]
If *arg* is given, the command specified by the arguments is executed in place of this shell without creating a new process. Input/output arguments may appear and affect the current process. If no arguments are given, the effect of this command is to modify file descriptors as prescribed by the input/output redirection list. In this case, any file descriptor having numbers greater than 2 and are opened with this mechanism, are closed when invoking another program.

`exit` [*n*]
Causes the shell to exit with the exit status specified by *n*. If *n* is omitted then the exit status is that of the last command executed. An end-of-file will also cause the shell to exit unless it has the `ignoreeof` option (see `set` later in this section) turned on.

†† `export` [*name* ...]
The given *name*s are marked for automatic export to the *environment* of subsequently-executed commands.

†† `fc` [–e *ename*] [–`nlr`] [*first*] [*last*]
†† `fc` –e – [*old=new*] [*command*]
In the first form, a range of commands from *first* to *last* is selected from the last `HISTSIZE` commands that were typed at the terminal. The arguments *first* and *last* may be specified as a number or as a string. A string is used to locate the most recent command that begins with the given string. A negative number is used as an offset to the current command number. If the flag `-l`, is selected, the commands are listed on standard output. Otherwise, the editor program *ename* is invoked on a file containing these keyboard commands. If *ename* is not supplied, then the value of the parameter `FCEDIT` (default `/bin/ed`) is used as the editor. When editing is complete, the edited command(s) is executed. If *last* is not specified then it will be set to *first*. If *first* is not specified, the default will be the previous command for editing and –16 for listing. The flag `-r` reverses the order of the commands and the flag `-n`, when listing, suppresses the command numbers. In the second form the *command* is re-executed after the substitution *old=new* is performed.

`fg` [*%job*]
If *job* is specified, `fg` brings it to the foreground. Otherwise, the current job is brought into the foreground.

`jobs` [`-l`]
Lists the active jobs; when given the `-l` option, it lists process IDs in addition to the normal information.

`kill` [*-sig*] *process* ...
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`). The signal numbers and names are listed by `kill -l`. If the signal being sent is `TERM` (terminate) or `HUP` (hangup), then the job or process will be sent a `CONT` (continue) signal if it is stopped. The argument *process* can be either a process ID or a job.

`let` *arg* ...
Each *arg* is an *arithmetic expression* to be evaluated. All calculations are executed with long integers and no check for overflow is performed. Expressions consist of constants, named parameters, and operators. The following set of operators, listed in order of decreasing precedence, have been implemented:

`-`	unary minus
`!`	logical negation
`* / %`	multiplication, division, remainder
`+ -`	addition, subtraction
`<= >= < >`	comparison
`== !=`	equality inequality
`=`	arithmetic replacement

Subexpressions in parentheses `()` are evaluated first and can be used to override the precedence rules as listed. The evaluation within a precedence group is from right to left for the = operator and from left to right for the others.

A parameter name must be a valid *identifier*. When a parameter is encountered, the value associated with the parameter name is substituted and expression evaluation resumes. Up to 9 levels of recursion are permitted.

The return code is 0 if the value of the last expression is nonzero, and 1 if otherwise.

†† `newgrp` [*arg* ...]
Equivalent to `exec newgrp` *arg* ...

`print` [-Rnprsu[*n*]] [*arg* ...]
The shell output mechanism. With no flags or with the flag -, the arguments are printed on standard output as described by `echo`(1). In raw mode, `-R` or `-r`, the escape conventions of `echo` are ignored. The `-R` option will print all subsequent arguments and options other than `-n`. The `-p` option causes the arguments to be written onto the pipe of the process spawned with `|&` instead of standard output. The `-s` option causes the arguments to be written onto the history file instead of onto standard output. The `-u` flag option can be used to specify the one digit file descriptor unit number `n` on which the output will be placed. The default is 1. If the flag option `-n` is used, no newline is added to the output.

`pwd`
Equivalent to `print -r - $PWD`

`read` [-prsu [*n*]] [*name*?*prompt*] [*name* ...]
The shell input mechanism. One line is read and is broken up into words using the characters in `IFS` as separators. In raw mode, `-r`, a \ at the end of a line does not signify line continuation. The first word is assigned to the first *name*, the second word to the second *name*, and so on, with leftover words assigned to the last *name*. The `-p` option causes the input line to be taken from the input pipe of a process spawned by the shell using `|&`. If the `-s` flag is present, the input will be saved as a command in the history file. The flag `-u` can be used to specify a one digit file descriptor unit to read from. The file descriptor can be opened with the `exec` special command. The default value of *n* is 0. If *name* is omitted then `REPLY` is used as the default *name*. The return code is 0 unless an end-of-file is encountered. An end-of-file with the `-p` option causes cleanup for this process so another can be spawned. If the first argument contains a `?`, the remainder of this word is used as a *prompt* when the shell is interactive. If the given file descriptor is open for writing and is a terminal device, then the prompt is placed on this unit. Otherwise the prompt is issued on file descriptor 2. The return code is 0 unless an end-of-file is encountered.

†† `readonly` [*name* ...]
The given *name*s are marked "read only" and these names cannot be changed by subsequent assignment.

†† `return` [*n*]
Causes a shell *function* to return to the invoking script with the return status specified by *n*. If *n* is omitted then the return status is that of the last command executed. If `return` is invoked while not in a *function* or a . script, then it is the same as an `exit`.

`set` [`-aefhkmnostuvx`] [`-o` *option* ...] [*arg* ...]
The flags for this command have meaning as follows:

`-a` All subsequent parameters defined are automatically exported.

`-e` If the shell is noninteractive and if a command fails, execute the ERR trap, if set, and exit immediately. This mode is disabled while reading profiles.

`-f` Disables filename generation.

`-h` Each command whose name is an *identifier* becomes a tracked alias when first encountered.

`-k` All parameter assignment arguments are placed in the environment for a command, not just those that precede the command name.

`-m` Background jobs will run in a separate process group and a line will print upon completion. The exit status of background jobs is reported in a completion message. On systems with job control, this flag is turned on automatically for interactive shells.

`-n` Reads commands but does not execute them. Ignored for interactive shells.

`-o` The following argument can be one of the following names:

`allexport`	Same as `-a`.
`errexit`	Same as `-e`.
`bgnice`	All background jobs are run at a lower priority.

`emacs`	Enters into an `emacs` style inline editor for command entry.
`gmacs`	Enters into a `gmacs` style inline editor for command entry.
`ignoreeof`	The shell will not exit on end-of-file. The command `exit` must be used.
`keyword`	Same as `-k`.
`markdirs`	All directory names resulting from filename generation have a trailing / appended.
`monitor`	Same as `-m`.
`noexec`	Same as `-n`.
`noglob`	Same as `-f`.
`nounset`	Same as `-u`.
`protected`	Same as `-p`.
`verbose`	Same as `-v`.
`trackall`	Same as `-h`.
`vi`	Enters into insert mode of a `vi` style inline editor until the escape character 033 is used. This enables the move mode. A RETURN sends the line.
`viraw`	Each character is processed as it is typed in `vi` mode.
`xtrace`	Same as `-x`.
	If no flag option name is supplied, then the current settings are printed.

`-p` Resets the `PATH` variable to the 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).

`-s` Sorts the positional parameters.

`-t` Exits after reading and executing one command.

`-u` Treats unset parameters as an error when substitution is necessary.

`-v` Prints shell input lines as they are read.

`-x` Prints commands and their arguments as they are executed.

`-` Turns off `-x` and `-v` flags and stops examining arguments for flags.

`--` Does not change any of the flags; useful in setting `$1` to a value beginning with `-`. If no arguments follow this flag, then the positional parameters are unset.

Using `+` rather than `-` causes these flags to be turned off. These flags can also be used upon invocation of the shell. The current set of flags may be found in `$-`. The remaining arguments are positional parameters and are assigned, in order, to `$1 $2 ...`. If no arguments are given then the values of all names are printed on the standard output.

† `shift` [*n*]
The positional parameters from `$`*n*`+1 ...` are renamed `$1 ...`; default *n* is 1. The parameter *n* can be any arithmetic expression that evaluates to a non-negative number less than or equal to `$#`.

`test` [*expr*]
Evaluate conditional expression, *expr*. `test` evaluates the expression *expr* and, if its value is true, returns a zero (true) exit status; otherwise, a nonzero (false) exit status is returned; `test` also returns a nonzero exit status if there are no arguments. The superuser is always granted execute permission even though (1) execute permission is meaningful only for directories and regular files, and (2) `exec` requires that at least one execute mode bit be set for a regular file to be executable. The following primitives are used to construct *expr*.

`-r` *file* True if *file* exists and is readable.

`-w` *file* True if *file* exists and is writable.

`-x` *file*	True if *file* exists and is executable.
`-f` *file*	True if *file* exists and is a regular file.
`-d` *file*	True if *file* exists and is a directory.
`-c` *file*	True if *file* exists and is a character special file.
`-b` *file*	True if *file* exists and is a block special file.
`-p` *file*	True if *file* exists and is a named pipe (FIFO).
`-u` *file*	True if *file* exists and its set-user-ID bit is set.
`-g` *file*	True if *file* exists and its set-group-ID bit is set.
`-k` *file*	True if *file* exists and its sticky bit is set.
`-s` *file*	True if *file* exists and has a size greater than zero.
`-t` [*fildes*]	True if the open file whose file descriptor number is *fildes* (1 by default) is associated with a terminal device.
`-z` *s1*	True if the length of string *s1* is zero.
`-n` *s1*	True if the length of the string *s1* is nonzero.
s1 = *s2*	True if strings *s1* and *s2* are identical.
s1 != *s2*	True if strings *s1* and *s2* are *not* identical.
s1	True if *s1* is *not* the null string.
n1 `-eq` *n2*	True if the integers *n1* and *n2* are algebraically equal. Any of the comparisons: `-ne`, `-gt`, `-ge`, `-lt`, and `-le` may be used in place of `-eq`.
	These primaries may be combined with the following operators:
!	Unary negation operator.
`-a`	Binary AND operator.
`-o`	Binary OR operator (`-a` has higher precedence than `-o`).
(*expr*)	Parentheses for grouping.
	Notice that all the operators and flags are separate arguments to `test`. Notice also that parentheses are meaningful to the shell and

therefore, must be escaped.

`test` is typically used in shell scripts as in the following example, which prints the message "`foo is a directory`" if it is found to be one when `test` is run.

```
if test -d foo
then
     echo "foo is a dir"
fi
```

the arithmetic comparison operators are not restricted to integers. They allow any arithmetic expression. Four additional primitive expressions are allowed:

`-L` *file* True, if *file* is a symbolic link.

file1 `-nt` *file2* True, if *file1* is newer than *file2*.

file1 `-ot` *file2* True, if *file1* is older than *file2*.

file1 `-ef` *file2* True, if *file1* has the same device and inode number as *file2*.

`times`

Prints the accumulated user and system times for the shell and for processes run from the shell.

`trap [` *arg* `] [` *sig* `] ...`

The command *arg* is to be read and executed when the shell receives signal(s) *sig*. (Note that *arg* is scanned once when the trap is set and once when the trap is taken.) Each *sig* can be given as a number or as the name of the signal. Trap commands are executed in order of signal number. Any attempt to set a trap on a signal that was ignored on entry to the current shell is ineffective. If *arg* is omitted or is –, then all trap(s) *sig* are reset to their original values. If *arg* is the null string then this signal is ignored by the shell and by the commands it invokes. If *sig* is `ERR` then *arg* will be executed whenever a command has a nonzero exit code. This trap is not inherited by

functions. If *sig* is `0` or `EXIT` and the `trap` statement is executed inside the body of a function, then the command *arg* is executed after the function completes. If *sig* is `0` or `EXIT` for a `trap` set outside any function, then the command *arg* is executed on exit from the shell. The `trap` command with no arguments prints a list of commands associated with each signal number.

†† `typeset` [`-HLRZfilprtux` [*n*] [*name* [`=`*value*]] ...]
When invoked inside a function, a new instance of the parameter *name* is created. The parameter value and type are restored when the function completes. The following list of attributes may be specified:

`-H` This flag provides A/UX to host namefile mapping on non-UNIX® machines.

`-L` Justifies left and removes leading blanks from *value*. If *n* is nonzero, it defines the width of the field, otherwise the width is determined by the width of the value of first assignment. When the parameter is assigned to, it is filled on the right with blanks or truncated, if necessary, to fit into the field. Leading zeros are removed if the `-Z` flag is also set. The `-R` flag is turned off.

`-R` Justifies right and fills with leading blanks. If *n* is nonzero, it defines the width of the field, otherwise the width is determined by the width of the value of first assignment. The field is filled with blanks or truncated from the end if the parameter is reassigned. The `L` flag is turned off.

`-Z` justifies right and fills with leading zeros if the first nonblank character is a digit and the `-L` flag has not been set. If the `-L` flag has been set, the field is

left adjusted and any leading zeros are removed. If *n* is nonzero, it defines the width of the field; otherwise the width is determined by the width of the value of first assignment.

`-f` The names refer to function names rather than parameter names. No assignments can be made and the only other valid flags are `-t`, which turns on execution-tracing for this function and `-x`, to allow the function to remain in effect across shell procedures executed in the same process environment.

`-i` Parameter is an integer. This makes arithmetic faster. If *n* is nonzero it defines the output arithmetic base, otherwise the first assignment determines the output base.

`-l` All uppercase characters converted to lowercase. The uppercase flag, `-u` is turned off.

`-p` The output of this command, if any, is written onto the two-way pipe.

`-r` The given *names* are marked read only and these names cannot be changed by subsequent assignment.

`-t` Tags the named parameters. Tags are user-definable and have no special meaning to the shell.

`-u` All lowercase characters are converted to uppercase characters. The lowercase flag, `-l` is turned off.

`-x` The given *names* are marked for automatic export to the *environment* of subsequently-executed commands.

Using + rather than – causes these flags to be turned off. If no *name* arguments are given but flags are specified, a list of *names* (and optionally the *values*) of the *parame-*

ters which have these flags set is printed. (Using + rather than – keeps the values to be printed.) If no *names* and flags are given, the *names* and *attributes* of all *parameters* are printed.

`ulimit` [`-f`] [*n*]

`-f` Imposes a size limit of *n* 512-byte blocks on files written by child processes (files of any size may be read). This is the default. If *n* is not given, the current limit is printed.

When using the `ulimit` feature, a regular user is only allowed to bring resource limits down. However, only the superuser can return the limit to a higher status.

`umask` [*nnn*] The user file-creation mask is set to *nnn* (see `umask`(2)). If *nnn* is omitted, the current value of the mask is printed.

`unalias` *name* ...
The *parameters* given by the list of *names* are removed from the `alias` list.

`unset` [`-f`] *name* ...
The parameters given by the list of *names* are unassigned. That is, their values and attributes are erased. Read only variables cannot be unset. If the flag, `-f`, is set, then the names refer to *function* names.

`wait` [*n*] Waits for the specified child process and reports its termination status. If *n* is not given then all currently active child processes are waited for. The return code from this command is that of the process waited for.

`whence` [`-v`] *name* ...
For each *name*, indicate how it would be interpreted if used as a command name.

The flag, `-v`, produces a more verbose report.

Invocation

If the shell is invoked by exec(2), and the first character of argument zero ($0) is –, then the shell is assumed to be a login shell and commands are read from /etc/profile and then from either .profile in the current directory or $HOME/.profile, if either file exists. Next, commands are read from the file named by performing parameter substitution on the value of the environment parameter ENV if the file exists. If the –s flag is not present and *arg* is, then a path search is performed on the first *arg* to determine the name of the script to execute. The script *arg* must have read permission and any setuid and setgid settings will be ignored. Commands are then read as described later; the following flags are interpreted by the shell when it is invoked.

–c *string* If the –c flag is present then commands are read from *string*.

–s If the –s flag is present or if no arguments remain, then commands are read from the standard input. Shell output, except for the output of the "Special Commands" listed earlier, is written to file descriptor 2.

–i If the –i flag is present or if the shell input and output are attached to a terminal (as told by ioctl(2)) then this shell is *interactive*. In this case TERM is ignored (so that kill 0 does not kill an interactive shell) and INTR is caught and ignored (so that wait is interruptible). In all cases, QUIT is ignored by the shell.

–r If the –r flag is present, the shell is a restricted shell and is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. The actions of the restricted shell are identical to those of ksh, except that changing the directory, setting the value of SHELL, ENV, or PATH, specifying path or command names containing /, and redirecting output (> and >>) are disallowed.

The restrictions above are enforced after .profile and the ENV files are interpreted.

When a command to be executed is found to be a shell procedure, the restricted shell invokes `ksh` to execute it. Thus, it is possible to provide to the end-user shell procedures that have access to the full power of the standard shell, while imposing a limited menu of commands; this scheme assumes that the end-user does not have write and execute permissions in the same directory.

The net effect of these rules is that the writer of the `.profile`, by performing guaranteed setup actions and leaving the user in an appropriate directory (probably *not* the login directory), has complete control over user actions.

The system administrator often sets up a directory of commands (for example, `/usr/rbin`) that can be safely invoked by the restricted shell. Some systems also provide a restricted editor `red`.

The remaining flags and arguments are described under the `set` command earlier.

EXIT STATUS

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 `exit` command earlier). If the shell is being used noninteractively, then execution of the shell file is abandoned. Runtime errors detected by the shell are reported by printing the command or function name and the error condition. If the line number that the error occurred on is greater than one, then the line number is also printed in square brackets (`[]`) after the command or function name.

FILES

```
/bin/ksh
/etc/passwd
/etc/profile
/etc/suid_profile
$HOME/.profile
/tmp/ksh*
/dev/null
```

SEE ALSO

`cat`(1), `csh`(1), `echo`(1), `ed`(1), `env`(1), `newgrp`(1), `sh`(1), `vi`(1), `dup`(2), `exec`(2), `fork`(2), `ioctl`(2), `lseek`(2), `pipe`(2), `umask`(2), `ulimit`(2), `wait`(2), `rand`(3), `signal`(3), `a.out`(4), `profile`(4), `environ`(5).
"Korn Shell Reference" in *A/UX User Interface*.

CAVEATS

If a command which 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 will continue to `exec` the original command. Use the `-t` option of the `alias` command to correct this situation.

Some very old shell scripts contain a `^` as a synonym for the pipe character `|`.

If a command is piped into a shell command, then all variables set in the shell command are lost when the command completes.

Using the `fc` built-in command within a compound command will cause the whole command to disappear from the history file.

The built-in command *.file* reads the whole file before any commands are executed. Therefore, `alias` and `unalias` commands in the file will not apply to any functions defined in the file.

NAME

last — display login and logout times for each user of the system

SYNOPSIS

last [*name* ...] [*tty* ...]

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.

last 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.

EXAMPLES

```
last reboot
```

will give an indication of mean time between reboots of the system.

last 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.

FILES

/usr/bin/last
/etc/wtmp

SEE ALSO

acct(1M), utmp(4).

NAME

`launch` — execute a Macintosh binary application

SYNOPSIS

`launch -[it]` *filename* [*document...*]
`launch -p[it]` *filename document...*

DESCRIPTION

`launch` executes a Macintosh binary file in A/UX.

filename is the name of the application to be executed. *document* is an individual document to be opened.

The *filename* and *document* parameters are set up as if they were icons selected through the Macintosh Finder™. Thus, the A/UX command

```
launch macpaint
```

is equivalent to double clicking on the icon for MacPaint. The A/UX command

```
launch macpaint mydwg
```

is equivalent to double clicking on the icon for the MacPaint document `mydwg`.

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.

You can specify one or both of these two options:

`-i` Initializes QuickDraw™, the Dialog Manager, and TextEdit. You must specify this option if the application does not explicitly initialize these libraries. (In the native Macintosh environment, the Finder initializes these libraries during startup. Therefore, some Macintosh applications do not explicitly initialize them.)

`-t` Sets up and maintains the Ticks, Time, and KeyMap low-memory global variables, which are not ordinarily supported in the A/UX Toolbox.

The `-t` option uses the Vertical Retrace Manager to set up a task that is invoked at every tick of the clock. Therefore, this option uses a lot of CPU time and should be used only if you are running a Macintosh binary file that requires to use one or more of these low-memory global variables.

Alternatively, you can specify the print option

-p Prints the specified document. The -p option requires 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.

If all of the application's code resides in an AppleDouble header file in A/UX (see Appendix B of the *A/UX Toolbox* document), you can make the associated data file either a copy of `launch` or a link to `launch`, setting up the simplest and most natural way to run a Macintosh application. For example, consider an application named `xyz`; the AppleDouble data file of `xyz` has the A/UX filename `xyz` and is a link to `launch`. The AppleDouble header file of `xyz` has the A/UX filename `%xyz` and contains a binary copy of the resource fork of the Macintosh file `xyz`.

To launch `xyz`, type this command

```
xyz
```

To launch `xyz` and specify a document file `abc`, type this command

```
xyz abc
```

Entering this command is equivalent to double-clicking the icon for the file `abc` from the Macintosh Finder.

EXAMPLES

The command

```
launch macpaint
```

executes the Macintosh binary application MacPaint®.

```
launch macpaint demo
```

executes MacPaint and opens the document `demo`.

FILES

`/mac/bin/launch`

NAME

lav — display 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

SEE ALSO

ruptime(1), uptime(1).

NAME

ld — link editor for common object files

SYNOPSIS

ld [-e*epsym*] [-f*fill*] [-l*x*] [-m] [-o*outfile*] [-r] [-s] [-t] [-u*symname*] [-x] [-z] [-F] [-L*dir*] [-M] [-N] [-V] [-VS*num*] *file* ...

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 it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. The library (archive) symbol table (see 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.

FLAG OPTIONS

The following flag options are recognized by ld:

-e*epsym*
: Set the default entry-point address for the output file to be that of the symbol *epsym*.

-f*fill*
: Set the default fill pattern for holes within an output section as well as initialized bss sections. The argument *fill* is a 2-byte constant.

-l*x* Search a library libx.a, where *x* is up to seven characters. A library is searched when its name is encountered, so the placement of a -l is significant. The default library location is /lib.

-m Produce a map or listing of the input/output sections on the standard output.

-o*outfile*
Produce an output object file by the name *outfile*. The name of the default object file is `a.out`.

-r Retain 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 Strip line-number entries and symbol-table information from the output object file.

-t Turn off the warning about multiply-defined symbols that are not the same size.

-u*symname*
Enter *symname* as an undefined symbol in the symbol table. This 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.

-x Do not 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 Load the text segment at an offset from 0 so that null pointer references generate a segmentation violation.

-F Create a demand-paged executable.

-L*dir*
Change the algorithm of searching for `lib`*x*`.a` to look in *dir* before looking in `/lib` and `/usr/lib`. This flag option is effective only if it precedes the -l flag option on the command line.

-M Produce an output message for each multiply-defined external definition. However, if the objects being loaded include debugging information, extraneous output is produced. (See the -g option in `cc`(1).

-N Put 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.

-V Produce an output a message giving information about the version of `ld` being used.

–VS*num*
Use *num* as a decimal version stamp identifying the `a.out` file that is produced. The version stamp is stored in the optional header.

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 (`.data` and `.bss` combined) 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, allowing you to 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 (`.data` and `.bss` combined) 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` flag 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 below. 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 2K-byte boundaries, 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 2*.

FILES

`/bin/ld`
`/lib/*`
`/usr/lib/*`
`a.out` default output file

SEE ALSO

`as`(1), `cc`(1), `a.out`(4), `ar`(4),
"`ld` Reference" in *A/UX Programming Languages and Tools, Volume 2*.

WARNINGS

Through its flag 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 insure 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, 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 insure that the program always calls exit( ), rather than falling through the end of the entry routine.

NAME

leave — remind you when you have to leave

SYNOPSIS

leave [*hhmm*]

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.

The time of day is in the form *hhmm*, where *hh* is a time in hours (on a 12 or 24 hour clock) and *mm* is a time in minutes. All times are converted to a 12 hour clock, and assumed to be in the next 12 hours.

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.

leave 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

SEE ALSO

calendar(1).

NAME

lex — generate programs for simple lexical tasks

SYNOPSIS

lex [-c] [-n] [-t] [-v] [*file*] ...

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 occurrences 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 non-blank 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`.

The flags must appear before any files. The `-c` flag option indicates C actions and is the default, `-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, `-n` will not print out the summary. Multiple files are treated as a single file. If no files are specified, standard input is used.

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` flag option, unless the `-n` flag option is used.

EXAMPLES

```
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;
                                 }
                       }
```

FILES

/usr/bin/lex

SEE ALSO

awk(1), grep(1), sed(1), yacc(1), malloc(3X).
"lex Reference" in the *A/UX Programming Languages and Tools, Volume 2.*

BUGS

When given an illegal flag option, lex reports the fact that it has been given an illegal flag option but then continues to execute with the default options, rather than stopping the execution and printing a usage statement.

NAME

line — read one line

SYNOPSIS

```
line
```

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

```
line
Hello world
```

will return

```
Hello world
```

In the Bourne shell (sh(1)):

```
a=`line`
hi there
echo $a
```

will return

```
hi there
```

In the C-shell (csh(1)):

```
set a=`line`
bye bye
echo $a
```

will return

```
bye bye
```

FILES

/bin/line

SEE ALSO

csh(1), ksh(1), sh(1), read(2).

NAME

lint — 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...*

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 flag option used. The .ln files are analogous to .o (object) files that are produced by the cc(1) command when given a .c file as input. Files with other suffixes are warned about and ignored.

lint 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 flag 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 flag options may be used, in any order, intermixed with filename arguments. The following flag options are used to suppress certain kinds of complaints.

-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).

-h Does not apply heuristic tests that attempt to intuit bugs, improve style, and reduce waste.

-u Suppresses complaints about functions and external variables used and not defined, or defined and not used. (This flag option is suitable for running `lint` on a subset of files of a larger program).

-v Suppresses complaints about unused arguments in functions.

-x Does not report variables referred to by external declarations but never used.

The following arguments alter the behavior of `lint` .

-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 flag 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.

-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.

-o *lib*

Cause `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`.

The `-o` flag 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` flag option is suggested.

The `-v` flag 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 flag option settings are also available through the use of `lint` *comments* (as shown later in this section).

The `-D`, `-U`, and `-I` flag options of `cpp`(1) and the `-g` and `-O` flag options of `cc`(1) are also recognized as separate arguments. The `-g` and `-O` flag options are ignored, but, by recognizing these flag options, the behavior of `lint` is closer to that of the `cc`(1) command. Other flag 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` flag option for the next function.

`/*LINTLIBRARY*/`
: at the beginning of a file, shuts off complaints about unused functions and function arguments in this file. This is equivalent to using the `-v` and `-x` flag 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` flag option indicates that unreachable `break` statements are not to be checked. This flag option might well be used on files that `lex`(1) generates.

FILES

`/usr/bin/lint`	
`/usr/lib`	the directory where the `lint` libraries specified by the `-l`*x* flag option must exist
`/usr/lib/lint[12]`	first and second passes
`/usr/lib/llib-lc.ln`	declarations for C Library functions (binary format)
`/usr/lib/llib-port.ln`	declarations for portable functions (binary format)
`/usr/lib/llib-lm.l`	declarations for Math Library functions (binary format)
`/usr/tmp/*lint*`	temporaries

SEE ALSO

`cc`(1), `cpp`(1), `make`(1),
"`lint` Reference" in *A/UX Programming Languages and Tools, Volume 1*.

BUGS

`exit`(2), `longjmp`(3C), and other functions that do not return are not understood; this causes various lies.

NAME

ln — make links

SYNOPSIS

ln [-s] *name1* [*name2*]

ln *name... directory*

ln -f *directory1 directory2*

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.

The -s flag option causes ln to create symbolic links. A symbolic link contains the name of the file to which it is linked. The referenced file is used when an open(2) operation is performed on the link. A stat(2) on a symbolic link will return the linked-to file; an lstat(2) must be done to obtain information about the link. The readlink(2) call may be used to read the contents of a symbolic link. Symbolic links may span file systems and may refer to directories.

ln may be invoked with one, two, or more than two arguments. If given one argument, ln creates a link in the current directory to *name1*. The file named by *name1* must not already exist in the current directory, or ln will exit with the message *name1*: File exists.

Given two arguments, ln creates a link to an existing file *name1* having the name *name2*. The argument *name2* 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 *name1*. If *name1* is not found, ln will so indicate and no link will be created. If *name2* 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.

The `-f` flag option causes `ln` to make a hard link to an existing directory. 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. Only the superuser is permitted to use this option.

FILES

`/bin/ln`

SEE ALSO

`cp`(1), `mv`(1), `rm`(1), `link`(2), `lstat`(2), `readlink`(2), `stat`(2), `symlink`(2).

NAME

login — sign on

SYNOPSIS

login [*name* [*env-var*...]]

DESCRIPTION

The login command 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]
```

login 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, a flag 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(1) or ksh(1), 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(1), 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(2) 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 arguments may take either the form *xxx* or *xxx*=*yyy*. Arguments without an equals sign are 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. login 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.

FILES

`/bin/login`	
`/etc/utmp`	accounting
`/etc/wtmp`	accounting
`/etc/motd`	message-of-the-day
`/etc/passwd`	password file
`/etc/profile`	systemwide personal profile (`sh`(1) and `ksh`(1))
`/etc/cshrc`	systemwide personal `csh` startup (`csh`(1))
`$HOME/.profile`	personal profile (`sh`(1) and `ksh`(1))
`$HOME/.login`	personal `csh` startup used at login time (`csh`(1))
`$HOME/.cshrc`	personal `csh` startup (`csh`(1))
`$HOME/.logout`	personal `csh` logout used at logout time (`csh`(1))
`/usr/mail/`*name*	mailbox for user *name*

SEE ALSO

`csh`(1), `ksh`(1), `mail`(1), `newgrp`(1), `rlogin`(1), `sh`(1), `su`(1), `getty`(1M), `init`(1M), `passwd`(4), `profile`(4), `environ`(5).
A/UX Essentials.
A/UX User Interface.

DIAGNOSTICS

`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 lowest level sh.`
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)).

NAME

logname — get 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

```
logname
```

displays the $LOGNAME of the user logged into the system on the current port.

FILES

/bin/logname
/etc/profile

SEE ALSO

env(1), login(1), printenv(1), logname(3X), environ(5).

NAME

lookbib — find references in a bibliography

SYNOPSIS

lookbib [-n] *database*

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 %.

lookbib 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.

lookbib will ask if you need instructions and will print some brief information if you reply y. The -n flag option turns off the prompt for instructions.

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

x.ia, *x*.ib, *x*.ic, where *x* is the first argument, or if these are not present, then *x*.ig.

SEE ALSO

addbib(1), indxbib(1), refer(1), roffbib(1), sortbib(1).

NAME

lorder — find ordering relation for an object library

SYNOPSIS

lorder *file* ...

DESCRIPTION

lorder produces a global cross-reference, given a list of object modules (.o files), which can then be passed to tsort(1) 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(1) to find an ordering of a library suitable for one-pass access by ld(1).

> *Note:* The link editor ld(1) is capable of multiple passes over an archive in the portable archive format (see ar(4)) and does not require that lorder(1) be used when building an archive.

Use of the lorder(1) command may, however, allow for a slightly more efficient access of the archive during the link edit process.

EXAMPLES

```
ar cr library `lorder *.o | tsort`
```

builds a new library from existing .o files.

FILES

/bin/lorder
*symref
*symdef

SEE ALSO

ar(1), ld(1), tsort(1), ar(4),

BUGS

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.

NAME

lp, cancel — send or cancel requests to a line printer for a Berkeley file system (4.2)

SYNOPSIS

lp [-c] [-d*dest*] [-m] [-n*number*] [-o*option*] [-s] [-t*title*] [-w] [*file...*]

cancel *jobno...* [*printers*]

cancel *printers* [*jobno*]...

DESCRIPTION

lp arranges for the named files and associated information to be printed by a line printer. Note that *files* must be readable by the lp user account since /usr/bin/lp is set to change the effective user ID to lp. If the permissions on *files* do not allow lp to read them, *files* must be piped to lp(1) to print them. (For example, cat *files* | lp).

If no filenames are mentioned, the standard input is assumed. The filename - stands for the standard input and may be supplied on the command line in conjunction with named *files*. The order in which *files* appear is the same order in which they will be printed.

lp associates a unique *jobno* with each request and prints it on the standard output. This *jobno* can be used later to cancel (described later in this section) or find the status (see lpstat(1)) of the request.

The following flag options to lp may appear in any order and may be intermixed with filenames.

-c Makes copies of the *files* to be printed immediately after lp is invoked. Normally, *files* will not be copied, but will be linked whenever possible. If the -c flag option is not given, then the user should be careful not to remove any of the *files* before the request has been printed in its entirety. It should also be noted that in the absence of the -c flag option, any changes made to the named *files* after the request is made but before it is printed will be reflected in the printed output.

-d*dest* Chooses *dest* as the printer or class of printers selected to do the printing. If *dest* is a printer, then the request will be printed only on that specific printer. If *dest* is a class of printers, then the request will be

printed on the first available printer that is a member of the class. Under certain conditions (printer unavailability, file space limitation, and so forth), requests for specific destinations may not be accepted (see `accept`(1M) and `lpstat`(1)). By default, *dest* is taken from the environment variable `LPDEST` (if it is set). Otherwise, a default destination (if one exists) for the computer system is used. Destination names vary between systems (see `lpstat`(1)).

`-m` Sends mail via `mail`(1) after the files have been printed. By default, no mail is sent upon normal completion of the print request.

`-n`*number* Prints *number* copies (default of 1) of the output.

`-o`*option* Specifies the printer-dependent or class-dependent *options*. Several such *options* may be collected by specifying the `-o` key character more than once. For more information about what is valid for *options*, see "Models" in `lpadmin`(1M).

`-s` Suppresses messages from `lp`(1) such as `request id is...`.

`-t`*title* Prints *title* on the banner page of the output.

`-w` Writes a message on the user's terminal after the *files* have been printed. If the user is not logged on, then mail will be sent instead.

`cancel` cancels line printer requests that were made by the `lp`(1) command. The command line arguments may be either request *ids* (as returned by `lp`(1)) or *printer* names (for a complete list, use `lpstat`(1)). Specifying a request *id* cancels the associated request even if it is currently printing. Specifying a *printer* cancels the request which is currently printing on that printer. In either case, the cancellation of a request that is currently printing frees the printer to print its next available request.

NOTES

`lp` is an AT&T command originally intended for use with line printers, but flexible enough to be useful with other devices. `lpr`(1) performs a parallel function, but is nevertheless a distinct command.

FILES

/usr/bin/lp
/usr/bin/cancel
/usr/spool/lp/*

SEE ALSO

enable(1), lpq(1), lpr(1), lpstat(1), mail(1),
accept(1M), lpadmin(1M), lpsched(1M).
"Managing Peripherals" in *A/UX Local System Administration.*

NAME

lpq — spool queue examination program

SYNOPSIS

lpq [+[*n*]] [-l] [-P*printer*] [*job #* ...] [*user* ...]

DESCRIPTION

lpq examines the spooling area used by lpd(1M) for printing files on the line printer and reports the status of the specified jobs or all jobs associated with a user. lpq invoked without any arguments reports on any jobs currently in the queue.

For each job submitted (for example, an invocation of lpr(1)) lpq reports the user's name, current rank in the queue, the names of files comprising the job, the job identifier (a number which may be supplied to lprm(1) for removing a specific job), and the total size in bytes.

If lpq warns that no daemon is present due to some malfunction, the lpc(1M) command can be used to restart the printer daemon.

FLAG OPTIONS

The following flag options are interpreted by lpq:

-P Specifies a particular *printer*, otherwise the default line printer is used (or the value of the PRINTER variable in the environment).

+*n* Displays the spool queue until it empties. Supplying a number immediately after the + sign indicates that lpq should sleep *n* seconds in between scans of the queue.

-l Prints information about each of the files comprising the job. Normally, only as much information as fits on one line is displayed. Job ordering is dependent on the algorithm used to scan the spooling directory and is supposed to be FIFO (First in First Out). Filenames comprising a job may be unavailable when lpr(1) is used as a sink in a pipeline, in which case the file is indicated as "(standard input)".

All other arguments supplied are interpreted as user names or job numbers to filter out only those jobs of interest.

FILES

/etc/termcap
To manipulate the screen for repeated display

/etc/printcap
To determine printer characteristics

`/usr/spool/*`
The spooling directory, as determined from `printcap`

`/usr/spool/*/cf*`
Control files specifying jobs

`/usr/spool/*/lock`
The lock file to obtain the currently active job

SEE ALSO

`lpr`(1), `lprm`(1), `lpc`(1M), `lpd`(1M).

BUGS

Due to the dynamic nature of the information in the spooling directory `lpq` may report unreliably. Output formatting is sensitive to the line length of the terminal; this can result in widely spaced columns.

DIAGNOSTICS

`lpq` may report that it is unable to open various files.

The lock file may be malformed.

Garbage files found in the spooling directory may be printed when no daemon is active.

NAME

lpr — off line print

SYNOPSIS

lpr [-P*printer*] [-#*num*] [-C *class*] [-J *job*] [-T *title*] [-i [*numcols*]] [-1234*font*] [-w*num*] [-pltndgvcfrmhs] [*name* ...]

DESCRTPTION

lpr uses a spooling daemon to print the named files when facilities become available. If no names appear, the standard input is assumed. The -P option may be used to force output to a specific printer. Normally, the default printer is used (site dependent), or the value of the environment variable PRINTER is used.

The following single letter options are used to notify the line printer spooler that the files are not standard text files. The spooling daemon will use the appropriate filters to print the data accordingly.

-p Use pr(1) to format the files (equivalent to print).

-l Use a filter which allows control characters to be printed and suppresses page breaks.

-t The files are assumed to contain data from troff(1) (cat phototypesetter commands).

-n The files are assumed to contain data from ditroff (device independent troff).

-d The files are assumed to contain data from tex(l) (DVI format from Stanford).

-g The files are assumed to contain standard plot data as produced by the plot(3X) routines (see also plot(1G) for the filters used by the printer spooler).

-v The files are assumed to contain a raster image for devices like the Benson Varian.

-c The files are assumed to contain data produced by cifplot(l).

-f Use a filter which interprets the first character of each line as a standard FORTRAN carriage control character.

The remaining single letter options have the following meaning.

`-r` Remove the file upon completion of spooling or upon completion of printing (with the `-s` option).

`-m` Send mail upon completion.

`-h` Suppress the printing of the burst page.

`-s` Use symbolic links. Usually files are copied to the spool directory.

The `-C` option takes the following argument as a job classification for use on the burst page. For example,

```
lpr -C EECS foo.c
```

causes the system name (the name returned by `hostname`(1)) to be replaced on the burst page by `EECS`, and the file `foo.c` to be printed.

The `-J` option takes the following argument as the job name to print on the burst page. Normally, the first file's name is used.

The `-T` option uses the next argument as the title used by `pr`(1) instead of the file name.

To get multiple copies of output, use the *-#num* option, where *num* is the number of copies desired of each file named. For example,

```
lpr -#3 foo.c bar.c more.c
```

would result in 3 copies of the file `foo.c`, followed by 3 copies of the file `bar.c`, etc. On the other hand,

```
cat foo.c bar.c more.c | lpr -#3
```

will give three copies of the concatenation of the files.

The `-i` option causes the output to be indented. If the next argument is numeric, it is used as the number of blanks to be printed before each line; otherwise, 8 characters are printed.

The `-w` option takes the immediately following number to be the page width for `pr`.

The `-s` option will use `symlink`(2) to link data files rather than trying to copy them so large files can be printed. This means the files should not be modified or removed until they have been printed.

The option -1234 Specifies a font to be mounted on font position i. The daemon will construct a .railmag file referencing /usr/lib/vfont/name.size.

FILES

/etc/passwd
personal identification

/etc/printcap
printer capabilities data base

/usr/lib/lpd*
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), symlink(2), printcap(4), lpc(1M), lpd(1M).

DIAGNOSTICS

If you try to spool too large a file, it will be truncated. lpr objects to printing binary files. 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 machine cannot be made, lpr says that the daemon cannot be started. Diagnostics may be printed in the daemon's log file regarding missing spool files by lpd.

BUGS

Fonts for troff and tex reside on the host with the printer. It is currently not possible to use local font libraries.

NAME

lprm — remove jobs from the line printer spooling queue for a Berkeley file system (4.2)

SYNOPSIS

lprm [-P*printer*] [-] [*jobno*]... [*user*]...

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.

lprm 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 superuser.

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
```

lprm announces the names of any files it removes and is silent if there are no jobs in the queue that match the request list.

lprm 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.

FLAG OPTIONS

The following flag options are interpreted by lprm:

-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.

-
: Removes all jobs that a user owns. If the superuser employs this flag, 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.

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

lpd(1M), lpr(1), lpq(1).

DIAGNOSTICS

A "Permission denied" is received if the user tries to remove files other than his own.

BUGS

Since there are race conditions possible in the update of the lock file, the currently active job may be incorrectly identified.

NAME

lpstat — print LP status information

SYNOPSIS

lpstat [-a[*list*]] [-c[*list*]] [-d] [-o[*list*]] [-p[*list*]] [-r] [-s] [-t] [-u[*list*]] [-v[*list*]]

DESCRIPTION

lpstat prints information about the current status of the LP line printer system.

If no flag options are given, then lpstat prints the status of all requests made to lp(1) by the user. Any arguments that are not flag options are assumed to be request IDs (as returned by lp). lpstat prints the status of such requests. Flag options may appear in any order and may be repeated and intermixed with other arguments. Some of the options below 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.

-a[*list*] Print acceptance status (with respect to lp) of destinations for requests. *list* is a list of intermixed printer names and class names.

-c[*list*] Print class names and their members. *list* is a list of class names.

-d Print the system default destination for lp.

-o[*list*] Print the status of output requests. *list* is a list of intermixed printer names, class names, and request IDs.

-p[*list*] Print the status of printers. *list* is a list of printer names.

-r Print the status of the LP request scheduler.

-s Print 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 Print all status information.

-u[*list*] Print status of output requests for users. *list* is a list of login names.

-v[*list*] Print the names of printers and the pathnames of the devices associated with them. *list* is a list of printer names.

FILES

/usr/bin/lpstat
/usr/spool/lp/*

SEE ALSO

enable(1), lp(1), lpq(1).

NAME

ls — list contents of directory

SYNOPSIS

ls [-R] [-a] [-d] [-C] [-x] [-m] [-l] [-L] [-n] [-o] [-g] [-r] [-t] [-u] [-c] [-p] [-F] [-b] [-q] [-i] [-s] [*names*]

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 flag options enable multicolumn formats, and the -m flag 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 flag 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.

There are many flag options, as follows.

-R Recursively list subdirectories encountered.

-a List all entries; usually entries whose names begin with a period (.) are not listed.

-d If an argument is a directory, list its name only (not its contents); often used with -l to get the status of a directory.

-L If an argument is a symbolic link, list the file or directory the link references rather than the link itself.

-C Multicolumn output with entries sorted vertically.

-x Multicolumn output with entries sorted horizontally, rather than down the page.

-m Stream output format.

`-l` List 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 ->.

`-n` The same as `-l`, except that the owner's user ID and group's group ID numbers, rather than the associated character strings, are printed.

`-o` The same as `-l`, except that the group is not printed.

`-g` The same as `-l`, except that the owner is not printed.

`-r` Reverse the order of sort to get reverse alphabetic or oldest first, as appropriate.

`-t` Sort by time modified (latest first), instead of by name.

`-u` Use time of last access, instead of last modification, for sorting (with the `-t` flag option) or printing (with the `-l` flag option).

`-c` Use time of last modification of the i-node (file created, mode changed, and so forth) for sorting (`-t`) or printing (`-l`).

`-p` Put a slash (`/`) after each filename if that file is a directory.

`-F` Put 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.

`-b` Force printing of nongraphic characters to be in the octal *ddd* notation.

`-q` Force printing of nongraphic characters in filenames as the character (`?`).

`-i` For each file, print the i-number in the first column of the report.

`-s` Give size in blocks, including indirect blocks, for each entry.

The mode printed under the `-l` flag option consists of 10 characters that are interpreted as follows:

The first character is:

`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.

EXAMPLE

```
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.

FILES

/bin/ls	
/etc/passwd	to get user IDs for ls -l and ls -o

/etc/group	to get group IDs for `ls -l` and `ls -g`
/usr/lib/terminfo/*	to get terminal information.

SEE ALSO

chgrp(1), chown(1), chmod(1), find(1).

BUGS

Unprintable characters in filenames may confuse the columnar output options.

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh® computers and troff running on A/UX. Proof and final pages were created on Apple LaserWriter® printers. POSTSCRIPT®, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type and display type are Times and Helvetica. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

Writers: J. Eric Akin, Mike Elola, George Towner, and Kathy Wallace

Editor: George Truett

Production Supervisor: Josephine Manuele

Acknowledgments: Lori Falls and Michael Hinkson

Special thanks to Lorraine Aochi, Vicki Brown, Sharon Everson, Pete Ferrante, Kristi Fredrickson, Don Gentner, Tim Monroe, Dave Payne, Henry Seltzer, and John Sovereign

030-0781