ULTRIX

Reference Pages Section 1: Commands A - L

Order Number: AA-PC0WA-TE
June 1990

Product Version: ULTRIX Version 4.0 or higher

This manual describes commands from A to L that are available to all ULTRIX users for both RISC and VAX platforms.

digital equipment corporation
maynard, massachusetts

Restricted Rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause of DFARS 252.227-7013.

© Digital Equipment Corporation 1984, 1986, 1988, 1990
All rights reserved.

Portions of the information herein are derived from copyrighted material as permitted under license agreements with AT&T and the Regents of the University of California. © AT&T 1979, 1984. All Rights Reserved.

Portions of the information herein are derived from copyrighted material as permitted under a license agreement with Sun MicroSystems, Inc. © Sun MicroSystems, Inc, 1985. All Rights Reserved.

Portions of this document © Massachusetts Institute of Technology, Cambridge, Massachusetts, 1984, 1985, 1986, 1988.

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital or its affiliated companies.

The following are trademarks of Digital Equipment Corporation:

digital
CDA
DDIF
DDIS
DEC
DECnet
DECstation
DECUS
DECwindows
DTIF
MASSBUS
MicroVAX
Q-bus
ULTRIX
ULTRIX Mail Connection
ULTRIX Worksystem Software
UNIBUS
VAX
VAXstation
VMS
VMS/ULTRIX Connection
VT
XUI

Ethernet is a registered trademark of Xerox Corporation.

Network File System and NFS are trademarks of Sun Microsystems, Inc.

POSIX is a registered trademark of the Institute of Electrical and Electronics Engineers.

System V is a registered trademark of AT&T.

Tektronix is a trademark of Tektronix, Inc.

Teletype is a registered trademark of AT&T in the USA and other countries.

UNIX is a registered trademark of AT&T in the USA and other countries.

About Reference Pages

The *ULTRIX Reference Pages* describe commands, system calls, routines, file formats, and special files for RISC and VAX platforms.

Sections

The reference pages are divided into eight sections according to topic. Within each section, the reference pages are organized alphabetically by title, except Section 3, which is divided into subsections. Each section and most subsections have an introductory reference page called `intro` that describes the organization and anything unique to that section.

Some reference pages carry a one- to three-letter suffix after the section number, for example, `scan`(1mh). The suffix indicates that there is a ''family'' of reference pages for that utility or feature. The Section 3 subsections all use suffixes and other sections may also have suffixes.

Following are the sections that make up the *ULTRIX Reference Pages*.

Section 1: Commands

This section describes commands that are available to all ULTRIX users. Section 1 is split between two binders. The first binder contains reference pages for titles that fall between A and L. The second binder contains reference pages for titles that fall between M and Z.

Section 2: System Calls

This section defines system calls (entries into the ULTRIX kernel) that are used by all programmers. The introduction to Section 2, `intro`(2), lists error numbers with brief descriptions of their meanings. The introduction also defines many of the terms used in this section.

Section 3: Routines

This section describes the routines available in ULTRIX libraries. Routines are sometimes referred to as subroutines or functions.

Section 4: Special Files

This section describes special files, related device driver functions, databases, and network support.

Section 5: File Formats

This section describes the format of system files and how the files are used. The files described include assembler and link editor output, system accounting, and file system formats.

Section 6: Games

The reference pages in this section describe the games that are available in the unsupported software subset. The reference pages for games are in the document *Reference Pages for Unsupported Software.*

Section 7: Macro Packages and Conventions

This section contains miscellaneous information, including ASCII character codes, mail addressing formats, text formatting macros, and a description of the root file system.

Section 8: Maintenance

This section describes commands for system operation and maintenance.

Platform Labels

The *ULTRIX Reference Pages* contain entries for both RISC and VAX platforms. Pages that have no platform label beside the title apply to both platforms. Reference pages that apply only to RISC platforms have a ''RISC'' label beside the title and the VAX-only reference pages that apply only to VAX platforms are likewise labeled with ''VAX.'' If each platform has the same command, system call, routine, file format, or special file, but functions differently on the different platforms, both reference pages are included, with the RISC page first.

Reference Page Format

Each reference page follows the same general format. Common to all reference pages is a title consisting of the name of a command or a descriptive title, followed by a section number; for example, `date`(1). This title is used throughout the documentation set.

The headings in each reference page provide specific information. The standard headings are:

Name	Provides the name of the entry and gives a short description.
Syntax	Describes the command syntax or the routine definition. Section 5 reference pages do not use the Syntax heading.
Description	Provides a detailed description of the entry's features, usage, and syntax variations.
Options	Describes the command-line options.
Restrictions	Describes limitations or restrictions on the use of a command or routine.
Examples	Provides examples of how a command or routine is used.

Return Values	Describes the values returned by a system call or routine. Used in Sections 2 and 3 only.
Diagnostics	Describes diagnostic and error messages that can appear.
Files	Lists related files that are either a part of the command or used during execution.
Environment	Describes the operation of the system call or routine when compiled in the POSIX and SYSTEM V environments. If the environment has no effect on the operation, this heading is not used. Used in Sections 2 and 3 only.
See Also	Lists related reference pages and documents in the ULTRIX documentation set.

Conventions

The following documentation conventions are used in the reference pages.

%	The default user prompt is your system name followed by a right angle bracket. In this manual, a percent sign (%) is used to represent this prompt.
#	A number sign is the default superuser prompt.
`user input`	This bold typeface is used in interactive examples to indicate typed user input.
`system output`	This typeface is used in text to indicate the exact name of a command, routine, partition, pathname, directory, or file. This typeface is also used in interactive examples to indicate system output and in code examples and other screen displays.
UPPERCASE lowercase	The ULTRIX system differentiates between lowercase and uppercase characters. Literal strings that appear in text, examples, syntax descriptions, and function definitions must be typed exactly as shown.
rlogin	This typeface is used for command names in the Syntax portion of the reference page to indicate that the command is entered exactly as shown. Options for commands are shown in bold wherever they appear.
filename	In examples, syntax descriptions, and routine definitions, italics are used to indicate variable values. In text, italics are used to give references to other documents.
[ ]	In syntax descriptions and routine definitions, brackets indicate items that are optional.
{ | }	In syntax descriptions and routine definitions, braces enclose lists from which one item must be chosen. Vertical bars are used to separate items.

. . .	In syntax descriptions and routine definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
. . .	A vertical ellipsis indicates that a portion of an example that would normally be present is not shown.
`cat(1)`	Cross-references to the *ULTRIX Reference Pages* include the appropriate section number in parentheses. For example, a reference to `cat(1)` indicates that you can find the material on the `cat` command in Section 1 of the reference pages.

Online Reference Pages

The ULTRIX reference pages are available online if installed by your system administrator. The `man` command is used to display the reference pages as follows:

To display the `ls(1)` reference page:

```
% man ls
```

To display the `passwd(1)` reference page:

```
% man passwd
```

To display the `passwd(5)` reference page:

```
% man 5 passwd
```

To display the Name lines of all reference pages that contain the word ''passwd'':

```
% man -k passwd
```

To display the introductory reference page for the family of 3xti reference pages:

```
% man 3xti intro
```

Users on ULTRIX workstations can display the reference pages using the unsupported `xman` utility if installed. See the `xman(1X)` reference page for details.

Reference Pages for Unsupported Software

The reference pages for the optionally installed, unsupported ULTRIX software are in the document *Reference Pages for Unsupported Software*.

Name

intro – introduction to commands

Description

This section describes publicly accessible commands in alphabetic order. Certain distinctions of purpose are made in the headings:

(1)	Commands of general utility.
(1c)	Commands for communication with other systems.
(1g)	Commands used primarily for graphics and computer-aided design.
(1int)	Commands used for internationalization. For more information see `internat`(1int).
(1mh)	Commands specific to the Message Handler.
(1ncs)	Commands used for NCS (Network Computing System).
(1sh5)	Commands interpreted by the sh5 (System V Release 2) shell.
(1yp)	Commands specific to the Yellow Pages (YP) service.

Note

Commands related to system maintenance used to appear in section 1 reference pages and were distinguished by (1m) at the top of the page. These reference pages now appear in section 8.

Diagnostics

Upon termination each command returns two bytes of status, one supplied by the system giving the cause for termination and, in the case of normal termination, one supplied by the program. For more information, see `wait`(1) and `exit`(2). The former byte is 0 for normal termination; the latter is customarily 0 for successful execution. A nonzero status indicates a problem such as erroneous parameters, or bad or inaccessible data. It is called variously exit code, exit status, or return code, and is described only where special conventions are involved.

Name

2780e – spooler for the IBM 2780 emulator

Syntax

2780e [**–m**] [**–a**] [**–q**] [**–b**] [**–t**] [**–S***file*] [**–#***num*] *file*... [**–o** *file*...]

Description

The `2780e` command puts the files named as arguments, along with a single control file that guides each file's execution, into `usr/spool/rje` and calls the `2780d` program. This program sends the files to the IBM system.

Options

The following options may be needed to format data transmitted to an IBM system.

–# Waits for *num* files to be received as output from job and gives default file names in the form *Ruseridpid.*

–a Send file as a priority job. Used only by the superuser. This file will be placed ahead of the next regular file or at the end of other priority jobs.

–b Transmits the file to an IBM system that accepts multiple record transmission.

–m Notifies user by mail that file was sent and output was received.

–o Name output files with specified file names. This option must be at the end of the command line. Anything listed after this option is interpreted as an output file name.

–q Prepares the file for transmission and places it in `/usr/spool/rje` but does not call 2780d to transmit.

–S Sends contents of file to the IBM as a sign-on card. If this option is not specified, a default sign-on card in the spool area is used.

–t Sends data in transparent mode. This option is used for files which contain special control or protocol characters.

Files

/etc/2780d	Program that transmits files.
/usr/spool/rje	Spool directory
/usr/spool/rje/rjetemp.out	Temporary file for incoming files

See Also

2780d(8), 3780e(1)

Name

3780e – spooler for the IBM 3780 emulator

Syntax

3780e [–C] [–m] [–a] [–q] [–t[b]] [–S*file*] [–#*num*] *file*... [–o *file*]

Description

The `3780e` command puts the files named as arguments, along with a single control file that guides each file's execution, into `usr/spool/rje` and calls the `2780d` program. This program sends the files to the IBM system.

Options

The following options may be needed to format the data transmitted to an IBM system.

–# Waits for *num* files to be received as output and gives default file names in the form *Ruseridpid.*

–a Send file as a priority job. Used only by the superuser. This file will be placed ahead of the next regular file or at the end of other priority jobs.

–C Prevents the compression of spaces when files are sent.

–m Sends mail when file is sent and when output from submitted file is received successfully.

–o Names output file with specified file names. This option must be at the end of the command line. Anything listed after this option is interpreted as an output file name.

–q Prepares file for transmission and places it in `/usr/spool/rje` but does not call `2780d` to transmit it.

–S Send contents of the file to the IBM as a sign-on card. If this option is not specified, then a default sign-on card in the spool area will be used.

–t Sends data in transparent mode. This option is used for files which contain special control or protocol characters. Use this option if the IBM system does not accept multiple 80 column card records in transparent mode.

–tb Transmits the file to IBM that accepts multiple 80 column card records in transparent mode.

Files

/etc/2780d	Program that transmits files
/usr/spool/rje	Spool directory
/usr/spool/rje/rjetemp.out	Temporary file for incoming files

See Also

2780d(8), 2780e(1)

Name

adb – interactive C program debugger

Syntax

adb [–w] [–k] [–I*dir*] [*objfil* [*corfil*]]

Description

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

The *objfil* is normally an executable program file, preferably containing a symbol table. If it does not contain a symbol table then the symbolic features of adb cannot be used. However, the file can still be examined. The default for *objfil* is a.out. The *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 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.

The **–k** option makes adb do UNIX kernel memory mapping; it should be used when *core* is a UNIX crash dump or /dev/mem.

The **–I** option specifies a directory where files to be read with $< or $<< (see the EXPRESSIONS section) are sought. The default directory is /usr/lib/adb.

The adb command ignores QUIT; INTERRUPT causes return to the next adb command.

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 is executed. The default *count* is 1. *Address* and *count* are expressions.

The interpretation of an address depends on its context. 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/mem to interactive examine and/or modify memory the maps are set to map the kernel virtual addresses which start at 0x80000000. 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 prefixes 0o and 0O ("zero oh") force interpretation in octal radix; the prefixes 0t and 0T force interpretation in decimal radix; the prefixes 0x and 0X force interpretation in hexadecimal radix. Thus 0o20 = 0t16 = 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 whose most significant digit 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.

< *name* The value of *name*, which is either a variable name or a register name. The `adb` debugger 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 lower case letters, underscores or digits, not starting with a digit. The backslash character \ may be used to escape other characters. The value of the *symbol* is taken from the symbol table in *objfil*. An initial underscore (_) will be prepended to *symbol* if needed.

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

routine.name The address of the variable *name* in the specified C routine. Both *routine* and *name* are *symbols*. If *name* is omitted the value is the address of the most recently activated C stack frame corresponding to *routine*. This form is currently broken on the VAX; local variables can be examined only with `dbx`(1).

(*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 question mark (?) and slash (/) may be followed by an asterisk (*); see the ADDRESSES section 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.

/*f* Locations starting at *address* in *corfil* are printed according to the format *f* and *dot* is incremented as for question mark (?).

=*f* The value of *address* itself is printed in the styles indicated by the format *f*. (For **i** format, the question mark (?) 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:

o2 Print 2 bytes in octal. All octal numbers output by `adb` are preceded by 0.
O4 Print 4 bytes in octal.
q2 Print in signed octal.
Q4 Print long signed octal.
d2 Print in decimal.
D4 Print long decimal.
x2 Print 2 bytes in hexadecimal.
X4 Print 4 bytes in hexadecimal.
u2 Print as an unsigned decimal number.
U4 Print long unsigned decimal.
f4 Print the 32 bit value as a floating point number.
F8 Print double floating point.
b1 Print the addressed byte in octal.
c1 Print the addressed character.
C1 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 the format **C1** above). *n* is the length of the string including its zero terminator.
Y4 Print 4 bytes in date format. For further information, see `ctime`(3).
i*n* Print as machine instructions. *n* is the number of bytes occupied by the instruction. This style of printing causes variables 1 and 2 to be set to the offset parts of the source and destination respectively.
a0 Print the value of *dot* in symbolic form. Symbols are checked to ensure that they have an appropriate type as indicated below:

/ local or global data symbol
? local or global text symbol
= local or global absolute symbol

p4 Print the addressed value in symbolic form using the same rules for symbol lookup as **a0**.

t0 When preceded by an integer tabs to the next appropriate tab stop. For example, **8t** moves to the next 8-space tab stop.

r0 Print a space.

n0 Print a new line.

"..."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 the '?' or '/' is followed by '*' then the second segment (*b2*, *e2*, *f2*) of the mapping is changed. 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.

! A shell (/bin/sh) is called to read the rest of the line following '!'.

$*modifier* Miscellaneous commands. The following *modifiers* are available:

<*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 registers such 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.

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 the contents of the frame–pointer register. If **C** is used then the names and (32 bit) values of all automatic and static variables are printed for each active function. (broken on the VAX). 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 All integers input are regarded as octal.

q Exit from `adb`.

v Print all non zero variables in octal.

m Print the address map.

p (*Kernel debugging*) Change the current kernel memory mapping to map the designated **user structure** to the address given by the symbol *_u*. The *address* argument is the address of the user's user page table entries.

x (*Kernel debugging*) The *address* argument is the CPU number. Change the current kernel memory mapping to that of the specified CPU. If no address is provided, the status of each of the CPUs in the system is displayed. This option is ONLY valid with the **–k** option.

:modifier Manage a subprocess. The following modifiers are available:

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.

c*s* The subprocess is continued with signal *s*, see `sigvec`(2). 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

The `adb` command provides a number of variables. Named variables are set initially by `adb` but are not used subsequently. The following numbered variables are reserved for communication:

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 or 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 two triples (*b1, e1, f1*) and (*b2, e2, f2*) and the *file address* corresponding to a written *address* is calculated as follows:

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

$b2 \leq address < e2 \Rightarrow file\ address = address + f2 - b2$,

otherwise, the requested *address* is not legal. In some cases (for example, for programs with separated I and D space) the two segments for a file may overlap. If a **?** or **/** is followed by an ***** then only the second triple is 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. This way the whole file can be examined with no address translation.

Restrictions

Because no shell is invoked to interpret the arguments of the `:r` command, the customary wildcard and variable expansions cannot occur.

Diagnostics

When there is no command or format given to `adb`, the string 'adb' appears. `adb` displays comments about inaccessible files, syntax errors, abnormal termination of commands, etc. Exit status is 0, unless last command failed or returned nonzero status.

Files

`a.out`
`core`

See Also

cc(1), dbx(1), ptrace(2), a.out(5), core(5)

Name

addbib – create or extend bibliographic database

Syntax

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

Description

When this program starts up, answering "y" to the initial "Instructions?" prompt yields directions; typing "n" or RETURN skips them. The addbib command 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).

Options

–a Suppresses prompting for an abstract. Asking for an abstract is the default. Abstracts are ended with a CTRL/D.

–p Causes use of a new prompting skeleton, defined in *promptfile*. This file should contain prompt strings, a tab, and the key-letters to be written to the *database*.

The most common key-letters and their meanings are given below. The addbib insulates you from these key-letters, since it gives you prompts in English. However, if you edit the bibliography file later, you need 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*
%M Bell Labs Memorandum (undefined)
%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,Z	ignored by *refer*

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

```
%A      Bill Tuthill
%T      Refer - A Bibliography System
%I      Computing Services
%C      Berkeley
%D      1982
%O      UNX 4.3.5.
```

Files

promptfile   Optional file to define prompting

See Also

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

Name

admin – SCCS file administrator

Syntax

admin [–n] [–i[*name*] **] [–r***rel* **] [–t**[*name*] **] [–f***flag* [*flag-val*] **] [–d***flag* [*flag-val*] **]**
[–a*login*] **[–e***login*] **[–m**[*list*] **] [–y**[*comment*] **] [–h] [–z]** *files*

Description

The `admin` command is used to create new SCCS files and to 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 path name does not begin with **s.**) and unreadable files are 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 ignored.

Options

Each keyletter argument is explained as though only one named file is to be processed, because the effects of the arguments apply independently to each named file. The list of arguments is as follows:

–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 the **–r** keyletter for the 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.

Only one SCCS file can be created by an `admin` command in which the **i** keyletter is supplied. Using a single `admin` command 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* The release 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 level of the initial delta is always 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 are also used), the descriptive text file name must also be supplied.

In the case of existing SCCS files: (1) a **–t** keyletter without a file name causes removal of descriptive text (if any) currently in the SCCS file; and (2) a **–t** keyletter with a file name causes text (if any) in the named file to replace the descriptive text (if any) currently in the SCCS file.

–f*flag* This keyletter specifies a *flag*, and, possibly, a value for the *flag*, to be placed in the SCCS file. Several **f** keyletters can be supplied on a single `admin` command line. The allowable *flag*s and their values are:

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

c*ceil* The highest release (''ceiling''), a positive number no higher than 9999, which may be retrieved by a `get`(1) command for editing. The default value for an unspecified **c** flag is 9999.

f*floor* The lowest release (''floor''), a positive 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 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 are found in the text retrieved or stored in the SCCS file. For further information, see `get`(1).

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 can 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. For example, 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 can later be created from them. The absence of this flag causes skipped releases to be nonexistent in the SCCS file, preventing branch deltas from being 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*ule name of the SCCS file substituted for all occurrences of the %M% keyword in SCCS file tex 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. The optional value specifies the name of an *MR* number validity checking program. For further information, 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 deletion of the specified *flag* from an SCCS file. The **–d** keyletter can be specified only when processing existing SCCS files. Several **–d** keyletters can 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 numerical ULTRIX System group ID to be added to the list of users which can 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 can be used on a single `admin` command line. As many *login*s or numerical group IDs as desired can be on the list simultaneously. If the list of users is empty, then anyone can add deltas.

–e*login* A *login* name or numerical 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 can 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*
The **–y** keyletter is valid only if the **–i** or **–n** keyletters are specified.

–m[*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). Diagnostics occur if the **v** flag is not set or *MR* validation fails.

–h Causes `admin` to check the structure of the SCCS file and to compare a newly computed check-sum (the sum of all the characters in the SCCS file except those in the first line) with the check-sum that is stored in the first line of the SCCS file. Appropriate error diagnostics are produced. For further information, see `sccsfile`(5).

This keyletter inhibits writing on the file, so that it nullifies the effect of any other keyletters supplied. It is, therefore, only meaningful when processing existing files.

–z The SCCS file check-sum 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.

Diagnostics

Use sccshelp(1) for explanations.

Restrictions

When creating a new SCCS file with the **-n** or **-i** options, the g-file path name cannot be of the form **s.***file-name.*

Files

The last component of all SCCS file names are of the form **s.***file-name*. New SCCS files are given mode 444. For further information, see `chmod`(1). Write permission in the pertinent directory is required to create a file. All writing done by `admin` is to a temporary x-file, called **x.***file-name*, 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. For further information, see `get`(1). 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.

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)`.

NOTE

> Care must be taken that correct commands are used when patching an SCCS file, otherwise further corruption of the file can occur.

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.

The `admin` command 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. For further information, see `get(1)`.

See Also

delta(1), ed(1), get(1), help(1), prs(1), sccs(1), what(1), sccsfile(5)
Guide to the Source Code Control System

Name

ali – list mail aliases

Syntax

ali [–alias *aliasfile*] **[–list] [–nolist] [–normalize] [–nonormalize] [–user** <*useradr*>] **[–nouser]** *aliases ...* **[–help]**

Description

The command `ali` searches the specified mail alias files for each of the given aliases. It creates a list of addresses for those aliases, and displays that list on the screen.

Options

You specify the alias files using the `-alias` *aliasfile* option. You can specify more than one alias file, but each *aliasfile* must be preceded by its own `-alias` flag. You must specify an *aliasfile*; either at the command line itself, or in your `.mh_profile`.

If you specify the `-list` option, each address appears on a separate line; otherwise, the addresses are separated by commas and printed on as few lines as possible.

You can make `ali` display all the aliases that contain a specific name by using the `-user` *useraddress* option The `-user` option directs `ali` to perform its processing in an inverted fashion: instead of listing the addresses that each given alias expands to, `ali` lists the aliases that expand to each given address. You must specify the complete username that you have used in your `mh-alias` file. The following example shows how this option can be used.

```
% ali -user Parker@Venus
Parker@Venus:  Group, Reviewers, Badminton
```

If the `-normalize` option is given, `ali` tries to track down the official hostname of the address.

Each alias is processed as described in `mh-alias`(5mh).

The defaults for `ali` are:

```
-alias /usr/new/lib/mh/MailAliases
-nolist
-nonormalize
-nouser
```

Files

$HOME/.mh_profile	The user profile
/etc/passwd	List of users
/etc/group	List of groups

See Also

mh-alias(5mh)

Name

anno – annotate messages

Syntax

anno [+*folder*] [*msgs*] **[––component** *field***] [–inplace] [–noinplace] [–text body] [–help]**

Description

The command `anno` annotates the specified messages in the named folder using the field and body. You can use `anno` with `dist`, `forw`, and `repl`, to keep track of the distribution and forwarding of, and replies to your messages. By using `anno`, you can perform arbitrary annotations of your own. Each message selected is annotated with the following lines:

```
field: date
field: body
```

Options

The `-inplace` switch causes annotation to be done in place in order to preserve links to the annotated message.

The `-component` *field* specified should be a valid RFC 822-style message field name, which means that it should consist of alphanumerics (or dashes) only. The body specified is arbitrary text.

If a `-component` *field* is not specified when `anno` is invoked, `anno` prompts you for the field-name for the annotation.

If a folder is given, it becomes the current folder. The first message annotated becomes the current message.

The defaults for `anno` are:
+folder defaults to the current folder
msgs defaults to the current message
`-noinplace.`

Files

`$HOME/.mh_profile`	The user profile

Profile Components

Path:	To determine your MH directory
Current–Folder:	To find the default current folder

See Also

dist(1mh), forw(1mh), repl(1mh)

apply(1)

Name

apply – apply a command to a set of arguments

Syntax

apply [–**a***c*] [–*n*] *command args...*

Description

The apply program 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 percent sign (%) character can be changed by the **–a** option.

Examples

The following command line is similar to ls(1):

```
apply echo *
```

The next example compares the specified a files to the specified b files:

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

The following example run the who command 5 times and links all files in the current directory to the directory /usr/joe:

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

Restrictions

Shell metacharacters in *command* may have unexpected results; it is best to enclose complicated commands in single quotes (' ').

You cannot pass a literal, '%2', if the percent sign (%) is the argument expansion character.

See Also

sh(1)

Name

apropos – locate commands by keyword lookup

Syntax

apropos *keyword...*

Description

The `apropos` command shows which manual sections contain instances of any of the given keywords in their title. Each word is considered separately and the case of letters is ignored. Words that are part of other words are listed. Thus, looking for the word compile hits all instances of 'compiler' also.

If the line starts 'name(section) ...' you can do 'man section name' to get the documentation for it. The following command line lists all commands that have to do with formatting:

```
apropos format
```

To then access the reference page for the *printf* subroutine that you see listed, type:

```
man 3s printf
```

The `apropos` command is actually just the **–k** option to the `man` command.

Files

/usr/lib/whatis data base

See Also

man(1), whatis(1), catman(8)

Name

ar – archive and library maintainer

Syntax

ar option [*posname*] file1 ... fileN

Description

The archiver **ar** maintains groups of files as a single archive file. This utility is generally used to create and update library files that the link editor uses; however, you can use the archiver for other similar purposes.

NOTE

This version uses a portable ASCII-format archive that you can use on various machines that run UNIX. If you have an archive that uses an older format, see `arcv`(8).

Options

This section describes the options and suboptions that you can use with the **ar** utility. Suboptions must be specified with options. Following is a list and description of the options:

d Deletes the specified files from the archive file.

r Replaces the specified files in the archive file. If you use the suboption **u** with **r**, the archiver only replaces those files that have last-modified dates later than the archive files. If you use a positioning character (from the set **abi**) you must specify the *posname* argument to tell the archiver to put the new files after (**a**) or before (**b** or **i**). Otherwise, the archiver puts new files at the end of the archive.

q Appends the specified files to the end of the archive file. The archiver does not accept suboption positioning characters with the **q** option. It also does not check whether the files you want to add already exist in the archive. Use the **q** option only to avoid quadratic behavior when you create a large archive piece by piece.

t Prints a table of contents for the files in the archive file. If you do not specify any filenames, the archiver builds a table of contents for all files. If you specify filenames, the archiver builds a table of contents only for those files.

p Prints the specified files from the archive.

m Moves the specified files to the end of the archive. If you specify a positioning character, you must also specify the *posname* (as in option **r**) to tell the archiver where to move the files.

x Extracts the specified files from the archive. If you do not specify any filenames, the archiver extracts all files. When it extracts files, the archiver does not change any file. Normally, the last-modified date for each extracted file shows the date when someone extracted it; however, when you use **o**, the archiver resets the last-modified date to the date recorded in the archive.

s Makes a symbol definition (symdef file) as the first file of an archive. This file contains a hash table of *ranlib* structures and a corresponding string table. The symdef file's name is based on the byte ordering of the hash table and the byte ordering of the file's target machine. Files must be consistent in their target byte ordering before the archiver can create a symdef file. If you change the archive contents, the symdef file becomes obsolete because the archive file's name changes. If you specify s, the archiver creates the symdef file as its last action before finishing execution. You must specify at least one other archive option (m, p, q, r, or t) when you use the **s** option. For UMIPS-V, archives include member objects based on the definition of a common object only. For UMIPS-BSD, they define the common object, but do not include the object.

v Gives a file-by-file description as the archiver makes a new archive file from an old archive and its constituent files. When you use this option with **t,** the archiver lists all information about the files in the archive. When you use this option with **p**, the archiver precedes each file with a name.

c Suppresses the normal message that the archiver prints when it creates the specified archive file. Normally, the archiver creates the specified archiver file when it needs to.

l Places temporary files in the local directory. If the l option is not used then the value of the environment symbol, TMPDIR, is used as the directory for temporary files. If TMPDIR is not defined or if the directory it references is not writable then /tmp is used.

The suboptions do these things:

a Specifies that the file goes after the existing file (*posname*). Use this suboption with the **m** or **r** options.

b Specifies that the file goes before the existing file (*posname*). Use this suboption with the **m** or **r** options.

i Specifies that the file goes before the existing file (*posname*). Use this suboption with the **m** or **r** options.

o Forces a newly created file to have the last-modified date that it had before it was extracted from the archive. Use this suboption with the **x** option.

u Prevents the archiver from replacing an existing file unless the replacement is newer than the existing file. This option uses the UNIX system last modified date for this comparison. Use this suboption with the **r** option.

Restrictions

If you specify the same file twice in an argument list, it can appear twice in the archive file.

The **o** option does not change the last-modified date of a file unless you own the extracted file or you are the superuser.

Files

/tmp/v* temporaries

See Also

lorder(1), ld(1), odump(1), ranlib(1), ranhash(3x), ar(5), arcv(8)

Name

ar – archive and library maintainer

Syntax

ar *–key* [*posname*] *afile name...*

Description

The ar command maintains groups of files combined into a single archive file. The ar command is used to create and update library files as they are used by the loader.

This version of *ar* uses a ASCII-format archive, which can be used by the various machines running UNIX. Programs for dealing with older formats are also available. For further information, see arcv(8).

The *key* is one character from the following set: **d**, **r**, **q**, **t**, **p**, **m**, **x**. The *key* character can be concatenated with one or more of the following optional characters: **v**, **u**, **a**, **i**, **b**, **c**, **l**, **o**. The *afile* is the archive file. The *names* are constituent files in the archive file.

Options

The OPTIONS section is divided into two sections: the first section lists the *key* characters and their meanings, and the second section lists the optional characters and their meanings.

For backward compatibility, the *keys* work without the dash (–). The definitions of the *key* characters are as follows:

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 **r**, it will specify where the files are to be moved.

p Prints the named files in the archive.

q Appends the named files to the end of the archive file. Optional positioning characters are invalid. The command does not check whether the added files are already in the archive. The **q** option is used primarily to avoid quadratic behavior when you are creating a large archive piece-by-piece.

r Replaces the named files in the archive file. If the optional character **u** is used with **r**, then only those files with last-modified dates later than the archive files are replaced. If an optional positioning character from the set **a**, **b**, or **i** is used, then the *posname* argument must be present and it specifies that new files should be placed after (**a**) or before (**b** or **i**) *posname*. Otherwise new files are placed at the end.

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

x Extracts the named files. If no names are given, all files in the archive are extracted. However, **x** does not alter the archive file. Normally the last-

modified date of each extracted file is the date when it is extracted. However, if **o** is used, the last-modified date is reset to the date recorded in the archive.

The following optional characters can be used in conjunction with the *key* characters:

a Tells the `ar` command that new files should be placed after *posname*.

b Tells the `ar` command that new files should be placed before *posname*.

c Suppresses the message that is normally produced when *afile* is created.

i Tells the `ar` command that new files should be placed before *posname*.

l Places files in the local directory. If the l option is not used then the value of the environment symbol, TMPDIR, is used as the directory for temporary files. If TMPDIR is not defined or if the directory it references is not writable then /tmp is used.

o Resets the last-modified date to the date recorded in the archive. Normally the last-modified date is the date when the file was extracted.

u Replaces only those files with last-modified dates later than the archive files. See the **r** *key* character.

v 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 the **t** option, it gives a long listing of all information about the files. When used with the **p** option, it precedes each file with a name.

Restrictions

The `ar` command truncates the filenames to 15 characters.

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

The last-modified date of a file is not altered by the **o** option if the user is not the owner of the extracted file or a super-user.

Files

/tmp/v* temporaries

See Also

ld(1), lorder(1), ranlib(1), ar(5), arcv(8)

Name

as – RISC assembler

Syntax

as [option] ... *file*

Description

The assembler, **as** , produces RISC object code in RISC extended **coff** format (the default) and binary assembly language. The **as** assembler does not run the loader. It accepts the argument *file* which is a symbolic assembly language source program. When assembled, it produces an object file.

The assembler, **as,** always defines the C preprocessor macros **mips, host_mips, unix** and **LANGUAGE_ASSEMBLY** to the C macro preprocessor. It also defines **SYSTYPE_BSD** by default, but this changes if the **–systype** *name* option is specified (see the OPTIONS section).

Options

The following options are available with **as.** In addition, these options can be used with cc(1).

–g0 Do not produce symbol table information for symbolic debugging. This is the default.

–g1 Produce additional symbol table information for accurate but limited symbolic debugging of partially optimized code.

–g or **–g2** Produce additional symbol table information for full symbolic debugging and do not perform optimizations that limit full symbolic debugging.

–g3 Produce additional symbol table information for full symbolic debugging for fully optimized code. This option makes the debugger inaccurate.

–w Suppress warning messages.

–P Run only the C macro preprocessor and place the result in a file. If the source file has a suffix, change the suffix to .i. If the source file does not have a suffix, an .i is added to the source file name. The .i file does not contain number lines (#). This sets the **–cpp** option.

–E Run only the C macro preprocessor on the file and send the result to the standard output. This sets the **–cpp** option.

–o *output* Name the final output file *output*. If this option is used, the a.out file is not affected.

–D*name=def*
–D*name* Define the *name* to the C macro preprocessor, as if by #define. If definition is not given, the name is defined as 1.

–U*name* Remove any initial definition of *name*.

–I*dir* Search for #include files whose names do not begin with slash (/) in the directory of the *file* argument, then in directories specified in **–I** options, and finally in the standard directory (**/usr/include**).

–I Do not search for #include files in the standard directory (**/usr/include**).

–G *num* Specify the maximum size, in bytes, of a data item that is to be accessed from the global pointer. The *num* argument is interpreted as a decimal number. If *num* is zero, data is not accessed from the global pointer. The default value for *num* is 8 bytes.

–v Print the passes as they execute with their arguments, input files, and output files. Also prints resource usage in the C-shell *time* format.

–V Print the version of the driver and the versions of all passes. This is performed with the `what(1)` command.

–cpp Run the C macro preprocessor on assembly source files before compiling. This is the default for `as(1)`.

–nocpp Do not run the C macro preprocessor on assembly source files before compiling.

Either object file target byte ordering can be produced by **as.** The default target byte ordering matches the machine where the assembler is running. The options **–EB** and **–EL** specify the target byte ordering (big-endian and little-endian, respectively). The assembler also defines a C preprocessor macro for the target byte ordering. These C preprocessor macros are **MIPSEB** and **MIPSEL** for big-endian and little-endian byte ordering respectively.

–EB Produce object files targeted for big-endian byte ordering. The C preprocessor macro **MIPSEB** is defined by the assembler.

–EL Produce object files targeted for little-endian byte ordering. The C preprocessor macro **MIPSEL** is defined by the assembler.

The following option can only be used with the **as** command:

–m Apply the M4 preprocessor to the source file before assembling it.

The following option is primarily used to provide UNIX compilation environments other than the native compilation environment.

–systype *name* Use the named compilation environment *name*. See `compilation(7)` for the compilation environments that are supported and their *name*s. This has the effect of changing the standard directory for #include files. The new items are located in their usual paths but with */name* prepended to their paths. Also a preprocessor macro of the form **SYSTYPE_***NAME* (with *name* capitalized) is defined in place of the default **SYSTYPE_BSD.**

The options described below primarily aid compiler development and are not generally used:

–H*c* Halt compiling after the pass specified by the character *c*, producing an intermediate file for the next pass. The *c* can be **[a].** It selects the assembler pass in the same way as the **–t** option. If this option is used, the symbol table file produced and used by the

passes is the last component of the source file with the suffix changed to .T, or a .T is added if the source file has no suffix. This file is not removed.

–K
: Build and use intermediate file names with the last component of the source file's name replacing its suffix with the conventional suffix for the type of file (for example, .G file for binary assembly language). If the source file has no suffix the conventional suffix is added to the source file name. These intermediate files are never removed even when a pass encounters a fatal error.

–W*c[c...],arg1[,arg2...]*
: Pass the argument[s] *argi* to the compiler pass[es] *c[c..]*. The *c's* are one of [**pab**]. The c's selects the compiler pass in the same way as the **–t** option.

The options **–t[hpab]**, **–h***path,* and **–B***string* select a name to use for a particular pass. These arguments are processed from left to right so their order is significant. When the **–B** option is encountered, the selection of names takes place using the last **–h** and **–t** options. Therefore, the **–B** option is always required when using **–h** or **–t**. Sets of these options can be used to select any combination of names.

–t[hpab]
: Select the names. The names selected are those designated by the characters following the **–t** option according to the following table:

Name	**Character**
include	h (see note below)
cpp	p
as0	a
as1	b

If the character h is in the **–t** argument then a directory is added to the list of directories to be used in searching for #include files. This directory name has the form COMP_TARGET_ROOT/usr/include*string* . This directory is to contain the include files for the *string* release of the compiler. The standard directory is still searched.

–h*path*
: Use *path* rather than the directory where the name is normally found.

–B*string*
: Append *string* to all names specified by the **–t** option. If **–t** option has not been processed before the **–B**, the **–t** option is assumed to be "hpab". This list designates all names.

Invoking the assembler with a name of the form **as***string* has the same effect as using a **–B***string* option on the command line.

If the environment variable COMP_HOST_ROOT is set, the value is used as the root directory for all pass names rather than the default slash (/). If the environment variable COMP_TARGET_ROOT is set, the value is used as the root directory for the includes rather than the default slash (/).

If the environment variable ROOTDIR is set, the value is used as the root directory for all names rather than the default **/usr/**. This also affects the standard directory for #include files, /usr/include .

If the environment variable TMPDIR is set, the value is used as the directory to place any temporary files rather than the default **/tmp/**.

Other arguments are ignored.

Files

file.o	object file
a.out	assembler output
/tmp/ctm?	temporary
/usr/lib/cpp	C macro preprocessor
/usr/lib/as0	symbolic to binary assembly language translator
/usr/lib/as1	binary assembly language assembler and reorganizer
/usr/include	standard directory for #include files

See Also

cc(1), as(1), what(1)

Name

as – assembler

Syntax

as [–d124] [–L] [–W] [–V] [–J] [–R] [–t *directory*] [**–o** *objfile*] [*name...*]

Description

The `as` assembler assembles the named files, or the standard input if no file name is specified.

Options

–d Specifies number of bytes for offsets that involve forward or external references and have sizes unspecified in assembly language. The default is **–d4.**

–J Uses long branches to resolve jumps when byte-displacement branches are insufficient. This must be used when a compiler-generated assembly contains branches of more than 32k bytes.

–L Saves defined labels beginning with L, which are normally discarded. The compilers generate such temporary labels.

–o Specifies the name of the output file. If this option is omitted, `a.out` is assumed.

–R Make initialized data segments read only, by concatenating them to the text segments. This prevents the need to run editor scripts on assembly code to make initialized data read only and shared.

–t Specifies a directory other than the default `/tmp` to receive the temporary file.

–V Uses virtual memory rather than a temporary file for immediate storage.

–W Do not complain about errors.

All undefined symbols in the assembly are treated as global.

The output of the assembly is left on the file *objfile;* if that is omitted, **a.out** is used.

Files

`/tmp/as*`	default temporary files
`a.out`	default resultant object file

See Also

adb(1), dbx(1), ld(1), nm(1), a.out(5)
ULTRIX Supplementary Documents, Vol. III: System Manager

Name

at, batch – execute commands at a later time

Syntax

at *time* [*day*] [*file*]
at –r *job...*
at –l [*job...*]

batch [*file*]

Description

The at and batch commands use a copy of the named *file* (standard input default) as input to sh(1) or csh(1) at a later time. A cd command to the current directory is inserted at the beginning, followed by assignments to all environment variables. When the script is run, it uses the user and group ID of the creator of the copy file.

The at command allows the user to specify when the commands should be executed, while jobs queued with batch execute when the load level of the system permits.

The environment variables, current directory, umask, and ulimit are retained when the commands are executed. However, open files, traps, and priority are lost.

Users are permitted to use the at and batch commands 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 and batch. If neither file exists, only the superuser is allowed to submit a job. If only the at.deny file exists and is empty, global usage is permitted. The allow/deny files consist of one user name per line.

The *time* is 1 to 4 digits. It can, but does not have to be, followed by A, P, N or M which stand for AM, PM, noon or midnight, respectively. The A, P, N, and M suffixes are case-insensitive. One and two digit numbers are interpreted as hours, three and four digits to be hours and minutes. If three digits are specified, the first digit is interpreted to be an hour in the range 0-9, and the second and third digits as minutes. If no letters follow the digits, a 24 hour clock time is presumed.

In addition to 1-4 digits, and suffixes A, P, M, N, you can also specify:

```
at hh:mm
at h:mm
at ham
at hpm
at noon
at midnight
```

The optional *day* is either a month name followed by a day number or by a day of the week. If the word week follows, the at or batch command is invoked in seven days. Both commands also recognize standard abbreviations for the days of the week and months of the year. The following are examples of legitimate commands:

```
at 8am jan 24

at 1530 fr week
```

The `at` programs are executed by periodic execution of the command `/usr/lib/atrun` from `cron`(8). The granularity of `at` depends upon how often *atrun* is executed. The `cron` command examines the `crontab` file every minute. The `crontab` file determines when `/usr/lib/atrun` is executed. The default is every 15 minutes on the 1/4 hour. Editing `/etc/crontab` makes `/usr/lib/atrun` run more or less frequently.

Standard output or error output is lost unless it is redirected.

The `at` and `batch` commands write the job number to standard error.

Options

-r Removes jobs previously scheduled by `at` or `batch`. The number is the number reported at invocation by `at` or `batch`. Only the superuser is allowed to remove another user's jobs.

-l Used to obtain or verify the job numbers.

Restrictions

Due to the granularity of the execution of `/usr/lib/atrun`, there may be bugs in scheduling jobs almost exactly 24 hours into the future.

Diagnostics

Complains about various syntax errors and times that are out of range.

Files

/usr/lib/atrun	executor run by cron(8)
in /usr/spool/at:	
yy.ddd.hhhh.*	activity for year yy, day dd, hour hhhh.
lasttimedone	last *hhhh*
past	activities in progress
/usr/spool/at/at.allow	list of allowed users
/usr/spool/at/at.deny	list of denied users
/usr/spool/at	spool directory
/usr/lib/cron	XOPEN compatibility

See Also

crontab(5), cron(8)

Name

awk – pattern scanning and processing language

Syntax

awk [**–F***c*] [**–f** *prog*] [-] [*file*...]

Description

The awk command 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** *prog.*

Files are read in order; if there are no files, the standard input is read. The file name '–' 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, as described below.) The fields are denoted $1, $2, ... ; $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     # skip remaining patterns on this input line
exit     # skip the rest of the input
```

Statements are terminated by semicolons, new lines 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 >*file* 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. For further information, 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*. 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* 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 new line); and OFMT, the output format for numbers (default "%.6g").

Options

–	Used for standard input file.
–F*c*	Sets interfield separator to named character.
–f*prog*	Uses prog file for patterns and actions.

Examples

Print lines longer than 72 characters:

```
length > 72
```

Print first two fields in opposite order:

```
{ print $2, $1 }
```

Add up first column, print sum and average:

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

Print fields in reverse order:

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

Print all lines between start/stop pairs:

```
/start/, /stop/
```

Print all lines whose first field is different from previous one:

```
$1 != prev { print; prev = $1 }
```

Restrictions

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 "" to it.

See Also

lex(1), sed(1)
"Awk – A Pattern Scanning and Processing Language" *ULTRIX Supplementary Documents* Vol. II: Programmer

Name

basename – strip directory names from pathname

Syntax

basename *string* [*suffix*]

Description

The `basename` command deletes from *string* any prefix ending in a slash (/) and the *suffix*, and prints the result on the standard output. The `basename` command also handles limited regular expressions in the same manner as `ed(1)`. The `basename` command is often used inside substitution marks (` `) within shell procedures.

Examples

The following example shell script compiles the file `/usr/src/bin/cat.c` and moves the output to `cat` in the current directory:

```
cc /usr/src/bin/cat.c
mv a.out `basename $1 .c`
```

The following example echoes only the base name of the file `/etc/syslog.conf` by removing the prefix and any possible sequence of characters following the period in the file's name:

```
% basename /etc/syslog.conf "..*"
syslog
```

See Also

dirname(1), ex(1), sh(1)

Name

bc – interactive arithmetic language processor

Syntax

bc [**–c**] [**–l**] [*file*...]

Description

The bc command provides an interactive processor for a language which 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 new lines 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.

The following example defines a function to compute an approximate value of the exponential function:

```
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
       }
}
```

The following command line then prints approximate values of the exponential function of the first ten integers:

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

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

Options

–c Compiles input only.

–l Names arbitrary precision math library.

Restrictions

The *for* statement must have all three E's.

Quit is interpreted when read, not when executed.

Variables must be a single lower case letter. Upper case letters are used only as digits for bases greater than 10.

Files

`/usr/lib/lib.b` mathematical library

See Also

dc(1)
"BC – An arbitrary precision desk-calculator language" *ULTRIX Supplementary Documents* Vol. 1: General User

Name

bdiff – big file differential comparator

Syntax

bdiff *file1 file2* [*n*] [–s]

Description

The bdiff command is used to find lines that must be changed in two files to bring them into agreement. Its purpose is to allow processing of files that are too large for diff(1).

The bdiff command ignores lines common to the beginning of both files, splits the remainder of each file into *n*-line segments, and invokes diff(1) upon corresponding segments. The value of *n* is 3500 by default. If the optional third argument is given and if 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.

The output of the bdiff command is the same as the output of the diff command: line numbers are adjusted to account for the segmenting of the files 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 the smallest sufficient set of file differences.

If either *file1* or *file2* is –, the standard input is read. The optional –s (silent) argument specifies that no diagnostics are to be printed by bdiff. However, this does not suppress possible exclamations by diff. If both optional arguments are specified, they must appear in the order indicated above.

Options

–s Suppresses normal diagnostic messages.

Diagnostics

Use sccshelp(1) for explanations.

Files

/tmp/bd?????

See Also

diff(1)

Name

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

Syntax

biff [**yn**]

Description

The `biff` command informs the system whether you want to be notified when mail arrives during the current terminal session. The command, `biff` **y**, enables notification; the command, `biff` **n**, disables it. The `biff` command with no options displays the current status of `biff`.

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.

The `biff` command operates asynchronously. For synchronous notification use the MAIL variable of `sh`(1) or the `mail` variable of `csh`(1).

Options

–n	Disables notification that mail has arrived.
–y	Enables notification that mail has arrived.

See Also

csh(1), mail(1), sh(1), comsat(8c)

Name

binmail – send or receive mail among users

Syntax

/bin/mail [+] [–i] [*person...*]
/bin/mail [+] [–i] **–f** *file*

Description

This is the old version 7 UNIX system mail program. The default `mail` command is described in `mail(1)`, and its binary is in the directory `/usr/ucb.` The `/bin/mail` program is still used to actually deliver a mail message into the users system-wide mailbox (`/usr/spool/mail/*`), however, the reading of these messages has been replaced with the program `/usr/ucb/mail.` Do not remove `/bin/mail` from your system.

The `mail` command with no argument prints a user's mail, message-by-message, in last-in, first-out order; the optional argument + displays the mail messages in first-in, first-out order. For each message, it reads a line from the standard input to direct the disposition of the message.

Issue the following commands from the `mail` program prompt:

<CR>	Go on to next message
d	Delete message and go on to the next.
p	Print message again.
–	Go back to previous message.
s [*file...*]	Save the message in the named *files* ('mbox' default).
w [*file...*]	Save the message, without a header, in the named *files* ('mbox' default).
m [*person...*]	Mail the message to the named *persons* (yourself is default).
EOT (control-D)	Put unexamined mail back in the mailbox and stop.
q	Same as EOT.
!*command*	Escape to the Shell to do *command.*
*	Print a command summary.

An interrupt normally terminates the `mail` command; the mail file is unchanged.

When *persons* are named, `mail` takes the standard input up to an end-of-file (or a line with just '.') and adds it to each *person's* mail file. The message is preceded by the sender's name and a postmark. Lines that look like postmarks are prepended with '>'. A *person* is usually a user name recognized by `login(1)`. To denote a recipient on a remote system, prefix *person* by the system name and exclamation mark. For further information, see `uucp(1c)`.

The `mail` program sends a message to the screen that there is mail when the user logs in.

When /bin/mail is used to deliver mail, (usually sendmail(8) calls /bin/mail to do this), a mailbox is created for the user in the directory /usr/spool/mail, if it doesn't already exist. The mailbox is created with the mode 700 so that only its owner can access it. In addition, the directory /usr/spool/mail has the mode 777 with the sticky bit set. The mode is 777 so that other mail programs, notably /usr/ucb/mail, can create the appropriate lock files to prevent another process from writing to the mailbox at the same time. The sticky bit set on the directory prevents one user from unlinking another user's mailbox.

Options

–f Displays mail messages contained in the specified file (next argument) in place of your mailbox file.

–i Notifies mail to continue after interrupts.

Restrictions

Race conditions sometimes result in a failure to remove a lock file.

Files

/etc/passwd	to identify sender and locate persons
/usr/spool/mail/*	incoming mail for user *
mbox	saved mail
/tmp/ma*	temp file
/usr/spool/mail/*.lock	lock for mail directory
dead.letter	unmailable text

See Also

mail(1), uucp(1c), uux(1c), write(1), sendmail(8)

Name

burst – explode digests into messages

Syntax

burst [+folder] ***[msgs]*** **[–inplace] [–noinplace] [–quiet] [–noquiet] [–verbose] [–noverbose] [–help]**

Description

The `burst` command extracts the original messages from a forwarded message, discards the forwarder's header details and places the burst message at the end of the current folder. You can specify messages, other than the current forwarded message, by using `burst` with the <+folder> and <msgs> arguments. If you specify a message, that message becomes the current folder. If you specify a folder, that folder becomes the current folder.

You can use `burst` to expand a message, that contains a number of messages that have been packed into one file for ease of mailing, into its constituent messages. The `packf` and `forw` commands can both pack individual messages into a single message or file.

The `burst` command can also be used on Internet digests.

As an example of the way in which `burst` can be used, imagine that you have gone on a business trip and have been allocated a guest account on a local machine. While you are away you redirect all your mail to the local machine. At the end of the trip, there is some mail that you want to keep when you return. Rather than send a number of mail messages, you can use `forw` to pack all the individual messages into one large message, and forward it to your normal account (after disabling the redirection).

When you return you can use `burst` to expand the single message into its constituent messages.

Options

If `-inplace` is given, each digest is replaced by the table of contents for the digest. The original digest is removed. The `burst` command then renumbers all of the messages in the folder following the digest to make room for each of the messages contained within the digest. These messages are placed immediately after the digest.

If `-noinplace` is given, each digest is preserved, no table of contents is produced, and the messages contained within the digest are placed at the end of the folder. Other messages are not tampered with in any way.

The `-quiet` switch directs `burst` to be silent about reporting messages that are not in digest format.

The `-verbose` switch directs `burst` to tell you the general actions that it is taking to explode the digest.

The burst command has the following defaults:

`+folder` defaults to the current folder
`msgs` defaults to cur
`-noinplace`

```
-noquiet
-noverbose
```

If `-inplace` is given, then the first message burst becomes the current message.

This leaves the context ready for a `show` of the table of contents of the digest, and a `next` to see the first message of the digest. If `-noinplace` is given, then the first message extracted from the first digest burst becomes the current message. This leaves the context in a similar, but not identical, state to the context achieved when using `-inplace`.

The `burst` program enforces a limit on the number of messages which may be expanded from a single message. This number is about 1000 messages. However, there is usually no limit on the number of messages which may reside in the folder after the messages have been expanded.

Although `burst` uses a sophisticated algorithm to determine where one encapsulated message ends and another begins, not all programs that create digests use an encapsulation algorithm. The `burst` command only works on messages that have been encapsulated according to the guidelines laid down by the proposed standard RFC 934. This basically means that the encapsulated message is considered to start after `burst` encounters a line of dashes. If you attempt to `burst` a message that has not been encapsulated according to RFC 934, the results may be unpredictable. In most cases, this means that `burst` may find an encapsulation boundary prematurely and split a single encapsulated message into two or more messages.

Furthermore, any text which appears after the last encapsulated message is not placed in a separate message by `burst`. In the case of encapsulated messages, this text is usually an End-of-digest string. Note that when the `-inplace` option is used, this trailing information is lost. However, in practice this is not a problem, since correspondents usually place remarks in text prior to the first encapsulated message, and this information is not lost.

Files

`$HOME/.mh_profile`	The user profile

Profile Components

`Path:`	To determine your MH directory
`Current-Folder:`	To find the default current folder
`Msg-Protect:`	To set mode when creating a new message

See Also

inc(1mh), msh(1mh)
Proposed Standard for Message Encapsulation (RFC 934)

Name

cal – print calendar

Syntax

cal [*month*] *year*

Description

The `cal` command prints a calendar for the specified year. If a month is also specified, a calendar just for that month is printed. The *year* can be between 1 and 9999. The *month* is a number between 1 and 12. The following example produces a calendar for October 1988.

```
cal 10 1988
```

Restrictions

The year is always considered to start in January.

Name

calendar – calendar reminder service

Syntax

calendar [–]

Description

The `calendar` command consults the file 'calendar' in the current directory and prints out lines that contain today's or tomorrow's date. The `calendar` command recognizes most month-day dates, such as Dec. 7, december 7, 12/7, but it does not recognize dates formatted in the following ways: 7 December or 7/12. If you give the month as * with a date, such as, * 1, that day in any month will do. On weekends, specifying tomorrow extends through Monday.

When an argument is present, the `calendar` command searches through a user's calendar file in his login directory and sends him any positive results by `mail`(1). Normally this is done daily under control of `cron`(8).

The calendar file is first run through the C preprocessor, `/lib/cpp,` to include any other calendar files specified with the #include syntax. Included calendars are shared by all users, and are maintained and documented by the local administration.

Options

– Functions for every user who has a calendar file in his login directory.

Restrictions

The `calendar's` extended idea of tomorrow does not account for holidays.

Files

calendar
/usr/lib/calendar to figure out today's and tomorrow's dates
/etc/passwd
/tmp/cal*
/lib/cpp, egrep, sed, mail as subprocesses

See Also

at(1), cron(8), mail(1)

Name

capsar – prepares documents not in ASCII format for transport in the mail system

Syntax

capsar [-c] [-t] [-x[hTD]] [*file*]

Description

The `capsar` utility allows ULTRIX mail to support documents containing non-ASCII data, such as DDIF. Only the DDIF and DOTS data types are currently supported. DDIF is Digital's standard format for document interchange. DOTS is an encapsulation of the encoded interchange form of a number of related data objects into a single composite object. For more information, see `DDIF`(5) and `DOTS`(5).

The `capsar` utility prepares a DOTS file or a DDIF document for transport in the mail system by performing the following steps:

1) The DDIF document is converted to DOTS format. As a DDIF document may contain more than one file, all files within the DDIF document are incorporated into one DOTS file which can be sent as one mail message.

2) Each DOTS file is then compressed and encoded using only printing ASCII characters. This is because ULTRIX mail software only supports 7 bit mail.

3) The `capsar` routine encapsulates coded documents by adding leading and trailing lines, each surrounded by a <CR>. The lines should begin with 2 or more dashes (–) and some text that indicates the nature of the encapsulated message. The following is a typical encapsulated mail message:

```
To: anybody@anynode
Cc:
Subject: Another DDIF document


-----------motd.ddif : DOTS.ctod.compress.uuencode message

begin 0 motd.ddif
M__]@*" ,(" !BO.#P$# 8$* &UO=&0N9&1I9H0$)%546     "A@"B !@8K
MS@ P$''T1$248M96YC;V1E9"!R979I<V%B;&4@9&]C=6UE;G2@@/__?X"@
M@( ! 8$! ((/1$1)1B1?4D5!1%]415A4HX#)% !$1$E&(%1E>'0@1G)O;G0@

end

-----------End of motd.ddif : DOTS.ctod.compress.uuencode message
```

The `capsar` command can also extract different parts of a mail message, namely, the header information, the text part of the message, and the DOTS file that was encapsulated as described above.

Extracting the DOTS file is done by parsing the mail message and detecting the leading and trailing encapsulation boundaries. Decoding and uncompressing the data results in the original DOTS file.

The `capsar` utility is built into Rand MH to provide DDIF mail support. It can, however, be used with `ucb` mail.

Options

–c Causes `capsar` to create an encapsulated DOTS bodypart from *file*. The *file* must be a DOTS/DDIF type document.

–t Causes `capsar` to write to the standard output the message type of *file*. Message type can be either text or DOTS.

–xh Extracts the mail header lines from *file*. The header line must be at the beginning of the *file* and separated from the remaining text by a <CR> or <CRLF>. Each header line is a string containing a header field name (for example, Subject), a colon (:), one or more spaces, and a field value. Each header line may have embedded continuation sequences it it (for example, LF followed by spaces or tabs).

–xT Extracts all the text parts of the mail message in *file* to the standard output.

–xD Extracts any DOTS bodyparts in *file*. The DOTS document is sent to the standard output. This is the reverse of the **–c** option above.

The *file* must be specified for the **–c** option. If *file* is not specified with the **–x** or **–t** option then the standard input is used.

Examples

The following are examples of how to use the `capsar` command:

Encapsulates a DDIF document

```
capsar -c file.ddif | more
```

Lists the header line from the mail message

```
capsar -xh file.mail
```

Extracts the encapsulated DOTS file from the file

```
capsar -xD file > file.dots

       or

capsar -xD file | dtoc
```

In order to mail a DDIF/DOTS document you can use one of the following:

```
capsar -c file.ddif | mail -s "subject" address

capsar -c file.ddif | mhmail -subject "subject" address
```

Use the second command if you are using RAND mh.

A DOTS file is extracted from dxmail first extracting the message into a file. The dxmail utility has an extract feature built in so `capsar -xD` isn't needed.

See Also

compress(1), ctod(1), dtoc(1), mail(1), mh(1mh), mhmail(1mh), uuencode(1), vdoc(1), prompter(1mh), DDIF(5), DOTS(5)

Name

cat – concatenate and print data

Syntax

cat [**–b**] [**–e**] [**–n**] [**–s**] [**–t**] [**–u**] [–v] *file*...

Description

The `cat` command reads each *file* in sequence and displays it on the standard output. Therefore, to display the file on the standard output you type:

```
cat file
```

To concatenate two files and place the result on the third you type:

```
cat file1 file2 > file3
```

If no input file is given, or if a minus sign (–) is encountered as an argument, `cat` reads from the standard input file. Output is buffered in 1024-byte blocks unless the standard output is a terminal, in which case it is line buffered. The `cat` utility supports the processing of 8-bit characters.

Options

–b Ignores blank lines and precedes each output line with its line number.

–e Displays a dollar sign ($) at the end of each output line.

–n Precedes all output lines (including blank lines) with line numbers.

–s Squeezes adjacent blank lines from output and single spaces output.

–t Displays non-printing characters (including tabs) in output. In addition to those representations used with the **–v** option, all tab characters are displayed as ^I.

–u Unbuffers output.

–v Displays non-printing characters (excluding tabs). For example, <CTRL/X> displays on the screen as ^X. The delete character (octal 0177) displays as ^?. Non-ASCII characters (with the high bit set) display as M- (which is the meta character) followed by the low 7 bits.

See Also

cp(1), ex(1), more(1), pr(1), tail(1)

Name

catpw – prints all password entries

Syntax

catpw

Description

The `catpw` command prints password entries for all users known to the system using the format in `passwd`(5). Password entries are gathered from all sources including Yellow Pages, Kerberos, and `/etc/passwd`.

See Also

getpwent(3), passwd(5)

cb(1)

Name

cb – C program beautifier

Syntax

cb

Description

The `cb` command places a copy of the C program from the standard input on the standard output with spacing and indentation that displays the structure of the program.

Name

cc – RISC C compiler

Syntax

cc [*option*] ... *file*

Description

The cc command invokes the RISC *ucode* C compiler. It produces RISC object code in RISC extended *coff* format (the default), binary or symbolic *ucode*, *ucode* object files and binary or symbolic assembly language.

The cc command accepts the following arguments:

- Arguments ending in .c are interpreted as C source programs. They are compiled, and the resulting object file has the same name as the source program except .o is substituted for .c. If a single C source program is compiled and loaded at once, the .o file is deleted.
- Arguments ending in .s are interpreted as assembly source programs. When they are assembled, they produce a .o file.
- Arguments ending in .i are interpreted as C source after being processed by the C preprocessor. They are compiled without being processed by the C preprocessor.

If the highest level of optimization is specified (with the **–O3** flag) or only *ucode* object files are to be produced (with the **–j** flag) each C source file is compiled into a *ucode* object file. The *ucode* object file is left in a file whose name consists of the last component of the source with .u substituted for .c.

The following suffixes aid compiler development, but are not generally used: .B, .O., .S, and .M. These arguments are interpreted as binary *ucode*, produced by the front end, optimizer, ucode object file splitter, and ucode merger respectively. Arguments whose names end with .U are assumed to be symbolic *ucode*. Arguments whose names end with .G are assumed to be binary assembly language, which is produced by the code generator and the symbolic to binary assembler.

Files that are assumed to be binary *ucode*, symbolic *ucode*, or binary assembly language by the suffix conventions are also assumed to have their corresponding symbol table in a file with a .T suffix.

The cc command always defines the C preprocessor macro **LANGUAGE_C** when a .c file is being compiled. The cc command defines the C preprocessor macro **LANGUAGE_ASSEMBLY** when a .s file is compiled.

Options

The following options are interpreted by cc(1). See ld(1) for load-time options.

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

–g0 Do not produce symbol table information for symbolic debugging. This is the default.

–g1 Produce additional symbol table information. Provides accurate, but limited symbolic debugging of partially optimized code.

–g or **–g2** Produce additional symbol table information for full symbolic debugging, but do not perform optimizations that limit full symbolic debugging.

–g3 Produce additional symbol table information for full symbolic debugging for fully optimized code. This option can affect debugger accuracy.

–w Suppress warning messages.

–p0 Do not permit profiling. This is the default. If loading happens, the standard runtime startup routine (**crt0.o**) is used and the profiling libraries are not searched.

–p1 or **–p** Set up for profiling by periodically sampling the value of the program counter. This option only affects the loading. When loading happens, this option replaces the standard runtime startup routine with the profiling runtime startup routine (**mcrt0.o**) and searches the level 1 profiling library (**libprof1.a**). When profiling happens, the startup routine calls `monstartup`(3) and produces a file *mon.out* that contains execution-profiling data for use with the postprocessor `prof`(1).

–O0 Turn off all optimizations.

–O1 Turn on all optimizations that complete fast. This is the default.

–O or **–O2** Invoke the global *ucode* optimizer.

–O3 Perform all optimizations, including global register allocation. This option must precede all source file arguments. With this option, a *ucode* object file is created for each C source file and left in a .u file. The newly created ucode object files, the ucode object files specified on the command line, the runtime startup routine, and all the runtime libraries are ucode linked. Optimization is performed on the resulting ucode linked file and then it is linked as normal producing an a.out file. A resulting .o file is not left from the ucode linked result. In fact **–c** cannot be specified with **–O3**.

–feedback *file* Use with the **–cord** option to specify the feedback file. This *file* is produced by `prof`(1) with its **–feedback** option from an execution of the program produced by `pixie`(1).

–cord Run the procedure-rearranger on the resulting file after linking. The rearrangement is performed to reduce the cache conflicts of the program's text. The output is left in the file specified by the **–o** *output* option or a.out by default. At least one **–feedback** *file* must be specified.

–j Compile the specified source programs, and leave the *ucode* object file output in corresponding files with the .u suffix.

–ko *output* Name the output file created by the ucode loader as *output*. This file is not removed. If this file is compiled, the object file is left in a file whose name consists of *output* with the suffix changed to an

.o. If *output* has no suffix, an .o suffix is appended to *output*.

–k Pass options that start with a **–k** to the ucode loader. This option is used to specify ucode libraries (with **–kl***x*) and other ucode loader options.

–S Compile the specified source programs and leave the symbolic assembly language output in corresponding files suffixed with .s.

–P Run only the C macro preprocessor and put the result for each source file using suffix convention (for example, .c and .s) in a corresponding .i file. The .i file does not have number lines (#) in it. This sets the **–cpp** option.

–E Run only the C macro preprocessor on the files (regardless of any suffix or not), and send the result to the standard output. This sets the **–cpp** option.

–o *output* Name the final output file *output*. If this option is used, the file a.out is unaffected.

–D*name=def*
–D*name* Define the *name* to the C macro preprocessor, as if by '#define'. If a definition is not given, the name is defined as 1.

–U*name* Remove any initial definition of *name*.

–I*dir* Search for #include files whose names do not begin with a slash (/) in the following order: (1) in the directory of the *dir* argument, (2) in the directories specified by **–I** options, (3) in the standard directory (**/usr/include**).

–I Do not search for #include in the standard directory (**/usr/include**).

–G *num* Specify the maximum size, in bytes, of a data item that is to be accessed from the global pointer. The *num* argument is interpreted as a decimal number. If *num* is zero, data is not accessed from the global pointer. The default value for *num* is 8 bytes.

–v Print the passes as they execute with their arguments and their input and output files. Also prints resource usage in the C shell *time* format.

–V Print the version of the driver and the versions of all passes. This is done with the `what(1)` command.

–std Produce warnings for things that are not standard in the language.

–Y*environment* Compiles C programs for *environment*. If *environment* is SYSTEM_FIVE or is omitted, it defines SYSTEM_FIVE for the preprocessor, `cpp`. If the loader is invoked, it specifies that the System V version of the C runtime library is used. Also, if the math library is specified with the **–lm** option, the System V version is used. If *environment* is POSIX, it defines POSIX for the preprocessor. If the environment variable PROG_ENV has the value SYSTEM_FIVE or POSIX, the effect is the same as when specifying the corresponding **–Y***environment* option to `cc`. The **–Y** option overrides the PROG_ENV variable; **–YBSD** can be used to override all special actions.

–cpp Run the C macro preprocessor on C and assembly source files before compiling. This is the default for cc(l).

–nocpp Do not run the C macro preprocessor on C and assembly source files before compiling.

–Olimit *num* Specify the maximum size, in basic blocks, of a routine that will be optimized by the global optimizer. If a routine has more than the specified number of basic blocks, it cannot be optimized and a message is printed. A **–O, –O2,** or **–O3** must be used to specify the global optimizer. The argument must also be specified. The argument *num* is interpreted as a decimal number. The default value for *num* is 1500 basic blocks.

–signed Causes all *char* declarations to be *signed char* declarations. This is the default.

–unsigned Causes all *char* declarations to be *unsigned char* declarations.

–volatile Causes all variables to be treated as *volatile.*

–varargs Prints warnings for lines that may require the *varargs.h* macros.

–f Causes the compiler not to promote expressions of type *float* to type *double*.

NOTE

The **–EB** and **–EL** options are needed only when compiling for RISC machines from vendors other than Digital.

The default target byte ordering matches the machine where the compiler is running. The options **–EB** and **–EL** specify the target byte ordering (big-endian and little-endian, respectively). The compiler also defines a C preprocessor macro for the target byte ordering. These C preprocessor macros are **MIPSEB** and **MIPSEL** for big-endian and little-endian byte ordering respectively.

If the specified target byte ordering does not match the machine where the compiler is running, then the runtime startups and libraries come from **/usr/libeb** for big-endian runtimes on a little-endian machine and from **/usr/libel** for little-endian runtimes on a big-endian machine.

–EB Produce object files targeted for big-endian byte ordering. The C preprocessor macro **MIPSEB** is defined by the compiler.

–EL Produce object files targeted for little-endian byte ordering. The C preprocessor macro **MIPSEL** is defined by the compiler.

The following options primarily aid compiler development and are not generally used:

–H*c* Halt compiling after the pass specified by the character *c*, producing an intermediate file for the next pass. The *c* can be [**fjusmoca**]. It selects the compiler pass in the same way as the **–t** option. If this option is used, the symbol table file produced and used by the passes is the last component of the source file with the suffix changed to .T. It is not removed.

–K Build and use intermediate file names with the last component of the source file's name replacing its suffix with the conventional

suffix for the type of file (for example, .B file for binary *ucode,* produced by the front end). These intermediate files are never removed even when a pass encounters a fatal error. When ucode linking is performed and the **–K** option is specified, the base name of the files created after the ucode link is u.out by default. If **–ko** *output* is specified, the base name of the object file is *output* without the suffix. Suffixes are appended to *output* if it does not have a suffix.

–#
Converts binary *ucode* files (.B) or optimized binary ucode files (.O) to symbolic *ucode* (a .U file). If a symbolic ucode file is to be produced by converting the binary *ucode* from the C compiler front end then the front end option **–Xu is used.**

–W*c[c...],arg1[,arg2...]*
Pass the argument[s] *argi* to the compiler pass[es] *c[c..]*. The *c's* are one of **[pfjusmocablyz].** The c's selects the compiler pass in the same way as the **–t** option.

The options **–t[hpfjusmocablyzrnt]**, **–h***path,* and **–B***string* select a name to use for a particular pass, startup routine, or standard library. These arguments are processed from left to right so their order is significant. When the **–B** option is encountered, the selection of names takes place using the last **–h** and **–t** options. Therefore, the **–B** option is always required when using **–h** or **–t**. Sets of these options can be used to select any combination of names.

The **–EB** or **–EL** options and the **–p[01]** options must precede all **–B** options because they can affect the location of runtime libraries and which runtime libraries are used.

–t[hpfjusmocablyzrnt]
Select the names. The names must be selected from the options in the following table:

Name	Character
include	h (see note below)
cpp	p
ccom	f
ujoin	j
uld	u
usplit	s
umerge	m
uopt	o
ugen	c
as0	a
as1	b
ld	l
ftoc	y
cord	z
[m]crt0.o	r
libprof1.a	n
btou, utob	t

If the character h is in the **–t** argument then a directory is added to the list of directories to be used in searching for #include files. This directory name has the form COMP_TARGET_ROOT/usr/include*string* . This directory is to contain

the include files for the *string* release of the compiler. The standard directory is still searched.

–h*path* Use *path* rather than the directory where the name is normally found.

–B*string* Append *string* to all names specified by the **–t** option. If the **–t** option has not been processed before the **–B,** the **–t** option is assumed to be the following: hpfjusmocablyzrnt. This list designates all names. If the **–t** argument has not been processed before the **–B** argument, **–B***string* is passed to the loader to use with its **–l***x* arguments.

Invoking the compiler with a name of the form cc*string* has the same effect as using a **–B***string* option on the command line.

If the environment variable COMP_HOST_ROOT is set, the value is used as the root directory for all pass names rather than the default slash (/). If the environment variable COMP_TARGET_ROOT is set, the value is used as the root directory for all include and library names rather than the default slash (/). This affects the standard directory for #include files, /usr/include, and the standard library, /usr/lib/libc.a. If this is set then the only directory that is searched for libraries, using the **–l***x* option, is COMP_TARGET_ROOT/usr/lib .

If the environment variable TMPDIR is set, the value is used as the directory to place any temporary files rather than the default **/tmp/ .**

If the environment variable RLS_ID_OBJECT is set, the value is used as the name of an object to link in if a link takes place. This is used to add release identification information to objects. It is always the last object specified to the loader.

Other arguments are assumed to be either loader options or C–compatible object files, typically produced by an earlier cc run, or perhaps libraries of C–compatible routines. These files, together with the results of any compilations specified, are loaded in the order given, producing an executable program with the default name **a.out.**

Options

The ULTRIX C compiler provides the following default symbols for your use. These symbols are useful in ifdef statements to isolate code for one of the particular cases. Thus, these symbols can be useful for ensuring portable code.

unix Any UNIX system

bsd4_2 Berkeley UNIX Version 4.2

ultrix ULTRIX only

mips Any RISC architecture

MIPSEL Little endian variant of MIPS architecture

host_mips Native compilation environment (as opposed to cross-compiler)

Restrictions

The standard library, /usr/lib/libc.a, is loaded by using the –lc loader option and not a full path name. The wrong library may be loaded if there are files with the name libc.a*string* in the directories specified with the –L loader option or in the default directories searched by the loader.

The handling of include directories and libc.a is confusing.

Files

file.c	input file
file.o	object file
a.out	loaded output
/tmp/ctm?	temporary
/usr/lib/cpp	C macro preprocessor
/usr/lib/ccom	C front end
/usr/lib/ujoin	binary ucode and symbol table joiner
/usr/bin/uld	ucode loader
/usr/lib/usplit	binary ucode and symbol table splitter
/usr/lib/umerge	procedure intergrator
/usr/lib/uopt	optional global ucode optimizer
/usr/lib/ugen	code generator
/usr/lib/as0	symbolic to binary assembly language translator
/usr/lib/as1	binary assembly language assembler and reorganizer
/usr/lib/crt0.o	runtime startup
/usr/lib/mcrt0.o	startup for profiling
/usr/lib/libc.a	standard library, see intro(3)
/usr/lib/libprof1.a	level 1 profiling library
/usr/include	standard directory for #include files
/usr/bin/ld	MIPS loader
/usr/lib/ftoc	interface between `prof(1)` and cord
/usr/lib/cord	procedure-rearranger
/usr/bin/btou	binary to symbolic ucode translator
/usr/bin/utob	symbolic to binary ucode translator
mon.out	file produced for analysis by `prof(1)`

Runtime startups and libraries for the opposite byte sex of machine the compiler is running on have the same names but are located in different directories. For big-endian runtimes on a little-endian machine the directory is /usr/libeb and for little-endian runtimes on a big-endian machine the directory is /usr/libel.

See Also

dbx(1), ld(1), pixie(1), prof(1), what(1), monitor(3)

Name

cc – C compiler

Syntax

cc [*option...*] *file...*

Description

The cc command invokes the ULTRIX C compiler and accepts the following types of arguments:

- Arguments whose names end with .c
- Arguments whose names end with .s
- Other arguments that are interpreted as either loader option arguments or C-compatible object programs

Arguments ending in .c are interpreted as C source programs. They are compiled, and each object program is left on a file whose name is the same as the source file except .o is substituted for .c. If a single C program is compiled and loaded all at once, the .o file is deleted.

Arguments ending with .s are interpreted as assembly source programs. They are assembled, producing an .o file.

Arguments other than those ending with .c or .s were produced by previous cc runs or by libraries of C-compatible routines.

The first argument passed to the ld(1) loader is always one of the three crt0 files used for start up. The compiler uses /lib/mcrt0.o when the **–p** flag is given, /usr/lib/gcrt0.o when the **–pg** is given, and /lib/crt0.o otherwise. If loading executables by hand, you must include the appropriate file.

Options

These options are accepted by cc. See ld(1) for load-time options.

–b Does not pass **–lc** to ld(1) by default.

–B*string* Finds substitute compiler passes in the files named *string* with the suffixes cpp, ccom, and c2.

–c Suppresses the loading phase of the compilation and forces an object file to be produced even if only one program is compiled.

–C Stops the macro preprocessor from omitting comments.

–D*name=def*
–D*name* Defines the *name* to the processor, as if by #define. If no definition is given, the name is defined as 1.

–E Runs only the macro preprocessor on the named C programs and sends the result to the standard output.

–Em Runs only the macro preprocessor on the named C programs and produces the makefile dependencies.

–f Specifies that computations involving only FFLOAT numbers be done in single precision and not promoted to double. Procedure arguments are still promoted to double. Programs with a large number of single-precision computations will run faster with this option; however, a slight loss in precision may result due to the saving of intermediate results in a single-precision representation.

–g Directs the compiler to produce additional symbol table information for `dbx`(1). Also passes the **–lg** flag to `ld`(1).

–I*dir* Searches first in the directory of the *dir* argument for **#include** files whose names do not begin with a slash (/), then in directories named in **–I** options, and, finally, in directories on a standard list.

–l*x* Abbreviates the library name /lib/lib*x*.a, where *x* is a string. If that library name does not exist, **ld** searches /usr/lib/lib*x*.a and then /usr/local/lib/lib*x*.a. The placement of the **–l** library option is significant because a library is searched when its name is encountered.

–M Specifies the floating point type to be used for double-precision floating point and is passed on to `ld`(1) as the map option.

–Md Specifies the default DFLOAT and passes the **–lc** flag to `ld`(1).

–Mg Specifies GFLOAT and passes the **–lcg** flag to `ld`(1), causing the GFLOAT version of **libc** to be used. If the math library is used with code compiled with the **–Mg** flag, it is linked to the GFLOAT version by specifying **–lmg** on the `cc`(1) or `ld`(1) command.

–o *output* Names the final output file *output*. If this option is used, the file a.out is left alone. If the named file has either .o or .a as a suffix, the following error message is displayed: **-o would overwrite.**

–O Uses the object code optimizer.

–p Arranges for the compiler to produce code which counts the number of times each routine is called. If loading takes place, the **–p** option replaces the standard startup routine with one that automatically calls `monitor`(3) and that arranges to write out a mon.out file at normal termination of execution of the object program. An execution profile can then be generated using `prof`(1).

–pg Causes the compiler to produce counting code as with **–p,** but invokes a run-time recorder that keeps more extensive statistics and produces a `gmon.out` file. Also, the **–pg** option searches a profiling library in lieu of the standard C library. An execution profile can then be generated by using `gprof`(1).

–R Passed on to *as,* which makes initialized variables shared and read-only.

–S Compiles programs and writes output to .s files.

–t [p02al] Finds the designated compiler passes in the files whose names are constructed by a **–B** option. In the absence of a **–B** option, the *string* is taken to be /usr/c/.

–U*name*	Removes any initial definition of *name*.
–w	Suppresses warning diagnostics.
–Y*environment*	Compiles C programs for *environment*. If *environment* is SYSTEM_FIVE or is omitted, it defines SYSTEM_FIVE for the preprocessor, `cpp`. If the loader is invoked, it specifies that the System V version of the C runtime library is used. Also, if the math library is specified with the **–lm** option, the System V version is used. If *environment* is POSIX, it defines POSIX for the preprocessor. If the environment variable PROG_ENV has the value SYSTEM_FIVE or POSIX, the effect is the same as when specifying the corresponding **–Y***environment* option to `cc`. The **-Y** option overrides the PROG_ENV variable; **-YBSD** can be used to override all special actions.

Default Symbols

The ULTRIX C compiler provides the following default symbols for your use. These symbols are useful in ifdef statements to isolate code for one of the particular cases. Thus, these symbols can be useful for ensuring portable code.

`unix`	Any UNIX system
`bsd4_2`	Berkeley UNIX Version 4.2
`ultrix`	ULTRIX only
`vax`	VAX only (as opposed to PDP-11)

Restrictions

The compiler ignores advice to put **char**, **unsigned char**, **short** or **unsigned short** variables in registers.

If the **–Mg** flag is used to produce GFLOAT code, it must be used when compiling all the modules which will be linked. Use the **–Mg** flag if you use the **cc** command to invoke `ld`(1) indirectly to link the modules. If `ld`(1) is invoked directly, use the **–lcg** flag rather than **–lc.** If the math library is used, specify the **–lmg** flag rather than the **–lm** flag in order to use the GFLOAT version.

The compiler and the linker `ld`(1) cannot detect the use of mixed double floating point types. If you use them, your program's results may be erroneous.

Diagnostics

The diagnostics produced by C are intended to be self-explanatory. Occasional messages may be produced by the assembler or loader.

Files

file.c	input file
file.o	object file
a.out	loaded output
/tmp/ctm?	temporary
/lib/cpp	preprocessor
/lib/ccom	compiler
/lib/c2	optional optimizer
/lib/crt0.o	runtime startoff
/lib/mcrt0.o	startoff for profiling

/usr/lib/gcrt0.o	startoff for gprof-profiling
/lib/libc.a	standard library, see intro(3)
/usr/libcg.a	GFLOAT version of the standard library, see intro(3)
/usr/lib/libc_p.a	profiling library, see intro(3)
/usr/include	standard directory for #include files
mon.out	file produced for analysis by prof(1)
gmon.out	file produced for analysis by gprof(1)

See Also

adb(1), as(1), cpp(1), dbx(1), error(1), gprof(1), ld(1), prof(1), monitor(3)

Name

cd – change current directory

Syntax

cd *directory*

Description

The *directory* becomes the new working directory. The process must have execute (search) permission in *directory*.

Because a new process is created to execute each command, `cd` would be ineffective if it were written as a normal command. It is therefore recognized and executed by the shells. In `csh` you may specify a list of directories in which *directory* is to be sought as a subdirectory if it is not a subdirectory of the current directory; see the description of the *cdpath* variable in `csh`(1).

See Also

csh(1), pwd(1), sh(1), chdir(2)

Name

cdc – change delta commentary of an SCCS file

Syntax

cdc -r*SID* [**-m**[*mrlist*]] [**-y**[*comment*]] *files*

Description

The cdc command changes the delta commentary of each named SCCS file, for the *SID* specified by the **-r** option.

The delta commentary is defined to be the Modification Request (MR) and comment information usually specified by the delta command (**-m** and **-y** options).

The delta commentary may consist of one or more lines, terminated by a dot in column one of a new line.

If a directory is named, cdc behaves as though each file in the directory were specified as a named file, except non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files, which are silently ignored. If a name of – is given, the standard input is read (see RESTRICTIONS). 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 option arguments, and file names.

All the described option arguments apply independently to each named file.

Options

-m[*mrlist*]
Adds or deletes modification numbers. If the SCCS file has the v flag set then a list of MR numbers to be added and/or deleted in the delta commentary of the *SID* specified by the **-r** option may be supplied. For further information, see admin(1). 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 is 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** option).
MRs in a list are separated by blanks and/or tab characters. An unescaped new-line character terminates the MR list.

Note that if the **v** flag has a value it is taken to be the name of a program (or shell procedure) which validates the correctness of the MR numbers. For further information, see admin(1). If a

nonzero exit status is returned from the MR number validation program, cdc terminates and the delta commentary remains unchanged.

–r*SID* Specifies the SCCS Identification string of a delta for which the delta commentary is to be changed.

–y[*comment*] Replaces existing commentary for the delta specified by the **–r** option. 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. A dot in column one of a new line terminates the *comment* text.

Certain permissions are necessary to modify the SCCS file; generally, however, if you made the delta, you can change its delta commentary, and if you own the file and directory you can modify the delta commentary.

Examples

This example shows how to add bl78-12345 and bl79-00001 to the MR list, remove bl77-54321 from the MR list, and add the comment **"trouble"** to delta 1.6 of s.file.

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

This example does the same thing.

```
sccscdc -r1.6 .file

-MRs? !bl77-54321 bl78-12345 bl79-00001

comments? trouble
```

Restrictions

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

Diagnostics

See sccshelp(1) for explanations.

Files

x-file For more information, see delta(1)

z-file For more information, see delta(1)

See Also

admin(1), delta(1), get(1), help(1), prs(1), sccs(1), sccsfile(5)
Guide to the Source Code Control System

Name

cdoc – invokes CDA Converter

Syntax

cdoc [–s *format*] [–d *format*] [–O *options_file*] [–o *outputfile*] *inputfile*

Description

The cdoc command converts the revisable format file, *inputfile*, to another revisable format or to a final form file. If *inputfile* is not specified, cdoc reads from standard input. Unless a destination file is specified with the –o option, the cdoc command writes files to standard output.

Options

–s *format*
Specifies the format of *inputfile* and invokes an appropriate input converter as part of CDA. The ddif, dtif, dots (for analysis output only) and text converters are provided in the base system kit. Additional converters can be added by the CDA Converter Library and other layered products. Converter Library and other layered products. Contact your system manager for a complete list of the input formats supported on your system. The default format is ddif.

–d *format*
Specifies the format of *outputfile* and invokes an appropriate output converter as part of CDA. The ddif, dtif, text, analysis, and ps converters are provided in the base system kit. Additional converters can be added by the CDA Converter Library and other layered products. Contact your system manager for a complete list of the output formats supported on your system. The default format is ddif.

–O *options_file*
Names the file passed to the input and output converters to control specific processing options for each converter. Refer to your documentation set for a description of converter options.

The options file has a default file type of .cda_options. Each line of the options file specifies a format name that can optionally be followed by _input or _output to restrict the option to either an input or output converter. The second word is a valid option preceded by one or more spaces, tabs, or a slash (/) and can contain upper- and lowercase letters, numbers, dollar signs, and underlines. The case of letters is not significant. If an option requires a value, then spaces, tabs, or an equal sign can separate the option from the value.

Each line can optionally be preceded by spaces and tabs and can be terminated by any character other than those that can be used to specify the format names and options. The syntax and interpretation of the text that follows the format name is specified by the supplier of the front and back end converters for the specified format.

To specify several options for the same input or output format, specify one option on a line. If an invalid option for an input or output format or an invalid value for an option is specified, the option may be ignored or an error message may be returned. Each input or output format that supports processing options specifies any restrictions or special formats required when specifying options.

By default, any messages that occur during processing of the options file are written to the system *standard error location*. For those input and output formats that support a LOG option, messages can be directed to a log file.

-o *outputfile* — Specifies the name of the output file. If not specified, `cdoc` writes to standard output.

See Also

vdoc(1), dxvdoc(1X), DDIF(5), DTIF(5), DOTS(5), CDA(5)

Name

cflow – generate C flow graph

Syntax

cflow [–r] [–ix] [–*i*_] **[–d***num*] *files*

Description

The cflow command 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** options of the C-preprocessor are also understood. Files suffixed with **.s** are assembled and information is extracted from the symbol table. The output of all this non-trivial processing 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, or line, number, followed by a suitable number of tabs indicating the level. Following the reference number is the name of the global, a colon, and the global's definition. (See the i_ option for information on names that begin with an underscore.) For information extracted from C source, the definition consists of an abstract type declaration (for example, **char** *), and, the name of the source file and the line number where the definition was found. The name of the source file and the line number are delimited by angel brackets. Definitions extracted from object files indicate the file name and location counter under which the symbol appeared (for example, *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.

The following is an example in *file.c*:

```
int     i;

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

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

The command

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

produces the following output:

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

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

Options

–d*num* The *num* decimal integer indicates the depth at which the flow graph 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.

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

–ix Includes external and static data symbols. The default is to include only functions in the flow graph.

–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.

Restrictions

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.

Diagnostics

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

See Also

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

Name

checknr – check nroff/troff files

Syntax

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

Description

The `checknr` command 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:

5 Font changes using \fx ... \fP.

Size changes using \sx ... \s0.

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

The `checknr` command knows about the `ms`(7) and `me`(7) macro packages.

The `checknr` command 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 \fx must be terminated with \fP and each \sx must be terminated with \s0. While it will work to directly go into the next font or explicitly specify the original font or point size, and many existing documents actually do this, such a practice will produce 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.

Options

–a Allows additional pairs of macros to be added to the list. 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 –**a**.BS.ES.

–c Defines commands otherwise complained about as undefined.

–f Ignores \f font changes.

–s Ignores \s size changes.

Restrictions

There is no way to define a 1 character macro name using –**a**.
Does not correctly recognize certain reasonable characters, such as conditionals.

Diagnostics

Complaints about unmatched delimiters.

Complaints about unrecognized commands.

Various complaints about the syntax of commands.

See Also

eqn(1), nroff(1), troff(1), ms(7), me(7)

Name

chfn – change system finger entry

Syntax

chfn [*loginname*]

Description

The chfn command is used to change information about users. This information is used by the finger(1) program, among others. It consists of the user's real name, office room number, office phone number, and home phone number. The chfn command 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 <CR>. To enter a blank field, type the word 'none'. This is an example:

```
% chfn
Changing finger information for doe
Name [John Doe]:
Office number [ABC-1/K0]: DEF-2/K1
Office Phone []: 1863
Home Phone [5771546]: none
```

The chfn command allows phone numbers to be entered with or without hyphens. No entries may contain colons, commas, or control characters.

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.

Restrictions

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

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.

If the passwd entry is distributed from another host chfn will not modify it.

Files

/etc/passwd

/etc/ptmp

See Also

chsh(1), finger(1), passwd(1), passwd(5yp)

chgrp(1)

Name

chgrp – change file group

Syntax

chgrp [**–fR**] *group file...*

Description

The `chgrp` command changes the group ID of one or more files or directories. For *file*, you may specify either a full or partial path. For *group*, you may specify either a decimal GID or a group name found in the group file.

The user entering the `chgrp` command must either be the superuser, or be the owner of the file and belong to the specified group.

Options

–f Inhibits display of errors that are returned if `chgrp` fails to change the group identifier of a file.

–R Causes `chgrp` to recursively descend any directories subordinate to *file* and to set the specified group for each file encountered. When symbolic links are encountered, `chgrp` changes the group identifier of the link file but does not traverse the path associated with the link.

Examples

Change group to admin for filea and fileb:

```
chgrp admin filea fileb
```

Files

`/etc/group`

`/etc/passwd`

`/etc/yp/src/group`

`/etc/yp/src/passwd`

See Also

chown(2), group(5), group(5yp), passwd(5), passwd(5yp)

Name

chmod – change file mode

Syntax

chmod [**-fR**] *mode file...*

Description

Permissions on files are set according to *mode* and *file* parameters.

For *file*, you can specify either a full or partial path. You can specify multiple files, separated by spaces.

For *mode*, you specify one of two variants: absolute mode or symbolic mode.

Absolute Mode

For *mode* in absolute form, you specify an octal number constructed from the sum of one or more of the following values:

4000	set user ID on execution (applies to executable files only)
2000	set group ID on execution (applies to executable files only)
1000	set sticky bit (see `chmod`(2) for more information)
0400	read by owner
0200	write by owner
0100	execute, or search if *file* is a directory, by owner
0040	read by group
0020	write by group
0010	execute, or search if *file* is a directory, by group
0004	read by others
0002	write by others
0001	execute, or search if *file* is a directory, by others

For example, the absolute mode value that provides read, write, and execute permission to owner, read and execute permission to group, and read and execute permission to others is 755 (400+200+100+40+10+4+1). The absolute mode value that provides read, write, and execute permission to owner and no permission to group or others is 700 (400+200+100).

Symbolic Mode

To specify *mode* in symbolic form, use the following format:

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

NOTE

Spaces are included in the preceding format so that you can read the arguments; however, as will be shown in examples that follow, you do not enter spaces between mode arguments.

Specify *who* using the letters **u** (for owner), **g** (for group) and **o** (for others) either alone or in combination. You can also specify the letter **a** (for all), which is is equivalent to the letter combination **ugo.** If you omit the *who* parameter, **a** is assumed. For more information, see `umask`(2).

For the *op* parameter, specify the plus sign (+) to add *permission* to the file's mode, the minus sign (–) to remove *permission* from the file's mode, or the equal sign (=) to assign *permission* absolutely (denying or revoking any permission not explicitly specified following the equal sign). The first command in the following example provides group with execute permission for `filea` in addition to any other permissions group currently has for `filea`. The second command limits the permission that group has for `fileb` to execute alone:

```
chmod g+x filea
chmod g=x fileb
```

For the *permission* parameter, specify any combination of the letters **r** (read), **w** (write), **x** (execute), **s** (set owner or group id), and **t** (save text – sticky). Alternatively, you can specify the letter **u**, **g**, or **o** to set *permission* for the *who* parameter to be the same as the permission currently granted to the user category indicated by the letter. In the following example, the group (**g**) is given the same permissions on `filea` as currently granted to owner (**u**):

```
chmod g=u filea
```

You can revoke all permissions by specifying the *who* argument followed by =, and omitting the *permission* argument. For example, the following command removes all permissions from others for `fileb`:

```
chmod o= fileb
```

When specifying more than one symbolic mode for *file*, separate the modes with commas. The mode changes are applied in the sequence specified. In the following example, write permission is added to the permissions already granted to the owner of `filea` and group is then granted the same permissions on `filea` as granted the owner:

```
chmod u+w,g=u filea
```

Options

–f Inhibits display of errors that are returned if `chmod` fails to change the mode on a file.

–R Causes `chmod` to recursively descend any directories subordinate to *file* and to set the specified mode for each file encountered. However, when symbolic links are encountered, `chmod` does not change the mode of the link file and does not traverse the path associated with the link. Note that the `-R` option is useful only when *file* identifies a directory that is not empty.

Restrictions

The *permission* letter **s** is used only with *who* letter **u** or **g**.

Only the owner of a file or someone logged on as superuser may change the mode of that file.

Examples

Using absolute mode, provide read, write, and search permission to the owner, and read and search permission to others for a directory named `public`:

```
chmod 755 ~harris/public
```

Using absolute mode, set the UID for `progrmb` execution to be the UID of of the file owner rather than the UID of the user running the program as follows:

```
chmod 4000 progrmb
```

Using symbolic mode, perform the same operation as described for the preceding example:

```
chmod u=s progrmb
```

Using symbolic mode, deny write permission to others for the file `ourspec`:

```
chmod o-w ourspec
```

Using symbolic mode, give execute permission on file `myprog` to all user categories:

```
chmod +x myprog
```

Using symbolic mode, give write permission to all group members, deny write permission to others, and give search permission to owner on `docdir`:

```
chmod g+w,o-r,u+x docdir
```

Using symbolic mode, give read and execute permissions to others for a directory named `programs`, and then recursively descend the paths subordinate to `programs`, adding the same permissions for others on all files and directories included in the subordinate paths:

```
chmod -R o+rx programs
```

NOTE

In the preceding example, if `programs` were the name of a file rather than a directory, `chmod` would change the mode only of the `programs` file.

See Also

ls(1), chmod(2), stat(2), umask(2), chown(8)

chsh (1)

Name

chsh – change login shell

Syntax

chsh [*loginname*]

Description

The chsh command is a command similar to passwd, except that it is used to change the login shell field of the password file rather than the password entry. The program will prompt you for a new shell. The shell name supplied must match one of the entries in /etc/shells. If no name is given the shell will be unchanged and the diagnostic "Login shell unchanged" will be printed.

An example use of this command is:

```
% chsh
Changing login shell for bill
Shell [/bin/csh]: sh
```

Restrictions

Both the new shell and the old shell must be found in /etc/shells to be able to change the shell.

If the passwd entry is distributed from another host chsh will not modify it.

Files

/etc/shells

See Also

chfn(1), passwd(1), yppasswd(1yp), passwd(5yp)

Name

clear – clear terminal screen

Syntax

clear

Description

The `clear` command clears your screen if this is possible. It looks in the environment for the terminal type and then in `/etc/termcap` to figure out how to clear the screen.

Files

/etc/termcap terminal capability data base

Name

cmp – compare two files

Syntax

cmp [–l | –s] *file1 file2* [*skip1*] [*skip2*]

Description

The cmp command compares two files. If either *file1* or *file2* is '–', standard input is used for the file. With no options, cmp makes no comment if the files are the same. If they differ, it reports the byte and line number at which the difference occurred to standard output. If one file is an initial subsequence of the other a message including the file name is written to standard error.

The optional *skip1* and *skip2* parameters are initial byte offsets into *file1* and *file2* respectively and may be either octal, by specifying a leading 0, or decimal. When using *skip1* and *skip2* the offset is treated as the start of the respective input file. Only one option may be specified at a time. Only one of the input files may be standard input at a time. Because the line number is not calculated when using either of the options the use of either flag will increase the speed of cmp.

Options

–l Long format: prints the byte number (decimal) and the differing bytes (octal) for each difference.

–s Suppresses normal output and sets the exit code only.

Diagnostics

Exit code 0 is returned for identical files, 1 for different files, and 2 for an inaccessible or missing argument.

See Also

comm(1), diff(1)

Name

col – filter reverse line feeds

Syntax

col [–*options*]

Description

The `col` command reads the standard input and writes the standard output. It performs the line overlays implied by reverse line feeds (ESC-7 in ASCII) and by forward and reverse half line feeds (ESC-9 and ESC-8, respectively). The `col` command is particularly useful for filtering multicolumn output made with the `.rt` command of `nroff`, and for filtering output resulting from the `tbl` preprocessor.

Although `col` accepts half line motions in its input, it does not normally output them. Instead, text that would appear between lines is moved to the next lower full line boundary.

The control characters SO (ASCII code 017) and SI (ASCII code 016) are assumed to start and end text in an alternate character set. The character set (primary or alternate) associated with each printing character read is remembered. On output, SO and SI characters are generated where necessary to maintain the correct treatment of each character.

The `col` command normally converts white space to tabs to shorten printing time. If the **–h** option is given, this conversion is suppressed.

On input, the only control characters accepted are <space>, <backspace>, <tab>, <return>, <newline>, etc... The VT character is an alternate form of full reverse linefeed, included for compatibility with earlier programs of this type. All other non-printing characters are ignored.

Options

–b Assumes that the output device does not have backspacing.

–f Suppresses moving half lines to the next full line.

–h Suppresses conversion of white space to tabs.

–p Forces through unchanged any unknown escape sequences that are found in its input. This option should be used with care.

–x Suppresses conversion of white space to tabs (same as –h).

Restrictions

Cannot back up more than 128 lines.
No more than 800 characters, including backspaces, on a line.

See Also

tbl(1), nroff(1)

colcrt(1)

Name

colcrt – filter nroff output for CRT previewing

Syntax

colcrt [–] [–**2**] [*file...*]

Description

The `colcrt` command 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 new lines in between the normal output lines.

Options

– Suppresses all underlining. It is especially useful for previewing *allboxed* tables from `tbl`(1).

–2 Causes half-lines to be printed, 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** option is useful for sending output to the line printer when the output contains superscripts and subscripts which would otherwise be invisible.

Examples

A typical use of `colcrt` would be:

```
tbl exum2.n | nroff -ms | colcrt - | more
```

Restrictions

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.

See Also

col(1), more(1), nroff(1), ul(1)

Name

colrm – remove columns from a file

Syntax

colrm [*startcol* [*endcol*]]

Description

The `colrm` command removes selected columns from a file. Input is taken from standard input. Output is sent to standard output.

If called with one parameter the columns of each line will be removed starting with the specified column. If called with two parameters the columns from the first column to the last column will be removed.

Column numbering starts with column 1.

See Also

expand(1)

Name

comb – combine delta versions of SCCS file

Syntax

comb [**–o**] [**–s**] [**–p***sid*] [**–c***list*] *files*

Description

The comb command generates a shell procedure which, when run, will reconstruct the given SCCS files. For further information, see sh(1). The reconstructed files are generally 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 non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files, which 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.

The generated shell procedure is written on the standard output.

Each keyletter argument 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.

Options

–c*list* Preserves specified deltas. See get(1) for the syntax of a *list*. All other deltas are discarded.

–o 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. 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.

–p*SID* Indicates oldest delta to be preserved. All older deltas are discarded in the reconstructed file.

–s Generates a shell procedure which produces a report. This report gives the file name, 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 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.

Restrictions

The comb command may rearrange the shape of the tree of deltas. It may not save any space; in fact, it is possible for the reconstructed file to actually be larger than the original.

Diagnostics

See `sccshelp(1)` for explanations.

Files

s.COMB	The name of the reconstructed SCCS file.
comb?????	Temporary.

See Also

admin(1), delta(1), get(1), help(1), prs(1), sccs(1), sccsfile(5)
The Guide to the Source Code Control System

comm(1)

Name

comm – compare sorted data

Syntax

comm [– [**123**]] *file1 file2*

Description

The comm command reads *file1* and *file2,* which should be ordered in ASCII collating sequence, 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.

Options

1 Suppresses column one: lines in file1 only.

2 Suppresses column two: lines in file2 only.

3 Suppresses column three: lines in file1 and file2.

Thus comm **–12** prints only the lines common to the two files. And comm **–23** prints only lines in the first file but not in the second. Finally, comm **–123** is not an option.

See Also

cmp(1), diff(1), diff3(1), diffmk(1), join(1), uniq(1)

Name

comp – compose a message

Syntax

comp [*+folder*] [*msg*] [**–draftfolder** *+folder*] [**–draftmessage** *msg*] [**–nodraftfolder**] [**–editor** *editorname*] [**–noedit**] [**–file** *filename*] [**–form** *formfile*] [**–use**] [**–nouse**] [**–whatnowproc** *program*] [**–nowhatnowproc**] [**–help**]

Description

Use comp to create a new message for mailing. When you run comp, it provides a message template for you to fill in and invokes an editor so that you can complete the message.

A mail message consists of a mail header and the body of the message. The mail header contains all the information that determines who is going to receive the mail message. It can also give the recipients some information about the sender. The body of the message is the actual text of the message that you want to send. The header is separated from the body of the text by a blank line or by a line of dashes. The header must be separated from the body of the message in this way for the message to be identified properly when it is sent (see send(1mh)).

The standard message header contains the following elements:

```
To:
cc:
Subject:
--------
```

Options

You can specify an alternative mail header by setting up a file called components in your MH directory. This is used instead of the default mail header by MH. You can also direct comp to use an alternative header by using the -form *formfile* option.

comp normally invokes an editor, unless you have used the -noedit flag. The default editor is prompter, which is a very rudimentary editor (see prompter(1mh)). You can specify your own choice of editor using the -editor *editorname* option. If you regularly use the same editor you can specify it by specifying it in the editor: line of your mh-profile. The following example shows how to set up vi as the editor you'll use to compose mail messages.

```
editor: /usr/ucb/vi
```

You can direct comp to use an existing message by specifying a folder or a *msg* argument. You can not supply both a **–form** *formfile* and a *+folder* or a *msg* argument.

If you supply a *+folder* argument, comp will use the current message in the specified folder as the draft for your message. If you specify a message number as an argument and you do not have a drafts folder set up, comp will use that message from the current folder. If you do have a drafts folder set up, comp will use the specified message from your drafts folder. This is similar to specifying comp–use, except that comp–use will only take messages from the draft or

drafts folders. The draft file or drafts folder are used by the comp, dist, repl, and forw commands. If any of these commands are terminated without sending the draft, you can edit the draft again by using **comp** *–use*.

If the draft already exists, comp will ask you what you want to do with the draft. The available options are:

quit	aborts comp leaving the draft intact
replace	replaces the existing draft with the appropriate message form
use	allows you to edit the existing draft
list	displays the draft message
refile	refiles the existing draft message in the specified folder and provides a new message form for you to complete

You have to specify a *+foldername* when you specify refile. If you use quit–d you will exit from the editor and delete the draft message. The *+foldername* argument to refile is required.

The -draftfolder *+folder* and -draftmessage *msg* switches invoke the MH draft folder facility. The -draftfolder *+foldername* switch lets you specify the folder that an unsent draft will be stored in. The -draftmessage *msg* switch lets you create a message with a meaningful name. If you quit without sending a message, that message will be stored, as a file with the specified name, in your Mail directory. You can use dist, forw, and send to access the specified draft file.

The -file *filename* switch makes comp use the named file as the message draft.

When you exit from the editor, comp invokes the whatnow program. See whatnow(1mh) for a discussion of available options. You can also specify your own whatnow process using the -whatnowproc *program.* If you do specify your own whatnow program, you should not call it whatnow. You can suppress the whatnow program entirely by using the -nowhatnowproc switch. However, as the whatnow program normally starts the initial edit, -nowhatnowproc will prevent you from editing the message.

The defaults for comp are:

+foldername defaults to the current folder
msg defaults to the current message
-nodraftfolder
-nouse

Files

/usr/new/lib/mh/components	The message skeleton
<mh-dir>/components	Alternative to the standard skeleton
$HOME/.mh_profile	Your user profile
<mh-dir>/draft	The draft file

Profile Components

Path:	To determine your MH directory
Draft-Folder:	To find the default draft–folder
Editor:	To override the default editor
Msg-Protect:	To set mode when creating a new message (draft)
fileproc:	Program to refile the message
whatnowproc:	Program to ask the ''What now?'' questions

See Also

dist(1mh), forw(1mh), prompter(1mh), repl(1mh), send(1mh), whatnow(1mh), mh-profile(5mh)

Name

compact, uncompact, ccat – compress and uncompress files, and cat them

Syntax

compact [*name...*]
uncompact [*name...*]
ccat [*file...*]

Description

The `compact` command compresses the named files using an adaptive Huffman code. If no file names are given, the standard input is compacted to the standard output. The `compact` command 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 prepend 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,

```
... | 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.

The `uncompact` command restores the original file from a file compressed by `compact`. If no file names are given, the standard input is uncompacted to the standard output.

The `ccat` command cats the original file from a file compressed by `compact`, without uncompressing the file.

The `compact` command is present only for compatibility. In general, the `compress`(1) command runs faster and gives better compression.

Restrictions

The last segment of the file name must contain fewer than thirteen characters to allow space for the appended '.C'.

Files

`*.C` compacted file created by compact, removed by uncompact

See Also

compress(1)

Name

compress, uncompress, zcat – compress and expand data

Syntax

compress [**–f**] [**–v**] [**–c**] [**–b** *bits*] [*name* ...]
uncompress [**–f**] [**–v**] [**–c**] [*name* ...]
zcat [*name* ...]

Description

The `compress` command 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`.

The **–f** option will force compression of *name*, even if it does not actually shrink *name*, or if the corresponding *name* .Z file already exists. If the **–f** option is omitted, the user is asked whether an existing *name*.Z file should be overwritten (unless `compress` is run in the background under `/bin/sh`).

The **–c** (cat) option makes *compress/uncompress* write to the standard output without changing any files. Neither `uncompress` **–c** nor `zcat` alter files.

The `compress` command uses the modified Lempel-Ziv algorithm. 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 *bits* 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 *bits* limit is attained, `compress` periodically checks the compression ratio. If the ratio 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.

Note that the **–b** flag is omitted for `uncompress`, since the *bits* parameter specified during compression is encoded within the output along with a number that ensures that neither decompression of random data nor recompression of compressed data is attempted.

How much each file is compressed depends on the size of the input, the number of *bits* per code, and the distribution of common substrings. Typically, text such as source code or English is reduced by 50–60%. Compression is generally much better than that achieved by Huffman coding or adaptive Huffman coding, and takes less time to compute.

The **–v** option displays the percent reduction of each file.

If an error occurs, exit status is 1. However, if the last file was not compressed because it became larger, the status is 2. Otherwise, the status is 0.

Options

–f Forces compression of *name*.

–c Makes *compress/uncompress* write to the standard output.

–b Specifies the allowable *bits* limit. The default is 16.

–v Displays the percent reduction of each file.

Diagnostics

Usage: compress [–fvc] [–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
: The *file* was compressed by a program that could deal with more *bits* than the compress code on this machine. Recompress the file with smaller *bits*.

file: already has .Z suffix -- no change
: The file is assumed to be compressed already. Rename the file and try again.

file already exists; do you wish to overwrite (y or n)?
: Type y if you want the output file to be replaced; type n if you do not.

uncompress: corrupt input
: A SIGSEGV violation was detected which usually means that the input file is corrupted.

Compression: *xx.xx%*
: Percent of the input saved by compression. (For the **–v** option only.)

-- not a regular file: unchanged
: If the input file is not a regular file (for example, a directory), it remains unchanged.

-- 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 unchanged.

Restrictions

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 (64KB or less).

Name

compress, uncompress, zcat – compress and expand data

Syntax

compress [*options*] [*name* ...]
uncompress [*options*] [*name* ...]
zcat [*name* ...]

Description

The compress command 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.

The compress command uses the modified Lempel-Ziv algorithm. 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 *bits* 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 *bits* limit is attained, compress periodically checks the compression ratio. If the ratio 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.

How much each file is compressed depends on the size of the input, the number of *bits* per code, and the distribution of common substrings. Typically, text such as source code or English is reduced by 50–60%. Compression is generally much better than that achieved by Huffman coding or adaptive Huffman coding, and takes less time to compute.

If an error occurs, exit status is 1. However, if the last file was not compressed because it became larger, the status is 2. Otherwise, the status is 0.

Options

–b The **–b** flag is omitted for uncompress, since the *bits* parameter specified during compression is encoded within the output along with a number that ensures that neither decompression of random data nor recompression of compressed data is attempted.

–c The cat option. Makes *compress/uncompress* write to the standard output without changing any files. Neither uncompress **–c** or zcat alter files.

–f Forces compression of *name* , even if it does not actually shrink *name* , or if the corresponding *name* .Z file already exists. If the **–f** option is omitted, the user is asked whether an existing *name* .Z file should be overwritten unless compress is run in the background under /bin/sh.

–q Quiet, not as verbose.

–v Displays the percent reduction of each file.

–V Prints version and options.

Diagnostics

Usage: compress [–fvc] [–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
: The *file* was compressed by a program that could deal with more *bits* than the compress code on this machine. Recompress the file with smaller *bits*.

file: already has .Z suffix -- no change
: The file is assumed to be compressed already. 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)?
: Type y if you want the output file to be replaced; type n if you do not.

uncompress: corrupt input
: A SIGSEGV violation was detected which usually means that the input file is corrupted.

Compression: *xx.xx%*
: Percent of the input saved by compression. (For the **–v** option only.)

-- not a regular file: unchanged
: If the input file is not a regular file (for example, a directory), it remains unchanged.

-- 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 unchanged.

Restrictions

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 (64KB or less).

Name

cord – rearranges procedures in an executable to facilitate better cache mapping.

Syntax

cord [**–c** *cachesize*] [**–f**] [**–o** *outfile*] [**–p** *maxphases*] [**–v**] *obj reorder*

Description

The `cord` command rearranges procedures in an executable object to maximize efficiency in a machine's cache. By rearranging the procedures properly, the instruction cache miss rate is reduced. The `cord` command does not attempt to determine the correct ordering, but is given a *reorder* file containing the desired procedure order. The *reorder* file is generated by the `ftoc` program which in turn generates a *reorder* file from a set of profile feedback files (see `prof(1)`).

Processed lines in the *reorder* file are called procedure lines. Each procedure line must be on a separate source line. Each procedure line must contain the source name of the file, followed by a blank followed by a qualified procedure name (nested procedures need to be qualified x.y where x is the outer procedure). A newline or blank can follow the procedure name:

```
foo.c bar >>i ignore this stuff<<
```

Lines beginning with a pound sign (#) are comments. Lines beginning with a dollar sign ($) are considered `cord` directive lines. The only directive currently understood is **$phase**. This directive will consider the rest of the file (until the end of file or next **$phase**) as a new phase of the program and will order the procedures accordingly. Procedures may appear in more than one phase, resulting in more than one copy of it in the final binary. The `cord` command will try to relocate references to a procedure to a copy in the requesting phase's list of procedures first and then a random copy if one is not found.

You should use the **–cord** option to a compiler driver like `cc` rather than execute cord directly. Options to `cord` can be specified with **–Wz,***cordarg0,cordarg1,*.... If you have to run `cord` manually, you should run it once with the driver using the **–v** flag on a simple program to see the exact passes and their arguments involved in using `cord`.

The *obj* argument is an executable object with its relocation information intact. This can be achieved by passing the **–r –z –d** options to the linker, `ld`. The **–r** linker option maintains relocation information in the object, but will not make it a ZMAGIC file (hence **–z**) nor will it allocate common variables (hence **–d**) as it would without the option.

Options

–c *cachesize* Specify the cachesize of the machine you want to execute on in bytes. This only affects the **–f** option. If not specified 65536 is used.

–f Flip the first cachepage size procedures. The assumption when `cord` was written was that procedures would be reordered by procedure density (cycles/byte). This option ensures that the densest part of each page following the first cachepage would conflict with least dense part of the first cachepage.

–o *outputfile* specifies the output file. If not specified, a.out is used.

–p *phasemax* Specifies the maximum number of phases allowed. The default is 20.

–v Prints verbose information. This includes listing those procedures considered part of other procedures and cannot be rearranged (these are basically assembler procedures that may contain relative branches to other procedures rather than relocatable ones). The listing also list those procedures in the flipped area (if any) and a mapping of old location to new.

Restrictions

Since `cord` works from an input list of procedures generated from profile output, the resulting binary is data dependent. In other words, it may only perform well on the same input data that generated the profile information and may perform worse than the original binary on other data. Furthermore, if the hot areas in the cache don't fit well into one cachepage, performance can degrade.

See Also

cc(1), ftoc(1), ld(1), prof(1)

Name

cord2 – rearranges basic blocks in an executable file to facilitate better cache mapping.

Syntax

cord2 [**-v**] [**-o** *outfile]* [**-c** *cachewords*] [**-d**] [**-b** *bridge_limit*] [**-n**] [**-A** *addersfile*] [[**-C** *countsfile*] **...**] *obj*

Description

The **cord2** command extracts basic blocks from a program and deposits them in a new area in the text, making jumps to and from that area as necessary. By separating the basic blocks, you can reduce instruction cache miss rates. The **cord2** command takes the output of a pixie profiling run as input (see `pixie(1)`).

The executable object file has the suffix *obj*. The **cord2** command only requires one *addersfile*; it creates the filename by appending *.Bbaddrs* to the *obj* filename if none is specified with **-A**. Multiple counts files can be specified from many runs with multiple **-C** arguments. If none are specified, **cord2** creates the counts filename by appending *.Counts* to the *obj* name.

Multiple counts files are added together into an internal counts array represented with C double-type elements. The counts array elements contain the density of a block or cycles/byte. If you specify **-n**, then the counts are normalized so that each counts array entry is cycles/totalcycles. When one counts file is specified, the default is to favor small blocks; **-n** negates that. When many counts files are specified, **-n** also negates favoring one counts file. This is because its totalcycles may exceed the totalcycles of another counts file.

The **cord2** command determines which basic blocks to insert by sorting the counts array and collecting the blocks with the highest counts that can fit into the new area. The **cord2** command may skip over huge blocks that do not fit at the end of the new area.

Once the blocks are determined, they are inserted into the new area, and their original location is modified to jump to the new area. At the end of each block in the new area, a jump is added back to the original block's subsequent or fall-through location, and the branch/jump target (if necessary). Both entering and exiting the new area is optimized to take advantage of other blocks in the new area and jump delay slots.

Often, there may be one or more fall-through blocks of a block in the new area which are small, hardly ever used, and not in the new area. If the block following these fall-through blocks is in the new area, the fall-through blocks are called bridge blocks. It may be more costly to generate jumps to and from bridge blocks rather than to simply copy them.

The **cord2** command allows you to specify that bridge blocks be added to the new area if they total less than the *bridge_limit* instructions between two new-area blocks. You can specify the *bridge_limit* with **-b**; the default is zero. Bridge blocks can bump blocks out of the new area that might normally fit into it.

NOTE

Because the **cord2** command works from profile output, the resulting binary is data dependent. In other words, it may perform well only on the same input data that generated the profile information, and may perform worse than the original binary on other data. Furthermore, if the hot areas in the cache do not fit well into one cachepage, performance can degrade.

Options

The **cord2** command also accepts these options:

-d Fill the delay slots with nops only when adding jumps to and from the new area.

-v Print verbose information. This includes statistics about the **cord2** process.

-v -v Print all of the **-v** information, but include detailed disassemblies of the code moved, changed, and generated by **cord2.**

-c cachewords
Specify the number of words in the cache of the machine on which you want to execute. This is actually the size of the new area. The *cachesize* may be a misnomer, as you can specify a size other than your machine's cache size; however, it is probably the correct number.

-o outputfile
Specify the output file. If it is not specified, the default is *a.out.cord2*.

Restrictions

The **cord2** command adds the new area to the end of text so any program using the *etext* symbol may not work. See `ld`(1).

See Also

pixie(1), cord(1)

Name

cp – copy file data

Syntax

cp [–f] [–i] [–p] *file1 file2*

cp [–f] [–i] [–p] [–r] *file... directory*

cp [–f] [–i] [–p] [–r] *directory... directory*

Description

The cp command copies *file1* onto *file2*. The mode and owner of *file2* are preserved if it already existed; the mode of *file1* is used otherwise. Note that the cp command will not copy a file onto itself.

In the second form, one or more *files* are copied into the *directory* with their original file names.

In the third form, one or more source *directories* are copied into the destination *directory* with their original file names.

Options

–f Forces existing destination pathnames to be removed before copying, without prompting for confirmation. The **–i** option is ignored if the **–f** option is specified.

–i Prompts user with the name of file whenever the copy will cause an old file to be overwritten. A yes answer will cause cp to continue. Any other answer will prevent it from overwriting the file.

–p Preserves (duplicates) in the copies the modification time, access time, file mode, user ID, and group ID as allowed by the permissions of the source files, ignoring the present umask.

–r Copies directories. Entire directory trees, including their subtrees and the individual files they contain, are copied to the specified destination directory. The directory, its subtrees, and the individual files retain their original names. For example, to copy the directory reports, including all of its subtrees and files, into the directory news, enter the following command:

```
cp -r reports news
```

See Also

cat(1), pr(1), mv(1)

Name

cpio – copy file archives in and out

Syntax

cpio –o [*keys*]

cpio –i [*keys*] [*patterns*]

cpio –p [*keys*] *directory*

Description

The cpio command is a filter designed to let you copy files to or from an archive. The cpio command differs from the ar command in that cpio lets you archive any kind of file, while ar is limited to program object files.

Options

–i Copies files that match the specified pattern. If the pattern is not specified, copies in all files. Extracts files from the standard input, which is assumed to be the product of a previous **cpio –o**, and places them into the user's current directory tree. For files with the same name, the newer file replaces the older file unless the **–u** option is used.

Only files with names that match *patterns* are selected. The *patterns* are specified using the notation for names described in sh(1). In *patterns,* the slash for directories (/) is included in searches using meta-characters. For example, suppose the archive contains the file filep and the pathname information in the archive indicates that the directory below contains the file file2p. This command copies both files into the user's current directory:

```
cpio -i *p < /dev/rmt01
```

Multiple *patterns* may be specified and if no *patterns* are specified, the default for *patterns* is * (that is, select all files). The extracted files are conditionally created and copied into the current directory tree based upon the options described below. The cpio command has three function keys, each with its own set of options.

–o Copies out the specified files. Reads the standard input to obtain a list of path names and copies those files onto the standard output together with path name and status information.

–p Copies files into the specified destination directory, which must already exist. Reads the standard input to obtain a list of path names of files that are conditionally created. This list of files is copied into the destination *directory* tree based upon the options used. For files with the same name, the newer file replaces the older file unless the **–u** option is used.

Keys

6 Processes a file with the UNIX System Sixth Edition format.

a Retains original access times of input files, and can be used with **–o** and **–p.** Normally, the read(s) used in the copy update the copied file's access time.

B Determines input/output is to be blocked 5,120 bytes to the record. This option is meaningful only with data directed to or from `/dev/rmt?h` or `/dev/rmt?l`.

b Swaps both bytes and halfwords.

c Creates header information in ASCII format and can be used with **–i** and **–o.**

d Creates subdirectories, as needed, below the specified destination directory.

f Copies all files except those that match the specified pattern.

k Enables symbolic link handling and is used with the **–i, –o,** and **–p** options.

l Creates links wherever possible.

m Retains modification time for each copied file. This option does not work on directories or symbolic links that are being copied; the directory is always reset to show the access time when the copy was made.

r Interactively renames files. If you respond with a null line, the file is skipped (not copied). Use only with the **–i** option.

s Swaps bytes while copying files in.

S Swaps half words while copying files in.

t Prints a table of contents of the input (no files are created).

u Copies files unconditionally. (Otherwise, an older file will not replace a newer file with the same name).

v Displays detailed (verbose) information as it copies and/or creates file. When used with the **t** option, the table of contents looks like the output of an **ls –l** command. For further information, see `ls`(1).

Examples

This example shows how to copy the contents of the user's current directory into an archive.

```
ls | cpio -o  > /dev/rmt0l
```

This example shows how to duplicate a directory hierarchy.

```
mkdir ~phares/newdir
cd ~phares/olddir
find . -print | cpio -pdl ~phares/newdir
```

This example shows how to copy all files and directories with names containing the characters "chapter" in user smith's home directory and underlying directories.

```
find ~smith -name '*chapter*' -print | cpio -o > /dev/rmt0h
```

This example shows the results of using the **r** option with the **–i** function key.

```
ls | cpio -ir > ~smith/newdir
Rename <file1>
```

```
newnamefile1
Rename <file2>
<RETURN>
Skipped
Rename <file3>
newnamefile3
```

In some cases, the **–cpio** option of the `find` command can be used more effectively than pipes and redirects using `cpio`. For instance, the following example

```
find . -print | cpio -oB > /dev/rmt01
```

can be handled more efficiently by:

```
find . -cpio /dev/rmt01
```

To copy the contents of a directory (with symbolic link handling enabled) to the tape drive, type:

```
ls | cpio -ok > /dev/rmt0h
```

To restore the archived files back into a directory, type:

```
cpio -ik < /dev/rmt0h
```

The following example moves files, including symbolic links, from an old directory to a new directory:

```
mkdir ~craig/newdir
cd ~craig/olddir
ls | cpio -pdk ~craig/newdir
```

Restrictions

Pathnames are restricted to 128 characters.

When there are too many unique linked files, the program runs out of memory and cannot trace them. In this case, linking information is lost.

Only the superuser can copy special files.

See Also

ar(1), find(1), cpio(5)

Name

cpp – the C language preprocessor

Syntax

/lib/cpp [*option* ...] [*ifile* [*ofile*]]

Description

The cpp command is the C language preprocessor which is invoked as the first pass of any C compilation using the cc(1) command. Thus, the output of cpp is designed to be in a form acceptable as input to the next pass of the C compiler.

The preferred way to invoke cpp, however, is through the cc(1) command. See m4(1) for a general macro processor.

Arguments

The cpp command optionally accepts two file names as arguments. The *ifile* and *ofile* are, respectively, the input and output for the preprocessor. They default to standard input and standard output if no argument is supplied.

Options

–B Strips C++-style comments (begin with // and end with newline).

–C Passes along all comments, except those found on cpp directive lines. By default, cpp strips C-style comments.

–M Generates dependency lists suitable for use with make(1) instead of the normal output.

–P Preprocesses the input without producing the line control information used by the next pass of the C compiler.

–R Permits recursion when a macro is expanded.

–U*name* Removes any initial definition of *name*, where *name* is a reserved symbol that is predefined by the preprocessor. The symbols predefined by this implementation are bsd4_2, ultrix, unix, mips, host_mips, and MIPSEL.

–D*name*
–D*name*=*def* Defines *name* as if by a **#define** directive. If no =*def* is given, *name* is defined as 1. The **–D** option has lower precedence than the **–U** option. That is, if the same name is used in both a **–U** option and a **–D** option, the name remains undefined regardless of the order of the options.

–I*dir* Changes the algorithm for searching for **#include** files whose names do not begin with a slash (/) to look in *dir* before looking in the directories on the standard list. Thus, **#include** files whose names are enclosed in quotes (" ") will be searched for first in the directory of the file with the **#include** line, then in directories named in **–I** options, and,

finally, in directories on a standard list. For **#include** files whose names are enclosed in angle brackets (<>), the directory of the file with the **#include** line is not searched.

Directives

All cpp directives start with lines that begin with a pound sign (#). Any number of blanks and tabs are allowed between the pound signs and the directive. The following is a list of the directives:

#define *name*(*arg*, ...,*arg*) *token-string*
: Replaces subsequent instances of *name* and the following set of tokens that is enclosed in parentheses by *token-string*. Each occurrence of an *arg* in the *token-string* is replaced by the corresponding set of tokens in the comma-separated list. Note that spaces between *name* and the left parenthesis (() are not allowed. When a macro with arguments is expanded, the arguments are placed unchanged into the expanded *token-string* . After the entire *token-string* has been expanded, cpp re-starts its scan for names to expand at the beginning of the newly created *token-string*.

#undef *name*
: Causes the definition of *name* (if any) to be forgotten.

#include *"filename"*
#include *<filename>*
: Includes the contents of *filename* , which will then be run through cpp. When the *<filename>* notation is used, *filename* is searched for in the standard places. See the **–I** option above for more detail.

#line *integer-constant "filename"*
: Causes cpp to generate line control information for the next pass of the C compiler. *Integer-constant* is the line number of the next line and *filename* is the file that it comes from. If *"filename"* is not given, the current file name is unchanged.

#endif
: Ends a section of lines begun by a test directive (**#if, #ifdef,** or **#ifndef**). Each test directive must have a matching **#endif.**

#ifdef *name*
: Defines text that will appear in the output if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**.

#ifndef *name*
: Defines text that will not appear in the output if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**.

#if *constant-expression*
: Defines text that will appear in the output if *constant-expression* is not zero. All binary non-assignment C operators, which include the **?:** , en dash (–), exclamation mark (**!**), and tilde (**~**) 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**, which can be used in *constant-expression* in these two forms: **defined**

(*name*) or **defined** *name*. This allows the utility of **#ifdef** and **#ifndef** in a **#if** directive. Only these operators, integer constants, and names which are known by cpp should be used in *constant-expression*. In particular, the **sizeof** operator is not available.

#else
Reverses the notion of the test directive which matches this directive. So if lines prior to this directive are ignored, the following lines will appear in the output. The reverse is also true.

#elif *constant-expression*
Defines text that will appear in the output if the preceding test directive and all intervening **#elif** directives equalled zero and the *constant-expression* did not equal zero. The rules for *constant-expression* are the same as for the **#if** directive.

The test directives and the possible **#else** and **#elif** directives can be nested.

In addition to these directives, the System V **#ident** directive is recognized and ignored.

Two special names are understood by cpp: **__LINE__** is defined as the current line number (as a decimal integer) and **__FILE__** is defined as the current file name (as a C string). They can be used in any situations where you would use other defined names, including in macros.

Diagnostics

The error messages produced by cpp are self-explanatory. The line number and filename where the error occurred are printed along with the diagnostic.

Files

/usr/include standard directory for **#include** files

See Also

cc(1), m4(1)

Name

cpp – the C language preprocessor

Syntax

/lib/cpp [*option* ...] [*ifile* [*ofile*]]

Description

The cpp command is the C language preprocessor which is invoked as the first pass of any C compilation using the cc(1) command. Thus, the output of cpp is designed to be in a form acceptable as input to the next pass of the C compiler.

The preferred way to invoke cpp, however, is through the cc(1) command. See m4(1) for a general macro processor.

Arguments

The cpp command optionally accepts two file names as arguments. The *ifile* and *ofile* are, respectively, the input and output for the preprocessor. They default to standard input and standard output if no argument is supplied.

Options

–B Strips C++-style comments (begin with // and end with newline).

–C Passes along all comments, except those found on cpp directive lines. By default, cpp strips C-style comments.

–M Generates dependency lists suitable for use with make(1) instead of the normal output.

–P Preprocesses the input without producing the line control information used by the next pass of the C compiler.

–R Permits recursion when a macro is expanded.

–U*name* Removes any initial definition of *name*, where *name* is a reserved symbol that is predefined by the preprocessor. The symbols predefined by this implementation are bsd4_2, ultrix, unix, and vax.

–D*name*
–D*name=def* Defines *name* as if by a **#define** directive. If no *=def* is given, *name* is defined as 1. The **–D** option has lower precedence than the **–U** option. That is, if the same name is used in both a **–U** option and a **–D** option, the name remains undefined regardless of the order of the options.

–I*dir* Changes the algorithm for searching for **#include** files whose names do not begin with a backslash (/) to look in *dir* before looking in the directories on the standard list. Thus, **#include** files whose names are enclosed in quotes (" ") will be searched for first in the directory of the file with the **#include** line, then in directories named in **–I** options, and,

finally, in directories on a standard list. For **#include** files whose names are enclosed in braces (<>), the directory of the file with the **#include** line is not searched.

Directives

All cpp directives start with lines that begin with a pound sign (#). Any number of blanks and tabs are allowed between the pound signs and the directive. The following is a list of the directives:

#define *name(arg, ...,arg) token-string*
: Replaces subsequent instances of *name* and the following set of tokens that is enclosed in parentheses by *token-string*. Each occurrence of an *arg* in the *token-string* is replaced by the corresponding set of tokens in the comma-separated list. Note that spaces between *name* and the left parenthesis (() are not allowed. When a macro with arguments is expanded, the arguments are placed unchanged into the expanded *token-string* . After the entire *token-string* has been expanded, cpp re-starts its scan for names to expand at the beginning of the newly created *token-string*.

#undef *name*
: Causes the definition of *name* (if any) to be forgotten.

#include *"filename"*
#include *<filename>*
: Includes the contents of *filename* , which will then be run through cpp. When the *<filename>* notation is used, *filename* is searched for in the standard places. See the **–I** option above for more detail.

#line *integer-constant "filename"*
: Causes cpp to generate line control information for the next pass of the C compiler. *Integer-constant* is the line number of the next line and *filename* is the file that it comes from. If *"filename"* is not given, the current file name is unchanged.

#endif
: Ends a section of lines begun by a test directive (**#if**, **#ifdef**, or **#ifndef**). Each test directive must have a matching **#endif.**

#ifdef *name*
: Defines text that will appear in the output if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**.

#ifndef *name*
: Defines text that will not appear in the output if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**.

#if *constant-expression*
: Defines text that will appear in the output if *constant-expression* is not zero. All binary non-assignment C operators, which include the **?:** , minus sign (–), exclamation mark (**!**), and tilde (**~**) 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**, which can be used in *constant-expression* in these two forms: **defined**

(*name*) or **defined** *name*. This allows the utility of **#ifdef** and **#ifndef** in a **#if** directive. Only these operators, integer constants, and names which are known by cpp should be used in *constant-expression*. In particular, the **sizeof** operator is not available.

#else Reverses the notion of the test directive which matches this directive. So if lines prior to this directive are ignored, the following lines will appear in the output. The reverse is also true.

#elif *constant-expression*
Defines text that will appear in the output if the preceding test directive and all intervening **#elif** directives equalled zero and the *constant-expression* did not equal zero. The rules for *constant-expression* are the same as for the **#if** directive.

The test directives and the possible **#else** and **#elif** directives can be nested.

In addition to these directives, the System V **#ident** directive is recognized and ignored.

Two special names are understood by cpp: **__LINE__** is defined as the current line number (as a decimal integer) and **__FILE__** is defined as the current file name (as a C string). They can be used in any situations where you would use other defined names, including in macros.

Diagnostics

The error messages produced by cpp are self-explanatory. The line number and filename where the error occurred are printed along with the diagnostic.

Files

/usr/include standard directory for **#include** files

See Also

cc(1), m4(1).

Name

cpustat – report CPU statistics

Syntax

cpustat [–cfhs] [*interval*] [*count*]

Description

The *cpustat* command displays statistics about each CPU in the system. A full screen interface is provided, and the display is updated at intervals specified by the user. If *interval* is specified, then successive updates are summaries over the last *interval* seconds. If *count* is specified, the statistics and/or the state are repeated *count* times.

The display format fields are:

Statistics : Information about how each CPU's time is being utilized

us%	Percent of time spent in user mode
ni%	Percent of time spent in nice mode
sy%	Percent of time spent in system mode
id%	Percent of time spent idle by the CPU
csw	Number of context switches
sys	Number of system calls
trap	Number of traps
intr	Number of device interrupts
ipi	Number of inter processor interrupts
ttyin	Number of characters input through tty
ttyout	Number of character output through tty

State : Information about different states of each CPU

cpuid — Unique identifier of the CPU

state — CPU state:
- B - boot CPU
- D - disable soft errors
- S - stopped
- R - running
- T - TB needs invalidation
- P - panicked

ipi-mask — interprocessor interupt mask:
- P - panic
- R - console print
- S - schedule
- D - disable
- T - TB invalidation
- H - stop CPU

proc Indicates if the CPU has an associated process (Y/N)

pid Process id of the running process

If any statistic field value exceeds 9999, it is shown in a scaled representation with the suffix k, which indicates multiplication by 1000, or with the suffix m, which indicates multiplication by 1000000. For example, the value 12345 would appear as 12k.

Options

–c Displays state information about each CPU.

–f Displays statistics and state information on a full screen. If the **–f** option is used, the following commands can be entered from the screen:

- **c** Display only state information about each CPU
- **d** Go to the default mode of display
- **h** Display the help screen. Typing any character while on the help screen will display the original screen.
- **s** Display only statistics

–s Displays statistics about each CPU in the system.

–h Provides help information about the usage of cpustat.

If none of the options are specified, `cpustat` will report a summary of the statistics since the system has been booted and the state of each CPU.

Examples

To print the system status every five seconds ten times, type the following:

```
% cpustat 5 10
```

Files

```
/dev/kmem
/vmunix
```

See Also

iostat(1), vmstat(1)

crypt(1)

Name

crypt – encode/decode (available only if the Encryption layered product is installed)

Syntax

crypt *key* **< input.File > output.File**

Description

This reference page describes software that is available only if the Encryption layered product is installed.

The `crypt` command reads from the standard input and writes on the standard output. You must supply a *key* which 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. The `crypt` command encrypts and decrypts with the same *key*.

Files encrypted by `crypt` are compatible with those treated by the ed, ex and vi 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, and sneak paths by which *keys* or clear text can become visible must be minimized.

The `crypt` command 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, for example, to take a substantial fraction of a second to compute. However, if *keys* are restricted to three lowercase letters, then encrypted files can be read by expending only a substantial fraction of five minutes of machine time.

Since the *key* you choose is an argument to the `crypt` command, it is potentially visible to users executing ps(1) or a derivative. To minimize this possibility, `crypt` destroys any record of the *key* immediately upon entry. The most vulnerable aspect of `crypt` is the choice of *keys* and *key* security.

Examples

The following examples use KEY as the key to encrypt and decrypt files. The first example encrypts the file `plain.File`, naming the resulting encrypted file `crypt.File`. The second example decrypts the file `crypt.File`, naming the resulting decrypted file `decrypt.File`. The third example prints the encrypted file in clear text.

```
crypt KEY < plain.File > crypt.File

crypt KEY < crypt.File > decrypt.File

crypt KEY < crypt.File | pr
```

Files

`/dev/tty` for typed *key*

See Also

ed(1), ex(1), vi(1), xsend(1), crypt(3), makekey(8)

Name

csh – C shell Command Interpreter

Syntax

csh [–cefinstvVxX] [*arg...*]

Description

The csh command is a command language interpreter that consists of a history mechanism, job control facilities, and a C-like syntax. While this command has a set of *built-in* functions that it performs directly, the command line interpreter also reads and translates commands that invokes other programs. Additionally, you can create shell scripts which the csh command can interpret. Shell scripts are files which contain executable instructions.

The percent sign (%) represents the system prompt. It indicates that you can begin entering commands to the system. Each command line that you type is read and broken into words. This sequence of words is placed on a command history list and then parsed. When the entire command line has executed, the percent sign reappears and you can enter another command. See the History Substitution and Jobs sections for more information.

To use the csh commands full job control facilities, you must invoke the tty driver described in tty(4). This driver allows generation of interrupt characters from the keyboard which stop execution of a job. For details on setting options in the tty driver, see stty(1).

Note that your environment setup is controlled by commands in the home directory of your .cshrc file. The csh ccommand executes these commands when you enter the system. Additionally, if this is a login shell, the Shell also executes the commands in your .login file. These files usually contain your options for the tty driver and tset(1), (terminal settings). When a login shell session ends, commands are executed from the .logout file in your home directory.

Lexical Structure

The shell splits input lines into words at blanks and tabs with the following exceptions:

- ampersand (&)
- bar (|)
- semicolon (;)
- Left (<) and right (>) angle brackets
- Left (() and right ()) parenthesis

The previous metacharacters form separate words. If doubled as follows, these metacharacters form single words:

- Doubled ampersand (&&)
- Double bars (||)
- Double left (<<) and right (>>) brackets

- Backslash (\)
- Single (` `) and double (" ") quotation marks.

Metacharacters can be a part of other words. Additionally, if you do not want a metacharacter to be interpreted as such by the system, you can precede it with a backslash (\). A new line that is preceded by a is equivalent to a blank.

Strings enclosed in single quotes (` `) or strings enclosed in double quotes (" ") form parts of a word. Metacharacters in these strings, including blanks and tabs, do not form separate words. This is described in more detail later. Within single quotes or double quotes, a new line preceded by a backslash (\) gives a true new line character.

When the shell's input is not a terminal, the pound sign (#) introduces a comment which continues to the end of the input line. It is prevented this special meaning when preceded by a backslash (\) and single or double quotation marks.

Commands

A command is a word or sequence of words that directs the system to perform a certain function. You can separate commands with a bar (|) which forms a pipeline. The output that results from each command in the pipeline is connected to the input of the next. For example, in the following pipeline, a file is copied and the output is piped to standard output (the screen):

```
% cp /example/dir/test . | more
```

You can form and execute several pipelines by separating each pipeline with a semicolon (;). You can also force a command to complete execution in the background by typing an ampersand (&) at the end of the command line.

You can form a simple command (which may be a component of a pipeline and so on) by placing any of the above in parenthesis (()). As in the C language, you can also separate pipelines with a double bar (||) or double ampersands (&&). The double bar tells the command interpreter to execute the second command only if the first command fails. The double ampersands tells the command interpreter to execute the second command if the first command is successful.

Jobs

The Shell associates each command or pipeline with a *job* index. By typing **jobs** at the system prompt, a table of the current jobs is printed on your screen. Each job listed has a small integer number associated with it. For example, if you force a job into the background using an ampersand (&), the shell displays the job number and process id of that job as follows:

```
[1] 1234
```

In the previous example, the job number is 1 indicating that this is a background job and the process id is 1234.

If you are running a job in the foreground, you can suspend execution of that job by typing a CTRL/Z. The Shell then indicates that the job has been stopped and the system prompt reappears. If you type jobs at the prompt, the display indicates that a job has been stopped. You can either enter another command at the prompt or you can manipulate the state of the job you suspended as follows:

- Place the job in background by using the **bg** command.

- Continue to execute the job by placing it in the foreground using the **fg** command.

A CTRL/Z takes effect immediately and is like an interrupt. For example, pending output and unread output are discarded when the CTRL/Z is issued. You can also type a CTRL/Y which does not generate a stop signal until a program attempts to perform a `read`(2) operation.

If a job that is being run in the background attempts to read from the terminal, it will stop. Background jobs can produce output. You can prevent background jobs from producing output by issuing the following command:

```
stty tostop
```

There are several ways to refer to jobs in the shell. For example, to bring job number 1 into the foreground, type %1 or fg %1. Similarly, %1 & returns job 1 to the background. If a job does not have an ambiguous prefix, you can restart a job by it's prefix. For example, %ex would restart a suspended `ex` job, if it is the only suspended `ex` job. You use also use %?*string* which specifies a job whose command line contains *string*. Again, *string* cannot be an ambiguous name.

The Shell tracks the current and previous jobs. For example, in output displays of jobs, the current job is marked with a plus sign (+) and the previous job is marked with a minus sign (-). Hence, you can type %+ for the current job and %- for the previous job. You can also specify %% which specifies the current job.

Status Reporting

The Shell performs status reporting when the process state changes. For example, if a job becomes blocked and further processing is not possible, the Shell informs you just before it prints a prompt. If, however, you set the Shell variable *notify,* the Shell provides you with immediate status of background jobs. As opposed to notifying you of all changes in background jobs, the Shell command *notify* can mark a single process so that only its status change is reported. To mark a single file, type notify after starting a background job. By default, only the current process is marked.

If you try to exit from the Shell while jobs are stopped, the following warning appears:

```
You have stopped jobs.
```

You can use the **jobs** command to view the stopped jobs. If you immediately try to exit again, the Shell does not provide a second warning and suspended jobs are terminated.

Substitutions

The various transformations the shell performs on the input is now described in the order in which they occur.

History Substitutions

History substitutions allow you to use words from previously typed commands as portions of new commands. This enables you to repeat commands, arguments, or fix spelling mistakes from the previous command.

An exclamation point (!) marks the beginning of a history substitution. It can appear anywhere in the input stream (including the beginning) as long as it is not nested. An input line that contains history substitution is echoed to the screen before it is executed.

The exclamation point (!) may be preceded by a backslash (\) if you want to escape its special meaning. If an exclamation point is followed by a blank, tab, new line, equal sign (=), or left parenthesis ((), it is passed unchanged.

Any command line that is typed at the terminal is saved on the history list. You can increase or decrease the size of your history list using the *history* variable; the previous command is always retained regardless of its value. Commands are numbered sequentially from 1. To display the history on your terminal, type history at the prompt as follows:

```
% history
```

This command lists the commands that were previously typed. For example:

```
1  write michael
2  ex write.c
3  cat oldwrite.c
4  diff*write.c
```

The commands are shown with their event numbers. Although it is not usually necessary to use event numbers, you can reinvoke any command by combining the exclamation point (!) with any event number. For example, if you are referencing the previous history list, !4 reinvokes the command line diff*write.c. You can also reinvoke a command without the event number as long as it is not ambiguous. For example, !c invokes event 3 or !wri invokes event 1. The line !?mic? also refers to event 1. If you type !!, the last command entered in reinvoked.

To select words from an event, follow the event specification with a colon (:) and a designator for the desired words. The words of a input line are numbered from 0, the first (usually 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 colon (:) 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 colon (:). 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 new lines.

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 hand 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 '&' in the right hand 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 new line follows immediately as may the trailing '?' in a contextual scan.

A special abbreviation of a history reference occurs when the first non-blank character of an input line is a circumflex (^). 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 lb 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 below.

In both cases the resulting text becomes (all or part of) a single word; only in one special case (see *Command Substitution* below) does a '"' quoted string yield parts of more than one word; '´' quoted strings never do.

Alias Substitution

The shell maintains a list of aliases that can be established, displayed, and modified by the *alias* and *unalias* commands.

After the shell scans a command line, it parses the line into distinct commands. Then, the shell checks the first word of each command, in left-to-right order, to determine if the command line contains an alias. When the shell finds an alias, it substitutes the definition of the alias for the alias in the command line. The shell reads the definition of the alias using the history mechanism and treats the definition as if it was the previous input line. If the alias definition makes no reference to the history list, the shell leaves the command's argument unchanged.

For example, the following command creates an alias called ''ls:''

```
% alias ls ´ls -l´
```

After you issue this *alias* command, you receive information about files such as their mode, number of links, owner, and so on when you use the ls alias. For example, the following shows the output from the ls alias created in the preceeding example:

```
% ls /usr/smith/text_file
-rw-r--r--  1 smith         21 Mar 12 11:53 text_file
```

You can also create aliases that allow you to supply arguments on the command line and arguments in the alias definition, as shown in the following example:

```
% alias lookup 'grep \!^ /etc/passwd'
```

You must specify '\' before the ! to prevent the substitution from occurring in the *alias* command. The following shows the output from the lookup alias:

```
% lookup smith
smith:2vruqPosbG/bE:1321:10::/usr/smith:/bin/csh
```

The lookup alias finds and displays user Smith's entry in the /etc/passwd file.

You can specify an alias within an alias definition. After the shell finds an alias and substitutes its definition, it searches again for aliases. The shell flags definitions that begin with the same word as the alias to prevent infinite loops. Other loops are detected and cause an error.

You can use parser metasyntax in an *alias* command. For example, the following is a valid command that creates the print alias:

```
% alias print 'pr \!* | lpr'
```

The print alias pipes output from the pr command to the lpr command.

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 the shell or referred to by it. 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** command line option.

Other operations treat variables numerically. The '@' command permits numeric calculations to be performed and the result 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 '"'s where it **always** occurs, and within '''s where it **never** occurs. Strings quoted by '`' are interpreted later (see *Command substitution* below) 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 to 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 file name 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 file name 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 20 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 below 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. 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 colon (:) modifier on each '$' expansion."

The following substitutions may not be modified with colon (:) modifiers.

$?name
${?name}
Substitutes the string ‘1’ if name is set, ‘0’ if it is not.

$?0
Substitutes ‘1’ if the current input file name is known, ‘0’ if it is not.

$$
Substitute 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 File Name Substitution

The remaining substitutions, command and file name 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 new lines, with null words being discarded, this text then replacing the original string. Within ‘"’s, only new lines force new words; blanks and tabs are preserved.

In any case, the single final new line 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.

File Name Substitution

If a word contains any of the characters ‘*’, ‘?’, ‘[’ or ‘{’ or begins with the character ‘~’, then that word is a candidate for file name 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 file name substitution it is an error for no pattern to match an existing file name, but it is not required for each pattern to match. Only the metacharacters ‘*’, ‘?’ and ‘[’ imply pattern matching, the characters ‘~’ and ‘{’ being more akin to abbreviations.

In matching file names, the character ‘.’ at the beginning of a file name 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 file name is used to refer to home directories. Standing alone, that is ‘~’, 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 their home directory; thus ‘~ken’ might expand to ‘/usr/ken’ and ‘~ken/chmach’ to

'/usr/ken/chmach'. If the character '~' is followed by a character other than a letter or '/' or appears not at the beginning of a word, it is left undisturbed.

The metanotation 'a{b,c,d}e' is a 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 file name 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, file name 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 new lines preserved, except for the final new line 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, its previous contents being lost.

  If the variable *noclobber* is set, then 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 into the specified file as well as the standard output. *Name* is expanded in the same way as '<' input file names 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 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** above.)

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, '==' '!=' '=~' and '!~', '<=' '>=' '<' and '>', '<<' and '>>', '+' and '–', '*' '/' and '%' being, in groups, at the same level. The '==' '!=' '=~' and '!~' operators compare their arguments as strings; all others operate on numbers. The operators '=~' and '!~' are like '!=' and '==' except that the right hand side is a *pattern* (containing, for example, '*'s, '?'s and instances of '[...]') against which the left hand 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 file name 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 below.

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 non-seekable 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 file name 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 address of the last location in the heap. 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. Multi-level 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 below.

cd
cd name
chdir
chdir name
Change 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 this is tried to see if it is a directory.

continue
Continue 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 shell's standard output, separated by spaces, and terminated with a new line unless the **–n** option is specified.

else
end
endif
endsw
See the description of the *foreach, if, switch,* and *while* statements below.

eval arg ...
As in sh(1). The arguments are read as input to the shell and the resulting command(s) 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 `test`(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 alone 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 rub it out.

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 file name expand a list of words.

goto word
The specified *word* is file name 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
Print a statistics line indicating how effective the internal hash table has been at locating commands (and avoiding *exec*'s). 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** option reverses the order of printout to be most recent first rather than oldest first. The **–h** option causes the history list to be printed without leading numbers. This is used to produce files suitable for sourcing using the –h option to *source*.

if (expr) command
If the specified expression evaluates true, then the single *command* with arguments is executed. Variable substitution on *command* happens early, at the same time it does for the rest of the *if* command. *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 commands to the first *else* are executed; else if *expr2* is true then the commands to the second else are executed, etc. 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; given the **–l** options 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, saying just 'kill' does not send a signal to the current job. 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.

limit
limit *resource*
limit *resource maximum-use*

Limits the consumption by the current process and each process it creates to not individually exceed *maximum-use* on the specified *resource.* If no *maximum-use* is given, then the current limit is printed; if no *resource* is given, then all limitations are given.

The following resources can be controlled:

- *cputime* (maximum number of cpu-seconds to be used by each process)
- *filesize* (largest single file which can be created)
- *datasize* (the maximum growth of the data+stack region by `sbrk`(2) beyond the end of the program text)
- *stacksize* (the maximum size of the automatically-extended stack region)
- *coredumpsize* (the size of the largest core dump that can be created).
- *memoryuse* (the maximum amount of main memory a process is allowed to occupy)

The *maximum-use* may be given as a (floating point or integer) number followed by a scale factor. For all limits other than *cputime* the default scale is 'k' or 'kilobytes' (1024 bytes); a scale factor of 'm' or 'megabytes' may also be used. For *cputime* the default scaling is 'seconds', while 'm' for minutes or 'h' for hours, or a time of the form 'mm:ss' giving minutes and seconds may be used.

For both *resource* names and scale factors, unambiguous prefixes of the names suffice.

login
Terminate 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
Terminate 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 super-user may specify negative niceness by using 'nice –number ...'. Command is always executed in a sub-shell, and the restrictions place 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 ignored. All processes detached with '&' are effectively *nohup'ed.*

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
Control 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 a argument '+*n*' 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 (using

cd) and pushes the old current working directory (as in *csw)* onto the directory stack. With a numeric argument, rotates the *n* th argument of the directory stack around to be the top element and changes to 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'th* 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 file name 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 environment variable *name* to be *value,* a single string. The most commonly used environment variable 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 option causes the commands to be placed in the history list without being executed.

stop
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 **^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 file name 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 nothing to be *unaliased*.

unhash
Use of the internal hash table to speed location of executed programs is disabled.

unlimit *resource*
unlimit
Removes the limitation on *resource*. If no *resource* is specified, then all *resource* limitations are removed.

unset pattern

All variables whose names match the specified pattern are removed. Thus all variables are removed by 'unset *'; this 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 above and `printenv`(1).

wait

All background jobs are waited for. It 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 non-zero, 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'th* argument of *name*. Both *name* and its *index'th* component must already exist.

The operators '*=', '+=', etc 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, that is '@ i++'.

Pre-defined 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 will not then be modified unless this is done 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 then 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. Causes each command and its arguments to be echoed just before it is executed. For non-built-in commands all expansions occur before echoing. Built-in commands are echoed before command and file name 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. Too large values of *history* 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 file name 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 modify 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 on *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, file name expansion is inhibited. This is most useful in shell scripts which are not dealing with file names, or after a list of file names has been obtained and further expansions are not desirable.

nonomatch If set, it is not an error for a file name expansion to 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 rather 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 path names will execute. The usual search path is '.', '/bin' and '/usr/bin', but this may vary from system to system. For the super-user the default search path is '/etc', '/bin' and '/usr/bin'. A shell which is given neither the **–c** nor the **–t** 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 commands 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 super-user.

savehist is given a numeric value 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 this many events will be saved. During start up the shell sources ~/.history into the history list enabling history to be saved across logins. Too large values of *savehist* 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 *Non-built-in Command Execution* below.) 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 giving user, system, and real times and a utilization percentage which is the ratio of user plus system times to real time to be printed when it terminates. The **time** command can be used to cause a command to be timed no matter how much CPU time it takes. Thus

```
% time cp /etc/rc /usr/bill/rc
0.0u 0.1s 0:01 8% 2+1k 3+2io 1pf+0w
% time wc /etc/rc /usr/bill/rc
  52   178  1347 /etc/rc
  52   178  1347 /usr/bill/rc
```

```
  104  356  2694 total
0.1u 0.1s 0:00 13% 3+3k 5+3io 7pf+0w
%
```

The preceding example indicates that the *cp* command used a negligible amount of user time (u) and about 1/10th of a second system time (s); the elapsed time was 1 second (0:01), there was an average memory usage of 2k bytes of program space and 1k bytes of data space over the cpu time involved (2+1k); the program did three disk reads and two disk writes (3+2io), and took one page fault and was not swapped (1pf+0w). The word count command `wc` on the other hand used 0.1 seconds of user time and 0.1 seconds of system time in less than a second of elapsed time. The percentage '13%' indicates that over the period when it was active the command 'wc' used an average of 13 percent of the available CPU cycles of the machine.

verbose Set by the **–v** command line option, causes the words of each command to be printed after history substitution.

Non-built-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** 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 path name 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 prepended to the argument list to form the shell command. The first word of the *alias* should be the full path name of the shell (for example, '$shell'). Note that this is a special, late occurring, case of *alias* substitution, and only allows words to be prepended to the argument list without modification.

Argument List Processing

If argument 0 to the shell is '–' then this is a login shell. The flag arguments are interpreted as follows:

–c The first argument word is taken to be a command string. All remaining argument words are placed in *argv*.

–e The shell exits if any invoked command terminates abnormally or yields a non-zero exit status.

–f The shell will start faster, because it will neither search for nor execute commands from the file '.cshrc' in the invokers home directory.

–i The shell is interactive and prompts for its top-level input, even if stdin appears not to 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 new line 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 before '.cshrc' is executed.

After processing of flag arguments if arguments remain but none of the **–c, –i, –s,** or **–t** options was 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 either the standard version 6 or version 7 shells 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 *terminate* 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'.

Command And Filename Recognition

The `csh` command recognizes and completes user name aliases, commands (including built-in `csh` commands), and filenames. To use this feature, do the following:

1. Type enough characters at the prompt to make your input to the system unique.
2. Press the ESC key.

If your input is unique, the Shell completes the input line. If the input is not unique, the terminal signals you with a beep. If you receive a beep, type CTRL/D for a list of options. You can then type the additional characters that will make your text unique. After you have provided more input, press the ESC key again.

Command Line Editing

The csh command allows you to visually edit command lines using either a vi or emacs environment. The vi interface is modal and supports a subset of vi commands. The emacs interface is modeless and supports a subset of emacs commands. See the Editing Interface section for a list of the available vi and emacs commands.

To set the editing environment, define the Shell environment variable CSHEDIT as vi or emacs. If the environment variable CSHEDIT is not defined, the csh command searches for your EDITOR environment variable. When your EDITOR environment variable is set to vi, ex, edit, or ed, the csh command defaults to the vi command interface. If your EDITOR environment is not set to any of the previously mentioned editors, the default is the emacs command interface. Note that if neither the CSHEDIT or EDITOR environment variables are defined, the csh command defaults to the vi command interface.

The new history modifier (:v) allows you to pull commands from the history list to make them available for editing in visual edit mode. The symbol :v tells the Shell that you want to enter visual edit mode. For example, the following command line invokes edit mode for the previously typed cp command line:

```
!cp:v
```

When you press the ESC key as the first character on a command line, it is equivalent to typing the following:

```
!!:v
```

Thus, the previous example invokes edit mode for the last command you entered.

Another useful editing feature is scrolling through the history list. After you have entered edit mode by typing either !command:v or the ESC key, you can use the up-arrow and down-arrow keys to scroll through the history list and you may edit any command line in that history list.

When you are in edit mode, all control characters are displayed as a space character. Additional control characters cannot be inserted. Existing control characters are preserved.

Editing Interface

The available vi commands follow:

h Move left one character (r).

l Move right one character (r).

0 Move to the start of the line.

$ Move to the end of the line.

w move forward one word (r).

b Move back one word (r).

e Move to end of word (r).

fx Move forward onto character (r).

Fx Move back onto character (r).

tx	Move forward up to character (r).
Tx	Move back up to character (r).
%	Move to matching bracket ({[]}).
i	Insert text before cursor.
I	Insert text at beginning of line.
a	Append text after cursor.
A	Append text at end of line.
c	Change text (o).
C	Change to end of line (eol) (c$).
<esc>	End insertion.
x	Delete char under cursor (r).
X	Delete character before cursor (r).
r	Replace a character (r).
~	Change case of current character (r).
d	Delete text (o).
D	Delete to eol (d$).
u	Undo last change.
U	Undo all changes.
.	Repeat last text change command (r).
p	Put text from previous delete after cursor (r).
P	Put text from previous delete before cursor (r).
^L,^R	Redraw command line.
/word	Search back through the history list for a command containing the specified *word*. If the specified *word* is not delineated by white space in the history list, the search fails. Typing ESCAPE or CTRL/C aborts this command.
n	Repeat last history search.
<RETURN>	End edit and execute command.
^C	Quit; no command executed.
(r)	A repeat count is accepted.
(o)	Works within a cursor motion object.

The available `emacs` commands follow:

^@	Set mark (keyword null).
^A	Move to beginning of line.

^B,	Move backward a character.
^C	Exit command line edit; do not execute a command.
^D	Delete next character (to kill buffer).
^E	Move to end of line.
^F,	Move forward a character.
^G	Cancel partial command.
^H,DEL	Delete previous character (to kill buffer).
^K	Kill (delete) to end of line (to kill buffer).
^L	Redraw line display.
^R	Search reverse for a single character.
^S	Search forward for a single character.
^T	Transpose two characters before cursor.
^Un	Specify a repeat count before command (default of n is 4).
^W	Delete between cursor and mark (to kill buffer).
^Y	Yank from kill buffer.
CR,NL	End edit and execute command.
ESC-^C	End edit and execute command.
ESC-B	Move backward a word.
ESC-D	Delete next word.
ESC-F	Move forward a word.
ESC-H	Delete previous word.
ESC-DEL	Delete previous word.
ESC-n	Repeat count before command.
^X^C	End edit and execute command.
^Xu	Undo last change.
^XU	Undo all changes.
^X~	Change case of next character.
^X^Sword	Search back through the history list for a command containing a specified *word*. If the specified *word* is not delineated by white space in the history list, the search fails. Typing ESCAPE or CTRL/C aborts this command.
^X^S	Repeat last history search command. You must be in search mode to issue this command. Note that ^G cancels the previous search *word* so that you can enter a new *word*.

Restrictions

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 file name expansion is limited to 1/6'th 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.

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 ()'s to force it to a subshell, that is '(a ; b ; c)'.

Commands within loops, prompted for by '?', are not placed in the *history* list. Control structure should be parsed rather than being recognized as built-in commands. This would allow control commands to be placed anywhere, to be combined with '|', and to be used with '&' and ';' metasyntax.

It should be possible to use the colon (:) modifiers on the output of command substitutions. All and more than one colon (:) 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.

Files

~/.cshrc	Read at beginning of execution by each shell.
~/.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'.

See Also

sh(1), time(1), access(2), execve(2), fork(2), killpg(2), pipe(2), sigvec(2), setrlimit(2), umask(2), wait(2), tty(4), a.out(5), environ(7), time(7)
An Introduction to the C shell

Name

csplit – context split

Syntax

csplit [**–s**] [**–k**] [**–f** *prefix*] *file arg1* [*...argn*]

Description

The `csplit` command reads *file* and separates it into n+1 sections, as defined by the arguments *arg1...argn*. By default, the sections are placed in xx00...xxn (*n* may not be greater than 99). The named *file* is sectioned in the following way:

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: From the line referenced by *argn* to the end of *file*.

If the *file* argument is an minus (–) then standard input is used. A minus is an ASCII octal 055.

Options

–s Suppresses the printing of all character counts. If the **–s** option is omitted, the `csplit` command prints the character counts for each file created.

–k Leaves previously created files intact. If the **–k** option is omitted, `csplit` automatically removes created files if an error occurs.

–f*prefix* Names the created files *prefix00...prefixn*. The default is *xx00...xxn.*

The arguments (*arg1...argn*) to `csplit` can be a combination of the following:

/rexp/ A file is 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 argument may be followed by an optional plus (+) or minus (–) number of lines. For example, /Page/–5.

%rexp% This argument is the same as */rexp/*, except that no file is created for the section.

lnno A file is 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* 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 should not contain embedded new-lines. The `csplit` command does not affect the original file; it is the user's responsibility to remove it.

Examples

```
csplit -f cobol file /procedure division/ /par5./ /par16./
```

This example creates four files, cobol00...cobol03. After editing the files that `csplit` created, they can be recombined as follows:

```
cat cobol0[0-3] > file
```

Note that this example overwrites the original file.

```
csplit -k file  100  {99}
```

This example splits the file every 100 lines, up to 10,000 lines. The **-k** 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 right brace (}) at the beginning of the line, this example creates a file containing each separate C routine (up to 21) in `prog.c`.

Diagnostics

The diagnostics are self explanatory except for the following:

```
arg - out of range
```

This message means that the given argument did not reference a line between the current position and the end of the file.

See Also

ed(1), sh(1)

Name

ctags – create a tags file

Syntax

ctags [*options*] *name...*

Description

The `ctags` command makes a tags file for `ex(1)` from the specified C, Pascal and Fortran sources.

A tags file gives the locations of specified objects (in this case functions and typedefs) in a group of files. Each line of the tags file contains the object name, the file in which it is defined, and an address specification for the object definition. Functions are searched with a pattern, typedefs with a line number. Specifiers are given in separate fields on the line, separated by blanks or tabs.

Using the *tags* file, `ex` can quickly find these objects definitions.

If the **–x** flag is given, `ctags` produces a list of object 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. This is a simple index which can be printed out as an off-line readable function index.

Options

–a Appends information to an existing tags file.

–B Uses backward search patterns (?...?).

–F Uses forward search patterns (/.../) (default).

–t Creates typedef tags.

–u Updates the specified tags file. All references to tags are deleted, and the new values are appended to the file. Note that this option is implemented in a way which is rather slow. It is usually faster to simply rebuild the *tags* file.)

The tag *main* is treated specially in C programs. The tag formed is created by prepending *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.

–v Generates an index listing function name, file name, and pages number. Since the output will be sorted into lexicographic order, it may be desired to run the output through **sort –f**. For example,

```
ctags -v files | sort -f > index
```

Files whose name ends in **.c** or **.h** are assumed to be C source files and are searched for C routine and macro definitions. Others are first examined to see if they contain any Pascal or Fortran routine definitions; if not, they are

processed again looking for C definitions.

-w Suppresses warning diagnostics and generates a listing. This list contains each object name, its line number, the file name in which it is defined, and the text.

Restrictions

Recognition of **functions, subroutines** and **procedures** for FORTRAN and Pascal do not deal with block structure. Therefore you cannot have two Pascal procedures in different blocks with the same name.

Does not know about #ifdefs.

Does not know about Pascal types. Relies on the input being well formed to detect typedefs. Use of **-tx** shows only the last line of typedefs.

Files

tags output tags file

See Also

ex(1), vi(1)

ctod(1)

Name

ctod – combine DDIS objects into DOTS format

Syntax

ctod [**–x**] *object.ddis*

Description

The `ctod` command combines a DDIS encoded object into a Data Object Transport Syntax (DOTS) format, which is written to standard output. The object may contain references to other DDIS files. The purpose of `ctod` is to create a single file from multiple references to other files, in order to transfer or move DDIS objects from one location to another.

object.ddis is a file name, or a minus sign (–) for standard input. If a minus sign is specified, or if no file name is present, standard input is read. The named object and its external references, if any, are combined into a DOTS data stream which is written to standard output.

Because a DOTS stream contains binary data, `ctod` output should be redirected to a file or a pipe.

Options

–x Specifies that `ctod` is to DOTS encode the input file without resolving any external references present in the file. This option is for use only with files containing no external references.

Restrictions

The only DDIS object types supported in this release are DDIF and DTIF.

Diagnostics

The exit status is 0 if all files were combined successfully and 1 if any of the files could not be combined. Consult 'standard error' to see what files failed, and why.

If the **–x** option is used and `object.ddis` contains any external references, `ctod` returns an error status of 1, and writes an error message to 'standard error'.

See Also

dtoc(1), DDIS(5), DDIF(5), DTIF(5), DOTS(5)

Name

ctrace – C program debugger

Syntax

ctrace [*options*] [*file*]
ctc [*options*] [*file*]
ctcr [*options*] [*file*]

Description

The `ctrace` command allows you to follow the execution of a C program, statement by statement. The `ctrace` command reads the C program in *file* (or from standard input if you do not specify *file*) and inserts statements to print both the text of each executable statement and the values of all variables referenced or modified. It then writes the modified program to the standard output. You must put the output of `ctrace` into a temporary file because the `cc` command does not allow the use of a pipe. You then compile and execute this file.

As each statement in the program executes it is listed at the terminal. The statement is followed by the name and value of any variables referenced or modified in the statement, which is 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 you can put it into a file for examination with an editor or the `tail` command.

The `ctc` command is a shell script that prepares the specified C program *file* for later execution. The `ctcr` command is a shell script that both prepares and executes the specified C program *file*.

Options

The only options you will commonly use are:

–f *functions*	Trace only these *functions*.
–v *functions*	Trace all but these *functions*.

You may want to add to the default formats for printing variables. 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 can request that variables be printed in additional formats, if appropriate, with these options:

–e	Floating point
–o	Octal
–u	Unsigned
–x	Hexadecimal

These options are used only in special circumstances:

–l *n* Checks *n* consecutively executed statements for looping trace output, instead of the default of 20. Use 0 to get all the trace output from loops.

–P Runs the C preprocessor on the input before tracing it. You can also use the **–D**, **–I**, and **–U** cc(1) preprocessor options.

–p *s* Changes the trace print functions from the default of ''printf(''. For example, ''fprintf(stderr,'' would send the trace to the standard error output.

–r *f* Uses 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. For further information, see the **–p** option.

–s Suppresses redundant trace output from simple assignment statements and string copy function calls. This option can hide a bug caused by use of the = operator in place of the == operator.

–t *n* Traces *n* variables per statement instead of the default of 10 (the maximum number is 20). The DIAGNOSTICS section explains when to use this option.

Examples

Assume the file *lc.c* contains the following 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 }
```

When you enter the following commands and test data the program is compiled and executed:

```
cc lc.c
a.out
1
<CTRL/D>
```

The output of the program is 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. When you invoke `ctrace` with the following commands:

```
ctrace lc.c >temp.c
cc temp.c
a.out
```

the output is

```
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 is the following:

```
      /* 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     while ((c = getchar()) != EOF)
```

If you now enter an end of file character <CTRL/D>, the final output is the following:

```
      /* 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 realize that you used the assignment operator (=) in place of the equal operator (==). You can easily miss this error during code reading.

Execution-time Trace Control

The default operation for `ctrace` is to trace the entire program file, unless you use the **–f** or **–v** 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 turn the trace off and on by setting static variable tr_ct_ to 0 and 1, respectively. This is useful if you are using a debugger that cannot call these functions directly.

Restrictions

The `ctrace` command does not know about the components of aggregates such as 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. The `ctrace` command may choose to print the address of an aggregate or use the wrong format (for example, %e for a structure with two integer members) when printing the value of an aggregate.

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.

Warnings

You 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. Use a different name.

The `ctrace` command assumes that BADMAG is a preprocessor macro, and that EOF and NULL are #defined constants. Declaring any of these to be variables, for example, "int EOF;", will cause a syntax error.

Diagnostics

This section contains diagnostic messages from both `ctrace` and `cc`, 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** 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** option to preprocess the `ctrace` input, along with any appropriate **–D**, **–I**, and **–U** preprocessor options. If you still get the error message, check the Warnings section above.

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** 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>.

unimplemented structure assignment

Use pcc instead of cc(1).

Files

/usr/bin/ctc	preparation shell script
/usr/bin/ctcr	preparation and run shell script
/usr/lib/ctrace/runtime.c	run-time trace package

See Also

ctype(3), printf(3s), setjmp(3), signal(3), string(3)

Name

cut – cut out selected fields of each line of a file

Syntax

cut –c*list* [*file1 file2*...]
cut –f*list* [**–d***char*] [–s] [*file1 file2*...]

Description

Use the `cut` command to cut out columns from a table or fields from each line of a file. The fields as specified by *list* can be fixed length, that is, character positions as on a punched card (**–c** option), or the length can vary from line to line and be marked with a field delimiter character like *tab* (**–f** option). The `cut` command can be used as a filter. If no files are given, the standard input is used.

Use `grep`(1) to make horizontal ''cuts'' (by context) through a file, or `paste`(1) to put files together in columns. To reorder columns in a table, use `cut` and `paste`.

Options

list — Specifies ranges that must be a comma-separated list of integer field numbers in increasing order. With optional – indicates ranges as in the **–o** option of nroff/troff for page ranges; for example, **1,4,7**; **1–3,8**; **–5,10** (short for **1–5,10**); or **3–** (short for third through last field).

–c *list* — Specifies character positions to be cut out. For example, **–c1–72** would pass the first 72 characters of each line.

–f *list* — Specifies the fields to be cut out. For example, **–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* — Uses the specified character as the field delimiter. Default is *tab*. Space or other characters with special meaning to the shell must be quoted.

–s — Suppresses lines with no delimiter characters. Unless specified, lines with no delimiters will be passed through untouched. Either the **–c** or **–f** option must be specified.

Examples

Mapping of user IDs to names:

```
cut -d: -f1,5 /etc/passwd
```

To set **name** to the current login name:

```
name="` who am i | cut -f1 -d" "`
```

Diagnostics

"line too long"	A line can have no more than 511 characters or fields.
"bad list for c/f option"	Missing **-c** or **-f** 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.

See Also

grep(1), paste(1)

Name

cxref – generate C program cross reference

Syntax

cxref [*options*] *files*

Description

The `cxref` command analyzes a collection of C files and attempts to build a cross reference table. The `cxref` command utilizes a special version of `cpp` to include **#define**'d information in its symbol table. It produces a listing on standard output of all symbols (auto, static, and global) in each file separately, or with the **–c** option, in combination. Each symbol contains an asterisk (*) before the declaring reference.

Options

–c	Prints a combined cross-reference of all input files.
–D*name*	Defines *name* to processor, as if by *#define*. Default value is 1.
–I*dir*	Searches named directory for `#include` files whose names do not begin with a backslash (/).
–o *file*	Directs output to named *file*.
–s	Operates silently; does not print input file names.
–t	Formats listing for 80-column width.
–U*name*	Removes any initial definition of *name*.
–w<*num*>	Width option which formats output no wider than <*num*> (decimal) columns. This option will default to 80 if <*num*> is not specified or is less than 51.

Diagnostics

Error messages usually indicate a problem that will prevent the file from compiling.

Files

/usr/lib/xcpp special version of C-preprocessor.

See Also

cc(1).

Name

date – print date and time

Syntax

date [-c | -u] [+format] [[yy[mm[dd]]]hhmm[.ss][-[-]tttt][z]]

Description

If no argument is given, or if the argument begins with +, the current date and time are printed. 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 clock); the second *mm* is the minute number; *.ss* the second; -[-]*tttt* is the minutes west of Greenwich; a positive number means your time zone is west of Greenwich (for example, North and South America) and a negative number means it is east of Greenwich (for example Europe); *z* is a one letter code indicating the dst correction mode (n=none, u=usa, a=australian, w=western europe, m=middle europe, e=eastern europe); *yy* is the last 2 digits of the year number and is optional. The following example sets the date to Oct 8, 12:45 AM:

```
date 10080045
```

The current year is the default if no year is mentioned. The system operates in GMT. The `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 is 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 new-line character.

Options

–c Perform operations using Coordinated Universal Time (UCT) instead of the default local time. The UCT does not use leap seconds so UCT is the same as GMT.

–u Perform operations using Greenwich Mean Time (GMT) instead of the default local time.

+*format* The following is a list of field Descriptors that can be used in the format (Note: date exits after processing format information) :

- **%a** Locale's abbreviated weekday name
- **%A** Locale's full weekday name
- **%b** Locale's abbreviated month name
- **%B** Locale's full month name
- **%c** Locale's date and time representation
- **%d** Day of month as a decimal number (01–31)
- **%D** Date (%m/%d/%y)

%h Locale's abbreviated month name
%H Hour as a decimal number (00–23)
%I Hour as a decimal number (01–12)
%j Day of year (001–366)
%m Number of month (01–12)
%M Minute number (00–59)
%n Newline character
%p Locale's equivalent to AM or PM
%r Time in AM/PM notation
%S Second number (00–59)
%t Tab character
%T Time (%H/%M/%S)
%U Week number (00–53), Sunday as first day of week
%w Weekday number (0[Sunday]–6)
%W Week number (00–53), Monday as first day of week
%x Locale's date representation
%X Locale's time representation
%y Year without century (00–99)
%Y Year with century
%Z Timezone name, no characters if no timezone
%% %

Examples

The following command line

```
date +%m/%d/%y
```

generates the following output

```
04/02/89
```

The following command line

```
date +"DATE: %m/%d/%y%nTIME: %H:%M:%S"
```

generates the following output

```
DATE: 04/02/89
TIME: 14:45:05
```

The quotes (") are necessary because the *format* contains blank characters. Use single quotes (') to prevent interpretation by the shell.

Diagnostics

Failed to set date: Not owner
You are not the super-user and you tryed to change the date.

CAUTION

Do not change the date while the system is running in multiuser mode.

Restrictions

An attempt to set a date to before 1/1/1970 will result in the date being set to 1/1/1970.

Files

/dev/kmem

Name

dbx – source level debugger

Syntax

dbx [–I *directory*] [–c *file*] [–i] [–r] [-pixie] [*object*] [*core*]

Description

The dbx command is a source-level debugger. This enhanced version of dbx works with cc(1), f77(1), pc(1), as(1), and machine code. Note that f77(1) is a layered product which may not be installed on your system.

The object file used with the debugger is produced by specifying an appropriate option (usually **–g**) to the compiler. The resulting object file contains symbol table information, including the names of all source files that the compiler translated to create the object file. These source files are accessible from the debugger. If **–g** is not specified, limited debugging is possible.

If a core file exists in the current directory or a coredump file is specified, the dbx command can be used to look at the state of the program when it faulted.

Running dbx

If a *.dbxinit* file resides in the current directory or in the user's home directory, the commands in it are executed when dbx is invoked.

When invoked, dbx recognizes the following command line options:

–I *directory* or **–I***directory*
: Tells dbx to look in the specified directory for source files. Multiple directories can be specified by using multiple *–I* options. The dbx command searches for source files in the current directory and in the object file's directory whether or not *–I* is used.

–c *file*
: Selects a command file other than *.dbxinit.*

–i
: Uses interactive mode. This option does not treat #s as comments in a file. It prompts for source even when it reads from a file. With this option, dbx also has extra formatting as if for a terminal.

–r
: Runs the object file immediately.

–pixie
: Uses pixie output. The executable must be an executable.pixie, and the non-pixie executable must be in the same directory as the pixie executable.

Multiple commands can be specified on the same command line if you separate them with a semicolon (;). If you type a string and press the stop character (see stty(1)), dbx tries to complete a symbol name from the program that matches the string.

The Monitor

The following commands control the dbx monitor:

![*string*] [*integer*] [*–integer]*
: Specifies a command from the history list.

help Pipes a list of the dbx commands through the more command.

history Prints the items from the history list. The default is 20.

quit Exit dbx.

Controlling dbx

alias [*name(arg1,...argN) "string"*]
: Lists all existing aliases, or, if an argument is specified, defines a new alias.

unalias *alias command_name*
: Removes the specified alias.

delete *expression1, ...expressionN*

delete all Deletes the specified item from the status list. The argument **all** deletes all items from the status list.

playback input [*file*]
: Replays commands that were saved with the **record input** command in a text file.

playback output [*file*]
: Replays debugger output that was saved with the **record output** command.

record input [*file*]
: Records all commands typed to the dbx command.

record output [*file*]
: Records all dbx output.

sh [*shell command*]
: Calls a shell from dbx or executes a shell command.

status Lists currently set **stop, record,** and **trace** commands.

tagvalue (*tagname*)
: Returns the value of *tagname.* If the tag extends to more than one line, or if it contains arguments, an error occurs. **tagvalue** can be used in any expression.

set [*variable = expression*]
: Lists existing debugger variables and their values. This command can also be used to assign a new value to an existing variable or to define a new variable.

unset *variable* Removes the setting of a specified debugger variable.

Examining Source

/regular expression
Searches ahead in the source code for the regular expression.

?regular expression
Searches back in the source code for the regular expression.

edit [*file*] Calls an editor from the `dbx` environment.

file [*file*] Prints the current file name, or, if a file name is specified, this command changes the current file to the specified file.

func [*expression*] [*procedure*]
Moves to the specified procedure (activation level), or, if an expression or procedure isn't specified, prints the current activation level.

list [*expression:integer*]

list [*expression*] Lists the specified lines. The default is 10 lines.

tag *tagname* Sets the current file/line to the location specified by *tagname*. Operations are similar to the tag operations in `vi(1)`.

use [*directory1* ... *directoryN*]
Lists source directories, or, if a directory name is specified, this command substitutes the new directories for the previous list.

whatis *variable* Prints the type declaration for the specified name.

which *variable* Finds the variable name currently being used.

whereis *variable* Prints all qualifications (the scopes) of the specified variable name.

Controlling Programs

assign *expression1* = *expression2*
Assigns the specified expression to a specified program variable.

[*n*] **cont** [*signal*]

cont [*signal*] **to** *line*

cont [*signal*] **in** *procedure*
Continues executing a program after a breakpoint. *n* breakpoints are ignored if *n* is specified before stepping; If specified, *signal* is delivered to the processing being debugged.

goto *line* Goes to the specified line in the source.

next [*integer*] Steps over the specified number of lines. The default is one. This command does not step into procedures.

rerun [*arg1* ... *argN*] [*<file1*][*>file2*]

rerun [*arg1* ... *argN*] [*<file1*][*>&file2*]
Reruns the program, using the same arguments that were specified to the **run** command. If new arguments are specified, **rerun** uses those arguments.

run [*arg1* ... *argN*] [*<file1*] [*>file2*]

run [*arg1 ... argN*] [<*file1*] [>&*file2*]
Runs the program with the specified arguments.

return [*procedure*]
Continues executing until the procedure returns. If a procedure isn't specified, dbx assumes the next procedure.

step [*integer*] Steps the specified number of lines. This command steps into procedures. The default is one line.

Setting Breakpoints

catch [*signal*] Lists all signals that dbx catches, or, if an argument is specified, adds a new signal to the catch list.

ignore [*signal*] Lists all signals that dbx does not catch. If a signal is specified, this command adds the signal to the ignore list.

stop [*variable*]

stop [*variable*] **at** *line* [**if** *expression*]

stop [*variable]* **in** *procedure* [*if expression*]

stop [*variable*] **if** *expression*
Sets a breakpoint at the specified point.

trace *variable* [**at** *line* [*if expression*]

trace *variable* [**in** *procedure [if expression*]
Traces the specified variable.

when [*variable*] [**at** *line*] *{command_list}*

when [*variable*] [**in** *procedure*] *{command_list}*
Executes the specified dbx comma separated command list.

Examining Program State

dump [*procedure*] [.]
Prints variable information about the procedure. If a dot (.) is specified, this command prints global variable information on all procedures in the stack and the variables of those procedures.

down [*expression*]
Moves down the specified number of activation levels in the stack. The default is one level.

up [*expression*] Moves up the specified number of activation levels on the stack. The default is one.

print *expression1,...expressionN*
Prints the value of the specified expression.

printf *"string", expression1,...expressionN*
Prints the value of the specified expression, using C language string formatting.

printregs Prints all register values.

where Does a stack trace, which shows the current activation levels.

where *n* Prints out only the top *n* levels of the stack.

Debugging At The Machine Level

[*n*] **conti** [*signal*]

conti [*signal*] **to** *address*

conti [*signal*] **in** *procedure*
Continues executing assembly code after a breakpoint. *n* breakpoints are ignored if *n* is specified before stepping; If specified, *signal* is delivered to the processing being debugged.

nexti [*integer*] Steps over the specified number of machine instructions. The default is one. This command does not step into procedures.

stepi [*integer*] Steps the specified number of machine instructions. This command steps into procedures. The default is one instruction.

stopi *[variable]* **at** *address* [**at** *address* **if** *expression*]

stopi [*variable*] **in** *procedure* [**if** *expression*]

stopi [*variable*] **if** *expression*
Sets a breakpoint in the machine code at the specified point.

tracei *variable* **at** *address* [**at** *address* **if** *expression*]

tracei *variable* **in** *procedure* [**at** *address* **if** *expression*]
Traces the specified variable in machine instructions.

wheni [*variable*] [**at** *address*] *{command_list}*

wheni [*variable*] [**in** *procedure*] *{command_list}*
Executes the specified dbx comma separated command list.

address[**?**]/<count><mode>
Searching forward (or backward, if **?** is specified,) prints the contents *address* or disassembles the code for the instruction *address*; *count* is the number of items to be printed at the specified address. *mode* is one of the characters in the following table producing the indicated result:

d	Print a short word in decimal
D	Print a long word in decimal
o	Print a short word in octal
O	Print a long word in octal
x	Print a short word in hexadecimal
X	Print a long word in hexadecimal
b	Print a byte in octal
c	Print a byte as a character
s	Print a string of characters that ends in a null
f	Print a single precision real number
g	Print a double precision real number
i	Print machine instructions
n	Prints data in typed format.

address/<countL><value><mask>
: Searches for a 32-bit word starting at the specified *address*; *count* specifies the number of word to process in the search; an address is printed when the the word at *address*, after an AND operation with *mask*, is equal to *value*.

Predefined dbxVariables

The debugger has the following predefined variables:

$addrfmt
: Specifies the format for addresses. This can be set to any specification that a C printf statement can format. The default is zero.

$byteaccess
: Same as $addrfmt.

$casesense
: When set to a nonzero value, specifies that uppercase and lowercase letters be taken into consideration during a search. When set to 0, the case is ignored. The default is 0.

$curevent
: Shows the last even number as seen in the status feature. Set only by `dbx`.

$curline
: Specifies the current line. Set only by `dbx`.

$cursrcline
: Shows the last line listed plus 1. Set only by `dbx`.

$curpc
: Specifies the current address. Used with the *wi* and *li* aliases.

$datacache
: Caches information from the data space so that `dbx` must access data space only once. To debug the operating system, set this variable to 0; otherwise, set it to a nonzero value. The default is 1.

$dbgmon
: For internal use by `dbx`.

$defaultin
: For internal use by `dbx`.

$defaultout
: For internal use by `dbx`.

$dispix
: For use when debugging pixie code. When set to 0, machine code is shown while debugging. When set to 1, pixie code is shown. The default is 0.

$hexchars
: Output characters are printed in hexadecimal format (set, unset).

$hexin
: Specifies that input constants are hexadecimal.

$hexints
: When set to a nonzero value, changes the default output constants to hexadecimal. Overrides *$octints*.

$hexstrings
: When set to 1, specifies that all strings are printed in hexadecimal; when set to 0, strings are printed in character format.

$historyevent
: Shows the current history line.

$lines
: Number of lines for history. The default is 20

$listwindow
: Specifies how many lines the *list* command prints.

$main
: Specifies the name of the procedure that `dbx` begins to process. The `dbx` command can point to any procedure. The default is "main".

$maxstrlen	Specifies how many characters of a string `dbx` prints for pointers to strings. The default is 128.
$octin	When set to non-zero, changes the default input constants to octal. When set, *$hexin* overrides this setting.
$octints	Output integers are printed octal format (set, unset).
$page	Specifies whether to page long information. A nonzero value turns on paging; a 0 turns it off. The default is 1.
$pagewindow	Specifies how many lines print when information runs longer than one screen. This can be changed to match the number of lines on any terminal. If set to 0, this variable assumes one line. The default is 22, leaving space for continuation query.
$printwhilestep	For use with the **step**[*n*] and **stepi**[*n*] instructions. A non-zero integer specifies that all *n* lines and/or instructions should be printed out. A zero specifies that only the last line and/or instruction should be printed out. The default is zero.
$pimode	Prints input when used with the *playback input* command. The default is 0.
$printdata	When set to a nonzero value, the contents of registers used are printed next to each instruction displayed. The default is 0.
$printwide	When se to a nonzero value, the contents of variables are printed in a horizontal format. The default is 0.
$prompt	Sets the prompt for `dbx`.
$readtextfile	When set to 1, `dbx` tries to read instructions from the object file rather than the process. The `dbx` command executes faster when debugging remotely using the System Programmer's Package. This variable should always be set to 0 when the process being debugged copies in code during the debugging process. The default is 1.
$regstyle	A zero value causes registers to be printed out in their normal *r* format (r0,r1,...r31). A nonzero value causes the registers to be printed out in a special format (zero, at, v0, v1,...) commonly used in debugging programs written in assembly language. The default is 0.
$repeatmode	When set to a nonzero value, after pressing the RETURN key (for an empty line), the last command is repeated. The default is 1.
$rimode	When set to a nonzero value, input is recorded while recording output. The default is 0.
$sigtramp	Tells `dbx` the name of the code called by the system to invoke user signal handlers. This variable is set to sigvec on ULTRIX systems.
$tagfile	Contains a filename, indicating the file in which the tag command and the tagvalue macro are to search for tags.

Predefined dbx Aliases

The debugger has the following predefined aliases:

?	Prints a list of all `dbx` commands.
a	Assigns a value to a program variable.
b	Sets a breakpoint at a specified line.
bp	Stops in a specified procedure.
c	Continues program execution after a breakpoint.
d	Deletes the specified item from the status list.
e	Looks at the specified file.
f	Moves to the specified activation level on the stack.
g	Goes to the specified line and begins executing the program there.
h	Lists all items currently on the history list.
j	Shows what items are on the status list.
l	Lists the next 10 lines of source code.
li	Lists the next 10 machine instructions.
n or **S**	Step over the specified number of lines without stepping into procedure calls.
ni or Si	Step over the specified number of assembly code instructions without stepping into procedure calls.
p	Prints the value of the specified expression or variable.
pd	Prints the value of the specified expression or variable in decimal.
pi	Replays `dbx` commands that were saved with the **record input** command.
po	Prints the value of the specified expression or variable in octal.
pr	Prints values for all registers.
px	Prints the value for the specified variable or expression in hexadecimal.
q	Ends the debugging session.
r	Runs the program again with the same arguments that were specified with the **run** command.
ri	Records in a file every command typed.
ro	Records all debugger output in the specified file.
s	Steps the next number of specified lines.
si	Steps the next number of specified lines of assembly code instructions.
t	Does a stack trace.
u	Lists the previous 10 lines.

w Lists the 5 lines preceding and following the current line.

W Lists the 10 lines preceding and following the current line.

wi Lists the 5 machine instructions preceding and following the machine instruction.

Name

dbx – debugger

Syntax

dbx [**–r**] [**–i**] [**–k**] [**–I** *dir*] [**–c** *file*] [*objfile* [*coredump*]]

Description

The dbx debugger is a tool for source level debugging and execution of programs running under the ULTRIX operating system.

After invoking dbx, you can debug interactively by using the commands described in the Commands section. If the file .dbxinit exists in the current directory, then the dbx commands in it are executed. If the file does not exist in the current directory, the user's home directory is then checked for a .dbxinit file. Note that the init file is built by appending the characters init to the first eight characters of the debugger's name. For example, if you renamed dbx to **abcdefghi**, the debugger would look for an initialization file named **.abcdefghinit.**

Arguments

objfile An object file that is produced by a compiler with the appropriate option (usually **–g**), and that is specified to produce symbol information in the object file. The cc(1), pc(1), and vcc(1), produce the appropriate source information. The machine level facilities of dbx can be used on any program.

The object file contains a symbol table that includes the name of all the source files translated by the compiler to create it. These files are available for perusal while using the debugger.

If no *objfile* is specified, dbx looks for a file named a.out in the current directory.

coredump If a file named *core* exists in the current directory, or a *coredump* file is specified, dbx can be used to examine the state of the program when it faulted.

Options

–r Executes *objfile* immediately. If it terminates successfully, dbx exits. Otherwise, the reason for termination will be reported and the user is offered the option of entering the debugger or letting the program fault. The dbx debugger will read from /dev/tty when **–r** is specified and standard input is not a terminal.

–i Forces dbx to act as though standard input is a terminal.

–k Maps memory addresses, useful for kernel debugging.

–I*dir* Adds *dir* to the list of directories that are searched when looking for a source file. Normally, dbx looks for source files in the current directory

and in the directory where *objfile* is located. The directory search path can also be set with the **use** command.

–cfile Executes the dbx commands in the *file* before reading from standard input.

Unless the **–r** option is specified, the dbx command just prompts and waits for a command.

Commands

Execution And Tracing Commands

run [*args*] [< *filename*] [> *filename*]
rerun [*args*] [< *filename*] [> *filename*]

Start executing *objfile*, passing *args* as command line arguments; Angle brackets (< or >) can be used to redirect input or output in the usual manner. When **rerun** is used without any arguments, the previous argument list is passed to the program. Otherwise it is identical to **run**. If *objfile* has been written since the last time the symbolic information was read in, dbx will read in the new information.

trace [**in** *procedure/function*] [**if** *condition*]
trace *source-line-number* [**if** *condition*]
trace *procedure/function* [**in** *procedure/function*] [**if** *condition*]
trace *expression* **at** *source-line-number* [**if** *condition*]
trace *variable* [**in** *procedure/function*] [**if** *condition*]

Have tracing information printed when the program is executed. A number is associated with the command that is used to turn the tracing off (see the **delete** command).

The first argument describes what is to be traced. If it is a *source-line-number*, then the line is printed immediately prior to being executed. Source line numbers in a file other than the current one must be preceded by the name of the file in quotes and a colon, e.g. "mumble.p":17.

If the argument is a procedure or function name, then every time it is called information is printed telling what routine called it, from what source line it was called, and what parameters were passed to it. In addition, its return is noted and if it's a function then the value it is returning is also printed.

If the argument is an *expression* with an **at** clause then the value of the expression is printed whenever the identified source line is reached.

If the argument is a variable then the name and value of the

variable is printed whenever it changes. Execution is substantially slower during this form of tracing.

If no argument is specified then all source lines are printed before they are executed. Execution is substantially slower during this form of tracing.

The clause **in** *procedure/function* allows tracing information to be printed only while executing inside the given procedure or function.

The *condition* is a boolean expression and is evaluated prior to printing the tracing information; if it is false then the information is not printed.

stop if *condition*

stop at *source-line-number* [**if** *condition*]
stop in *procedure/function* [**if** *condition*]
stop *variable* [**if** *condition*]
Stop execution when the given line is reached, procedure or function called, variable changed, or condition true.

status [> *filename*] Print out the currently active **trace** and **stop** commands.

delete *command-number* ...
The traces or stops corresponding to the given numbers are removed. The numbers associated with traces and stops are printed by the **status** command. The command **delete*** removes all existing breakpoints and tracepoints at once.

catch *number*

catch *signal-name*

ignore *number*

ignore *signal-name* Start or stop trapping a signal before it is sent to the program. This is useful when a program being debugged handles signals such as interrupts. A signal may be specified by number or by a name (for example, SIGINT). Signal names are case insensitive and the SIG prefix is optional. By default all signals are trapped except SIGCONT, SIGCHILD, SIGALRM, and SIGKILL.

cont *integer*

cont *signal-name* Continue execution from where it stopped. If a signal is specified, the process continues as though it received the signal. Otherwise, program execution continues as if a signal had not been encountered.

Execution cannot be continued if the process has called the standard procedure exit. The `dbx` debugger does not allow the process to exit, thereby letting the user examine the program state.

step Execute one source line.

next Execute up to the next source line. The difference between **next** and **step** is that if the line contains a call to a procedure or function the **step** command will stop at the beginning of that block, while the **next** command will not.

return [*procedure*] Continue until a return to *procedure* is executed, or until the current procedure returns if none is specified.

call *procedure(parameters)*
Execute the object code associated with the named procedure or function.

Printing Variables And Expressions

Names are resolved first using the static scope of the current function, then using the dynamic scope if the name is not defined in the static scope. If static and dynamic searches do not yield a result, an arbitrary symbol is chosen and the message [using *qualified name*] is printed. The name resolution procedure may be overridden by qualifying an identifier with a block name, for example, *module.variable*. For C, source files are treated as modules named by the file name without .c.

Expressions are specified with an approximately common subset of C and Pascal (or equivalently Modula-2) syntax. Indirection can be denoted using either an asterisk (*) as a prefix or a circumflex (^) as a postfix. Array expressions are enclosed in brackets ([ ]), and the field reference operator (.) can be used with pointers as well as records, making the C operator (->) unnecessary (although it is supported).

Types of expressions are checked; the type of an expression may be overridden by using (*expression*)*type-name*.

assign *variable* = *expression*
Assign the value of the expression to the variable.

dump [*procedure*] [> *filename*]
Print the names and values of variables in the given procedure, or the current one if none is specified. If the procedure given is the field reference operator (.), then the all active variables are dumped.

print *expression* [, *expression* ...]
Print out the values of the expressions.

whatis *name* Print the declaration of the given name, which may be qualified with block names as above.

which *identifier* Print the full qualification of the given identifier, i.e. the outer blocks that the identifier is associated with.

up [*count*]

down [*count*] Move the current function, which is used for resolving names, up or down the stack *count* levels. The default *count* is 1.

where Print out a list of the active procedures and function.

whereis *identifier* Print the full qualification of all the symbols whose name

matches the given identifier. The order in which the symbols are printed is not meaningful.

Accessing Source Files

/regular expression[/]

?*regular expression*[?] Search forward or backward in the current source file for the given pattern.

edit [*filename*]

edit *procedure/function-name*
Invoke an editor on *filename* or the current source file if none is specified. If a *procedure* or *function* name is specified, the editor is invoked on the file that contains it. Which editor is invoked by default depends on the installation. The default can be overridden by setting the environment variable EDITOR to the name of the desired editor.

file [*filename*] Change the current source file name to *filename*. If none is specified then the current source file name is printed.

func [*procedure/function*]
Change the current function. If none is specified then print the current function. Changing the current function implicitly changes the current source file to the one that contains the function; it also changes the current scope used for name resolution.

list [*source-line-number* [, *source-line-number*]]
list *procedure/function* List the lines in the current source file from the first line number to the second inclusive. If no lines are specified, the next 10 lines are listed. If the name of a procedure or function is given, lines *n-k* to *n+k* are listed, where *n* is the first statement in the procedure or function and *k* is small.

use *directory-list* Set the list of directories to be searched when looking for source files.

Command Aliases And Variables

alias *name name*

alias *name string*

alias *name* (*parameters*) *string*
When commands are processed, `dbx` first checks to see if the word is an alias for either a command or a string. If it is an alias, then `dbx` treats the input as though the corresponding string (with values substituted for any parameters) had been entered. For example, to define an alias rr for the command `rerun`, type

alias rr rerun

To define b as an an alias that sets a stop at a particular line type

alias b(x) "stop at x"

Subsequently, the command `b(12)` will be interpreted as stop at 12.

set *name* [= *expression*] The **set** command defines values for debugger variables. The names of these variables cannot conflict with names in the program being debugged and are expanded to the corresponding expression within other commands. The following variables have a special meaning:

$frame
: Setting this variable to an address causes `dbx` to use the stack frame pointed to by the address for doing stack traces and accessing local variables. This facility is of particular use for kernel debugging.

$hexchars
$hexints
$hexoffsets
$hexstrings
: When set, `dbx` prints out characters, integers, offsets from registers, or character pointers respectively in hexadecimal.

$listwindow
: The value of this variable specifies the number of lines to list around a function or when the **list** command is given without any parameters. Its default value is 10.

$mapaddrs
: Setting (unsetting) this variable causes `dbx` to start (stop) mapping addresses. As with the $frame variable, this is useful for kernel debugging.

$unsafecall
$unsafeassign
: When the $unsafecall variable is set, strict type checking is turned off for arguments to subroutine or function calls (for example, in the **call** statement), as is strict type checking between the two sides of an **assign** statement. These variables should be used with care, because they severely limit `dbx`'s usefulness for detecting errors.

unalias *name* Remove the alias with the given name.

unset *name* Delete the debugger variable associated with *name*.

Machine Level Commands

tracei [*address*] [**if** *cond*]
tracei [*variable*] [**at** *address*] [**if** *cond*]
stopi [*address*] [**if** *cond*]
stopi [**at**] [*address*] [**if** *cond*]

Turn on tracing or set a stop using a machine instruction address.

stepi

nexti Single step as in **step** or **next**, but do a single instruction rather than source line.

address ,address/ [*mode*]
address / [*count*] [*mode*]

Print the contents of memory starting at the first *address* and continuing up to the second *address* or until *count* items are printed. If you type a period (.) in the address field, the address following the one printed most recently is used. The *mode* specifies how memory will be printed; if it is omitted, the previous mode that was specified is used. The initial mode is X.

The following modes are supported:

i print the machine instruction
d print a short word in decimal
D print a long word in decimal
o print a short word in octal
O print a long word in octal
x print a short word in hexadecimal
X print a long word in hexadecimal
b print a byte in octal
c print a byte as a character
s print a string of characters terminated by a null byte
f print a single precision real number
g print a double precision real number

Symbolic addresses are specified by preceding the name with an ampersand (&). Registers are denoted by $rN where N is the number of the register. Addresses may be expressions made up of other addresses and the operators plus (+), (-), and indirection (unary asterisk, *).

Miscellaneous Commands

help Print out a synopsis of dbx commands.

quit Exit dbx.

sh *command-line*

Pass the command line to the shell for execution. The SHELL

environment variable determines which shell is used.

source *filename*
Read dbx commands from the given *filename.*

Restrictions

If you have a program consisting of several object files and each is built from source files that include header files, the symbolic information for the header files is reproduced in each object code file. Since one debugger startup usually is done for each link, having the linker ld(1) reorganize the symbol information will not save much time, although it would reduce some of the disk space used.

The problem results from the unrestricted semantics of #include statements in C. For example, an include file can contain static declarations that are separate entities for each file in which they are included. If your image is too large for dbx to run, compile with the **-g** switch only those files that you are interrested in debugging. However, even with Modula-2, there is a substantial amount of duplication of symbol information necessary for inter-module type checking.

Some problems remain with the support for individual languages. Fortran problems include: (a) inability to assign to logical, logical*2, complex, and double complex variables, (b) inability to represent parameter constants which are not type integer or real, (c) peculiar representation for the values of dummy procedures. (The value shown for a dummy procedure is actually the first few bytes of the procedure text. To find the location of the procedure, use an ampersand (&) to take the address of the variable.)

The dbx debugger does not allow you to run a program you do not own unless you are root. If you are not root, the message message string can't-write-to-process may be displayed on your screen when you issue the run command. This occurs when the dbx debugger tries to set breakpoints because of restrictions on ptrace(2). If you repeat the run command, your program runs without breakpoints. The dbx debugger always tries set a breakpoint on exit.

Files

a.out	Object file
.dbxinit	Initial commands

See Also

cc(1), pc(1), ptrace(1), vcc(1)

Name

dc – desktop calculator

Syntax

dc [*file*]

Description

The dc command is an arbitrary precision arithmetic package. Ordinarily it operates on decimal integers, but you can 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 The value of the number is pushed on the stack. A number is an unbroken string of the digits 0-9. It may be preceded by an underscore _ to input a negative number. Numbers may contain decimal points.

+ – / * % ^
: The top two values on the stack 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* The top of the stack is popped and stored into a register named *x*, where *x* may be any character. If the **s** is capitalized, *x* is treated as a stack and the value is pushed on it.

l*x* The value in register *x* is pushed on the stack. The register *x* is not altered. All registers start with zero value. If the **l** is capitalized, register *x* is treated as a stack and its top value is popped onto the main stack.

d The top value on the stack is duplicated.

p The top value on the stack is printed. The top value remains unchanged. **P** interprets the top of the stack as an ascii string, removes it, and prints it.

f All values on the stack are printed.

q Exits the program. If executing a string, the recursion level is popped by two. If **q** is capitalized, the top value on the stack is popped and the string execution level is popped by that value.

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.

[...] Puts the bracketed ascii string onto the top of the stack.

<*x* >*x* =*x*
: The top two elements of the stack are popped and compared. Register *x* is executed 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 UNIX command.

c All values on the stack are popped.

i The top value on the stack is popped and used as the number radix for further input. When the base (number radix) is re-set, all subsequent numbers are interpreted in the new base.

For example, if the command is issued twice, first to set the base to base 2, then to reset it back to base 10, the new base value must be given in the base originally set (that is, '2 i' will set the base to base 2, after which '1010 i' will set it back to base 10).

I Pushes the input base on the top of the stack.

o The top value on the stack is popped and used as the number radix for further output.

O Pushes the output base on the top of the stack.

k The top of the stack is popped, and that value is used as a non-negative scale factor: the appropriate number of places are printed on output, and maintained during multiplication, division, and exponentiation. The interaction of scale factor, input base, and output base will be reasonable if all are changed together.

z The stack level is pushed onto the stack.

Z Replaces the number on the top of the stack with its length.

? A line of input is taken from the input source (usually the terminal) and executed.

; : Used by `bc` for array operations.

An example which prints the first ten values of n! is the following:

```
[la1+dsa*pla10>y]sy
0sa1
lyx
```

Diagnostics

"x is unimplemented"
x is an octal number.

"stack empty"
Not enough elements on the stack to do what was asked.

"Out of space"
The free list is exhausted (too many digits).

"Out of headers"
Too many numbers being kept around.

"Out of pushdown"
Too many items on the stack.

"Nesting Depth"
Too many levels of nested execution.

See Also

bc(1)

Name

dd – copy and convert data

Syntax

dd [*option* = *value...*]

Description

The dd command copies an input file to an output with any requested conversions. The dd command is especially suited to I/O on the raw physical devices because it allows reading and writing in arbitrary record sizes.

After completion, dd reports the number of whole and partial input and output blocks.

This utility supports EOT handling which allows the use of multiple media. The utility prompts for the next volume when it encounters the end of the current volume.

Options

Where sizes (*n*) are given for an option, the number may end with **k** for kilobytes (1024 bytes), **b** for blocks (512 bytes), or **w** for words (2 bytes). Also, two numbers may be separated by the character x to indicate a product.

if=*name* Input file name. The standard input is the default.

of=*name* Output file name. The standard output is the default.

ibs=*n* Input block size, *n* bytes. The default is 512 bytes. Some devices do not support greater than 65,535 bytes.

obs=*n* Output block size, *n* bytes. The default is 512 bytes. Some devices do not support greater than 65,535 bytes.

bs=*n* Set both input and output block size to *n* bytes, superseding **ibs** and **obs.** Also, if **bs** is specified, the copy is more efficient, since no blocking conversion is necessary.

cbs=*n* Conversion buffer size, *n* bytes. Use only if **ascii, unblock, ebcdic, ibm**, or **block** conversion is specified. For **ascii** and **unblock**, *n* characters are placed into the conversion buffer, any specified character mapping is done, trailing blanks are trimmed and new line added before sending the line to the output. For **ebcdic, ibm**, or **block**, characters are read into the conversion buffer, and blanks added to make an output record of size *n* bytes.

skip=*n* Skip *n* input records before starting to copy.

files=*n* Copy *n* input files before terminating. This option is useful only when the input is a magnetic tape or similar device.

seek=*n* Seek *n* records from beginning of output file before copying.

rbuf=*n* Use *n* buffers for reading from those raw devices that support n-buffered I/O. (See Section 4 to check whether a specific device supports n-buffered I/O.) All *n* reads are

started and each read must complete before the data can be used. This allows an n-buffered read-ahead on supported raw devices.

A default of eight read buffers are used if the read device supports n-buffered I/O and the write device does not.

The **rbuf** option cannot be used with the **wbuf** option.

wbuf=*n*
: Use *n* buffers for writing from those raw devices that support n-buffered I/O. (See Section 4 to check whether a specific device supports n-buffered I/O.) Each write is started but not known to be complete until all *n* buffers have been used. (This allows an n-buffered write-behind on supported raw devices).

  A default of eight write buffers are used if the write device supports n-buffered I/O.

  The **wbuf** option cannot be used with the **rbuf** option.

count=*n*
: Copy only *n* input records.

conv=ascii
: Convert EBCDIC to ASCII.

conv=ebcdic
: Convert ASCII to EBCDIC.

conv=ibm
: Slightly different map of ASCII to EBCDIC (see RESTRICTIONS).

conv=block
: Convert variable length records to fixed length.

conv=unblock
: Convert fixed length records to variable length.

conv=lcase
: Map alphabetics to lower case.

conv=ucase
: Map alphabetics to upper case.

conv=swab
: Swap every pair of bytes.

conv=noerror
: Do not stop processing on an error.

conv=sync
: Pad every input record to *ibs*.

conv=nomulti
: Disable multiple tape volumes.

conv=sparse
: Create a sparse output file.

conv=... , ...
: Include several arguments for the **conv** option, separated by commas (see example below).

Examples

The following example shows how to read an EBCDIC tape blocked ten 80-byte EBCDIC card images per record into the ASCII file *x:*

```
dd if=/dev/rmt0h of=x ibs=800 cbs=80 conv=ascii,lcase
```

Note the use of raw magtape. As noted in the DESCRIPTION, the dd command is especially suited to I/O on the raw physical devices because it allows reading and writing in arbitrary record sizes.

Restrictions

The ASCII/EBCDIC conversion tables are taken from the 256-character standard in the *Communications of the ACM,* November, 1968.

The **ibm** conversion corresponds to certain IBM print train conventions.

One must specify ''conv=noerror,sync'' when copying raw disks with bad sectors to ensure that *dd* stays synchronized.
On SCSI tape devices when reading a multi-volume tape set the command will exit normally upon hitting EOT on any volume rather than automatically unloading the volume and prompting for the next volume as is normal. The user should load the next volume and issue the command anew.

Diagnostics

f+p records in(out): numbers of full and partial records read(written)

See Also

cp(1), tr(1), nbuf(4)

Name

delta – create new SCCS delta to save changes

Syntax

delta [**–r***SID*] [**–s**] [**–n**] [**–g***list*] [**–m** [*mrlist*]] [**–y** [*comment*]] [**–p**] *files*

Description

The `delta` command 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).

The `delta` command 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 RESTRICTIONS); each line of the standard input is taken to be the name of an SCCS file to be processed.

The `delta` command may issue prompts on the standard output depending upon certain keyletters specified and flags that may be present in the SCCS file. For further information, see **–m** and **–y** keyletters below and `admin`(1).

The `delta` includes commentary, input by the user, that consists of one or more lines, terminated by a period (.) in column one of a new line.

Keyletter arguments apply independently to each named file.

Options

Keyletter arguments:

–g*list* Ignores specified list of deltas.

–m[*mrlist*] Indicates the modification request number. (**–m**[*mrlist*]).

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 new-line character terminates the **MR** list.

Note that if the **v** flag has a value it is taken to be the name of a program (or shell procedure) which will validate the correctness of the **MR** numbers. For further information, see `admin`(1). If a non-zero exit status is returned from **MR** number validation program, `delta` terminates (it is assumed that the **MR** numbers were not all valid).

–n Does not delete edited file.

–p Displays differences before and after delta is applied.

–r*SID* Identifies which delta is to be made to the SCCS file. Use

this keyletter only if two or more outstanding *gets* for editing (**get –e**) on the same SCCS file has been 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. For further information, see get(1). A diagnostic results if the specified SID is ambiguous, or, if necessary and omitted on the command line.

–s
: Suppresses all messages.

–y[*comment*]
: Creates delta with specified commentary. text A null string is considered a valid *comment.*
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. A period (.) in column one of a newline terminates the *comment* text.

Restrictions

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 and will cause an error. For further information, see sccsfile(5).

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.

Diagnostics

See sccshelp(1) for explanations.

Files

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*.

See Also

admin(1), bdiff(1), cdc(1), get(1), help(1), prs(1), rmdel(1), sccs(1), sccsfile(5)
Guide to the Source Code Control System

Name

deroff – remove formatting codes from text

Syntax

deroff [–w] *file*...

Description

The `deroff` command reads each file in sequence and removes all `nroff` and `troff` command lines, backslash constructions, macro definitions, `eqn` constructs (between .EQ and .EN lines or between delimiters), and table descriptions and writes the remainder on the standard output. The `deroff` command follows chains of included files (.so and .nx commands); if a file has already been included, the .so command is ignored and an .nx command terminates execution. If no input file is given, `deroff` reads from the standard input file.

Options

–w Generates word list (one word per line).

Restrictions

The `deroff` command is not a complete `troff` interpreter, so it can be confused by subtle constructs. Most errors result in too much rather than too little output.

The **–w** flag removes every one– or two–character word.

See Also

diction(1), nroff(1)

Name

df – display free and used disk space

Syntax

df [**–i**] [**–n**] [*filesystem...*] [*file...*]

Description

The df command displays the amount of disk space available on the specified *file system,* for example, /dev/ra0a. It also displays the amount of available disk space on the file system in which the specified *file* is contained, for example, $HOME. If a device is given that has no file systems mounted on it, df displays the information for the root file system. Without any arguments or options, df displays shows all mounted filesystems, including those manually mounted without use of the /etc/fstab file. The numbers are reported in kilobytes.

Unless the **–n** option is specified, df updates the statistics stored in memory for the file system specified, before it returns the information.

Options

–i Also report the number of used and free inodes.

–n Do not update the file system statistics stored in memory. Instead, return whatever statistics are stored in memory. This prevents df from hanging in the event that a server containing the specified file system is down.

Restrictions

You cannot use the df command to find free space on an unmounted file system using the block or character special device name. Instead, use the dumpfs command.

Examples

```
% df
Filesystem      Total    kbytes  kbytes   %
node            kbytes   used    free     used    Mounted on
/dev/ra1a        7429     2085    4602    31%    /tmp
/dev/ra0e       30519    14817   12651    54%    /usr/spool
/dev/ra0h      313233   122858  159052    44%    /usr/staff1
```

The total disk space is the total space that was created during the making of the file system. The addition of the used space, the free space and a percentage of reserved space is the total space. The default value for the reserved space is 10%.

Files

`/etc/fstab`
List of mounted file systems

See Also

getmnt(2), fstab(5), dumpfs(8), icheck(8), mkfs(8), newfs(8), quot(8)

dgate (1c)

Name

dgate – log in to a DECnet remote system through an intermediate ULTRIX DECnet host (gateway system)

Syntax

dgate *host*

Description

The `dgate` command lets you log in from an ULTRIX system without DECnet to a remote system on DECnet (specified by the *host* argument) through an intermediate host, or gateway system: an ULTRIX system attached to DECnet.

The login is accomplished through an intermediate host, or gateway system, to which your system is connected through a local area (TCP/IP) network. The gateway system is specified at the local system in the file `/etc/dgateway`. The gateway system must be connected through DECnet to the ultimate host system that you specify in the `dgate` command.

The `dgate` program scans input for lines beginning with a tilde character (~). A tilde-period line disconnects you from your current `dgate` session. A tilde-CTRL/Z line suspends `dgate` and returns you to the parent process. A tilde-tilde line passes the tilde character on to the remote login session.

Files

`/etc/dgateway`
`~/.dgateway`

See Also

dgateway(5)

Name

diction, explain – print wordy sentences; thesaurus for diction

Syntax

diction [**–ml**] [**–mm**] [**–n**] [**–f** *pfile*] *file...*
explain

Description

The `diction` command 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 `explain` command is an interactive thesaurus for the phrases found by diction.

Options

–mm	Overrides default macro package **–ms**.
–ml	Causes **deroff** to skip lists.
–fpfile	Specifies pattern file in addition to default file. Note that you can specify the **–n** flag to suppress the default file.

Restrictions

Use of non-standard formatting macros may cause incorrect sentence breaks.

See Also

deroff(1)

Name

diff – differential file comparator

Syntax

diff [*options*] *dir1 dir2*
diff [*options*] *file1 file2*

Description

The `diff` command compares the contents of files or groups of files, and lists any differences it finds. When run on regular files, and when comparing text files that 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 can be specified 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 likewise if *file2* is a directory.

If both arguments are directories, `diff` sorts the contents of the directories by name, and then runs the regular file `diff` algorithm on text files that are different. Binary files that differ, common subdirectories, and files that appear in only one directory are listed.

Options

The following options are used when comparing directories:

–l Displays the output in long format. Each text file is piped through `pr(1)` to paginate it; other differences are summarized after all text file differences are reported.

–n Produces a script similar to that of **–e,** but in reverse order and with a count of changed lines on each insert or delete command.

–r Recursively checks files in common subdirectories.

–s Displays names of files that are the same.

–S*name* Starts a directory in the middle beginning with the specified file.

Except for the **–b, i, t, and w** options, which may be given with any of the others, the following formatting options are mutually exclusive:

–b Ignores trailing blanks and other strings of blanks and treats such portions as equal.

–c Displays three context lines with each output line. For backwards compatibility, **–c***n* causes **n** number of context lines.

–C *n* Displays specified number of context lines with each output line. With **–c** or **–C** the output format is modified slightly: the output begins with identification of the files involved and their creation dates and then each change is separated by a line with a dozen asterisks (*). The lines removed from *file1* are marked with minus sign (-); those added to *file2* are marked plus sign (+). Lines that are changed from one file to the other are marked in both files with an exclamation point (!).

Changes within *n* context lines of each other are grouped together in the output. This results in output that is usually much easier to interpret.

–D*string* Causes `diff` to create a merged version of *file1* and *file2* on the standard output. With C preprocessor controls included, a compilation of the result without defining *string* is equivalent to compiling *file1*, while defining *string* will yield *file2*.

–e Writes output to an `ed` script. In connection with **–e**, the following shell program can 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 available. A latest version message appears on the standard output.

```
(shift; cat $*; echo '1,$p') | ed - $1
```

If you specify **–e** when comparing directories the result is a `sh(1)` script for converting text files that are common to the two directories from their state in *dir1* to their state in *dir2*.

–f Writes the output in reverse order to a script.

–h Makes a hasty comparison. It works only when changed portions are short and well separated, but does work on files of unlimited length.

–i Ignores the case of letters. For example 'A' will compare equal to 'a'.

–t Expand tabs in output lines. Normal or **–c** output adds character(s) to the front of each line which may affect the indentation of the original source lines and make the output listing difficult to interpret. This option will preserves the original indentation.

–w Causes whitespace (blanks and tabs) to be totally ignored. For example, 'if (a == b)' will compare equal to 'if(a==b)'.

There are several 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 you can tell 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 a left angle bracket (<). Then all the lines that are affected in the second file are listed, flagged by a right angle bracket (>).

Restrictions

Editing scripts produced under the **–e** or **–f** option have trouble creating lines consisting of a single period (.).

When comparing directories with the **–b, i, t, or w** options specified, `diff` first compares the files as `cmp` does, and then runs the `diff` algorithm if they are not equal. If the only differences are in the blank strings, `diff` may report these as differences.

Diagnostics

Exit status is 0 for no differences, 1 for some differences,and 2 if the specified file cannot be found.

Files

`/tmp/d?????`

`/usr/lib/diffh` for **-h**

`/bin/pr`

See Also

cmp(1), cc(1), comm(1), diff3(1), ed(1)

Name

diff3 – 3-way differential file comparison

Syntax

diff3 [**–ex3**] *file1 file2 file3*

Description

The `diff3` command compares three versions of a file, and publishes the ranges of text that disagree, flagged with the following codes:

====	all three files differ
====1	*file1* is different
====2	*file2* is different
====3	*file3* is different

The type of change needed to convert 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.

Options

–3 Produces an `ed` editor script containing the changes between file1 and file2 that are to be incorporated into file3.

–e Produces an `ed` editor script containing the changes between file2 and file3 that are to be incorporated into file1.

–x Produces an `ed` editor script containing the changes among all three files.

Examples

Under the **–e** option, `diff3` publishes a script for the editor `ed` that incorporates into *file1* all changes between *file2* and *file3* – that is, the changes that would normally be flagged ==== and ====3. Option **–x** (**–3**) produces a script to incorporate only changes flagged ==== (====3). The following command applies the resulting script to 'file1':

```
(cat script; echo '1,$p') | ed - file1
```

Restrictions

Text lines that consist of a single '.' defeat **-e**.

Files

/tmp/d3?????
/usr/lib/diff3

See Also

cmp(1), comm(1), diff(1), dffmk(1), join(1), sccsdiff(1), uniq(1)

Name

diffmk – mark differences between files

Syntax

diffmk *name1 name2 name3*

Description

The `diffmk` command compares two versions of a file and creates a third file that includes "change mark" commands for `nroff` or `troff`. The *name1* and *name2* are the old and new versions of the file. The `diffmk` command generates *name3*, which contains the lines of *name2* plus inserted formatter "change mark" (**.mc**) requests. When *name3* 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 *.

The `diffmk` command can be used to produce listings of C (or other) programs with changes marked. A typical command line for such use is the following:

```
diffmk old.c new.c tmp; nroff macs tmp | pr
```

In this example 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.

If the characters | and * are inappropriate, a copy of `diffmk` can be edited to change them. The `diffmk` command is a shell procedure.

Restrictions

Aesthetic considerations may dictate manual adjustment of some output. File differences involving only formatting requests may produce undesirable output, that is, replacing **.sp** by **.sp 2** will produce a "change mark" on the preceding or following line of output.

See Also

cmp(1), comm(1), diff(1), nroff(1), join(1), sccsdiff(1), troff(1), uniq(1)

dircmp(1)

Name

dircmp – directory comparison

Syntax

dircmp [**–d**] [**–s**] [**–w***n*] *dir...*

Description

The `dircmp` command examines *dir1* and *dir2* and generates tabulated information about the contents of the directories. Listings of files that are unique to each directory are generated for all the options. If no option is entered, a list is output indicating whether the filenames common to both directories have the same contents.

This command is supplied for X/OPEN compliance. The same results are available from `diff`(1), which produces results more quickly and effectively.

Options

–d Compares 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 Suppresses messages about identical files.

–w*n* Changes the width of the output line to *n* characters. The default width is 72.

See Also

cmp(1), diff(1).

Name

dirname – deliver directory names from pathname

Syntax

dirname *string*

Description

The `dirname` command delivers all but the last level of the path name in *string*.

Examples

The following example sets the Bourne shell variable NAME to `/usr/src/cmd`:

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

See Also

basename(1), sh(1), sh5(1), ksh(1)

Name

dis – disassemble an object file

Syntax

dis [-h] [-S] [-p *procedure*] [*file* ...]

Description

The **dis** command disassembles object files into machine instructions. Note that assembler code and machine code can differ depending on the machine type. A *file* can be an object or an archive file.

Options

–h Prints the general register names, rather than the software register names.

–p Disassembles only the specified procedure from the object file.

–S Lists the source listings. Otherwise, only instructions are listed.

Restrictions

You cannot disassemble an archive.

Name

dist – redistribute a message to additional addresses

Syntax

dist [*+folder*] [*msg*] **[–annotate] [–noannotate] [–draftfolder** *+folder*]
[–draftmessage *msg*] **[–nodraftfolder] [–editor** *editorname*] **[–noedit]**
[–form *formfile*] **[–inplace] [–noinplace] [–whatnowproc** *program*]
[–nowhatnowproc] [–help]

Description

Use `dist` to redistribute the current message to addresses that are not on the original address list.

The program `dist` is similar to `forw`. The main difference between the two commands is that `forw` encapsulates the message, whereas `dist` merely resends it. This is manifested in the scan listing of the messages. A message that has been forwarded will appear to have been sent by the person who forwarded the message. A message that has been redistributed using `dist` will appear to have come from the sender of the original message. In the following example, messages one and two are identical apart from the method used to send the message on to additional recipients.

```
1    20/06 goodman   ULTRIX <<As you will see from the attached
2+   21/06 John      As previous, but forwarded <<------- Forwa
```

When you use `dist`, you will get a message form to fill in with the details of the additional recipients. The default message form contains the following elements:

```
Resent-To:
Resent-cc:
```

You can only put recognized header lines in this message form. The `dist` program recognizes addresses in the following fields:

```
Resent-To:
Resent-cc:
Resent-Bcc:
Resent-Fcc: folder
```

The `Resent-Fcc:` field will be honored only if you have a corresponding `Resent-Fcc:` folder set up in your `mh_profile` (see send(1mh)). The headers and the body of the original message are copied to the draft when the message is sent.

If the file named `distcomps` exists in your MH directory, it is used instead of the standard mail header. In either case, the file specified by `-form` *formfile* is used if given.

If the draft already exists, `dist` asks you what you want to do with the existing draft. A reply of `quit` aborts `dist`, leaving the draft intact; `replace` replaces the existing draft with a blank skeleton; and `list` displays the draft.

Options

If the -annotate option is used, the message being distributed is annotated with the lines:

```
Resent: date
Resent: addrs
```

where each address list contains as many lines as required. This annotation is done only if the message is sent directly from dist. If the message is not sent immediately from dist, comp -use may be used to re-edit and send the constructed message, but the annotations do not take place. The -inplace option causes annotation to be done in place in order to preserve links to the annotated message.

The -editor and -noedit switches allow you to specify an editor of your choice; or to suppress the editor entirely.

Note that while in the editor, the message being resent is available through a link named @ (assuming the default whatnowproc). In addition, the actual pathname of the message is stored in the environment variable $editalt, and the pathname of the folder containing the message is stored in the variable $mhfolder.

The dist command normally creates the draft of the message in the draft file, or in the +drafts folder if you have one set up. The **–draftfolder** *+folder* and **–draftmessage** *filename* options allow you to create draft messages in alternative locations. See comp(1mh) for more details.

When you exit from the editor, dist invokes the whatnow program. See whatnow(1mh) for details of the available options. The invocation of this program can be inhibited by using the -nowhatnowproc switch. However the whatnow program starts the initial edit, hence, -nowhatnowproc prevents any edit from occurring.

The dist command does not rigorously check the message being distributed for adherence to the transport standard, but post called by send does.

The post program will not deliver poorly formatted messages, and dist will not correct things for you.

If whatnowproc is whatnow, then dist uses the built-in whatnow, program. However, it does not actually run the whatnow program. Hence, if you define your own whatnowproc, do not call it whatnow since dist will not run it.

If your current working directory is not writable, the link named @ is not available.

Context

If a folder is given, it will become the current folder. The message distributed will become the current message. Dist originally used headers of the form Distribute-xxx: instead of Resent-xxx:. In order to conform with the ARPA Internet standard, RFC-822, the Resent-xxx: form is now used. Dist will recognize distribute-xxx: type headers and automatically convert them to Resent -xxx. The defaults for dist are:

+foldername defaults to the current folder
msg defaults to cur
-noannotate
-nodraftfolder

-noinplace

Files

/usr/new/lib/mh/distcomps	The message skeleton
<mh-dir>/distcomps	Alternative to the standard skeleton
$HOME/.mh_profile	The user profile
<mh-dir>/draft	The draft file

Profile Components

Path:	To determine your MH directory (mh-dir)
Current-Folder:	To find the default current folder
Draft-Folder:	To find the default draft-folder
Editor:	To override the default editor
fileproc:	Program to refile the message
whatnowproc:	Program to ask the What now? questions

See Also

comp(1mh), forw(1mh), repl(1mh), send(1mh), whatnow(1mh)

domainname(1yp)

Name

domainname – display or set the name of the current domain for this system

Syntax

domainname [*nameofdomain*]

Description

The domainname command, when used without an argument, displays the name of the current domain. The /etc/rc.local startup script must be used to set the current domain name before any other YP commands can be issued.

A domain is a logical grouping of networked-connected systems established for the purpose of sharing a common set of data files. Domains are only used by the yellow pages (YP) service and are called YP domains. A YP domain is a directory in /etc/yp, established through the use of the domainname command, where a YP server holds all of the YP maps. Each YP map contains a set of keys and associated key values. For example, in a map called hosts.byname, the host names stored there constitute the keys. The corresponding internet addresses of each host constitute the associated key values.

See Also

ypfiles(5yp), ypsetup(8yp)

Name

dtoc – unpack objects from a DOTS file

Syntax

dtoc [**–f**] [**–p**] [[*object.dots*] *directory*]

Description

The `dtoc` command unpacks the contents of a Data Object Transport Syntax (DOTS) file or standard input.

object.dots can be either a file name, or a minus sign (–). If a minus sign (–) is specified, or if no file name is present, `dtoc` reads from the standard input. If *directory* is specified, the contents of the DOTS input is unpacked and stored in the specified directory. If *directory* is not specified, the content of the DOTS input is unpacked into the current directory. The names of the files created are written to standard output.

A DOTS file may contain a data object which consists of more than one component. Therefore, it is possible that more than one output file may be generated. As the object is unpacked, duplicate file or directory names may be encountered. If a duplicate is encountered, a new output file is generated with a sequential number appended to its name. For example, if `dtoc` discovers an existing file `foo.ddif` during unpacking, `foo.ddif.1` is created.

As an object is unpacked, the external references within each object component are updated. Because DOTS files may have originated from non-ULTRIX systems, names of components may be modified as components are unpacked. References to those renamed components are updated accordingly.

Options

–f Suppresses output of unpacked file names.

–p Causes only the name of the primary input file to be written to standard output.

Implementation

Standard Input

If a minus sign (–) is specified, or if no parameters are specified, *standard input* is read until a <CTRL/D> or **EOF** (end of file) is read. It cannot be specified more than once. The contents of *standard input* must conform to the syntax of a single DOTS file.

Reconstitution Of Names

Object file names and file names of referenced components may be modified as objects are extracted or unpacked. If names are modified, the references in the unpacked objects are updated. The handling of names depends in part on the name-type of the object, as follows:

ULTRIX file names

Names are unmodified.

VMS file names

The set of rules is as follows:

Convert uppercase letters to lower case.

Convert dollar signs ($) to underscores (_) because dollar signs have meaning on ULTRIX systems.

Ignore disk volume and directory specifications, if they are present, because they are not likely to be meaningful on ULTRIX systems.

Append duplicate file names with a period and a unique number.

Leave all other characters alone.

Restrictions

A DOTS file is expected to contain only a single primary DDIF or DTIF object in this release. Any subsequent objects in the DOTS file are external references of the primary object.

Diagnostics

The exit status is 0 if all objects were unpacked successfully, and 1 if any of the objects could not be unpacked. Consult *standard error* to see what failed, and why.

If a nonexistent target directory is specified, `dtoc` returns error status.

See Also

ctod(1), DDIF(5), DTIF(5), DOTS(5)

Name

du – print amount of disk usage

Syntax

du [–as] [*name...*]

Description

The du command gives the number of kilobytes contained in all files and, recursively, directories within each specified directory or file *name*. If *name* is missing, '.' is used.

Absence of either **–a** or **–s** causes an entry to be generated for each directory only.

A file that has two links to it is only counted once.

Options

–a Displays the disk usage for each file.

–s Displays a summary total only.

Restrictions

Non-directories given as arguments (not under **–a** option) are not listed.

If there are too many distinct linked files, du counts the excess files multiply.

See Also

df(1), quot(8)

Name

echo – echo arguments

Syntax

echo [**–n**] [*arg...*]

Description

The `echo` command writes its arguments separated by blanks and terminated by a new line on the standard output.

Options

–n Suppresses newlines from output.

Examples

The `echo` command is useful for producing diagnostics in shell programs and for writing constant data on pipes.

To send diagnostics to the standard error file, type the following:

```
echo ... 1>&2
```

Name

echo – echo arguments

Syntax

echo [arg] ...

Description

The `echo` command writes its arguments separated by blanks and terminated by a new-line on the standard output. It also understands C-like escape conventions; however, beware of conflicts with the shell's use of the backslash (\) character:

\b	backspace
\c	print line without new-line
\f	form-feed
\n	new-line
\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.

The `echo` is useful for producing diagnostics in command files and for sending known data into a pipe.

See Also

sh5(1)

Name

ed, red – text editor

Syntax

ed [–] [–**p***string*] [–**x**] [*file*]

red [–] [–**x**] [*file*]

Description

The ed text editor is the standard text editor. If you give the *file* argument, 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 – option 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** option allows you to specify a prompt string.

NOTE

The –**x** option is available only if the Encryption layered product is installed.

If you supply the –**x** option, an **x** command is simulated first to handle an encrypted file. The ed text editor operates on a copy of the file it is editing; changes made to the copy have no effect on the file until you give a **w** (write) command. The copy of the text being edited resides in a temporary file called the *buffer*. There is only one buffer.

The red text editor is a restricted version of ed. It allows editing of files only in the current directory, and prohibits executing shell commands with !*shell command*. Attempts to bypass these restrictions result in an error message *(restricted shell).*

NOTE

When you enter text, tab characters 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, possibly followed by 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 can frequently be omitted.

In general, only one command appears 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 input mode, no commands are recognized; all input is merely collected. Input mode is exited by typing a period (.) alone at the beginning of a line.

The ed text editor supports a limited form of *regular expression* notation; regular expressions are used in addresses to specify lines and in some commands (for example, *s*) to specify portions of a line that are to be substituted. A regular expression (RE) 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:

- An ordinary character (*not* one of those discussed below) is a one-character RE that matches itself.
- 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 ([ ]).
  b. ^ (caret or circumflex), which is special at the *beginning* of an *entire* RE (see below), or when it immediately follows the left of a pair of square brackets ([ ]) (see below).
  c. $ (currency symbol), which is special at the *end* of an entire RE (see below).
  d. The character used to bound (that is, delimit) an entire RE, which is special for that RE (for example, see how slash (/) is used in the **g** command, below.)
- A period (.) is a one-character RE that matches any character except new-line.
- A non-empty 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* new-line 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). For example, [ ]**a–f]** matches either a right square bracket (]) or one of the letters **a** through **f** inclusive. The four characters listed in a above stand for themselves within such a string of characters.

The following rules may be used to construct *RE*s from one-character REs:

- A one-character RE is a RE that matches whatever the one-character RE matches.
- 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.
- 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 non-negative 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.
- The concatenation of REs is a RE that matches the concatenation of the strings matched by each component of the RE.

- A RE enclosed between the character sequences \(and \) is a RE that matches whatever the unadorned RE matches.
- 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):

- A circumflex (∧) at the beginning of an entire RE constrains that RE to match an *initial* segment of a line.
- 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 (for example, //) is equivalent to the last RE encountered. See also the last 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 lower-case letter. Lines are marked with the **k** command described below.
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 also the last 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. For example, **–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. However, any command (except **e**, **f**, **r**, or **w**) may be suffixed by **l**, **n** or **p**, 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 new line 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 file name, if any, is used (see the **f** command). The number of characters read is typed; *file* is remembered for possible use as a default file name 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 file name. See also DIAGNOSTICS below.

E *file*

The `edit` 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, the file-name command changes the currently-remembered file name to *file*; otherwise, it prints the currently-remembered file name.

(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. The **g, G, v,** and **V** commands are *not* permitted in the *command list*. See also RESTRICTIONS and the last paragraph before FILES below.

(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 new-line 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 DEL or BREAK).

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 new line character).

(.,.+1)j

The join command joins contiguous lines by removing the appropriate new-line characters. If exactly one address is given, this command does nothing.

(.)k*x*

The mark command marks the addressed line with name *x*, which must be a lower-case letter. The address *'x* then addresses this line; **.** is unchanged.

(.,.)l

The list command prints the addressed lines in an unambiguous way: a few non-printing characters (for example, *tab, backspace*) are represented by (hopefully) mnemonic overstrikes, all other non-printing 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*

The B. move command repositions the addressed line(s) 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

The number command 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

The print command prints 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

The quit command causes `ed` to exit. No automatic write of a file is done (but see DIAGNOSTICS below).

Q

The editor exits 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 file name is given, the currently-remembered file name, if any, is used (see **e** and **f** commands). The currently-remembered file name is *not* changed unless *file* is the very first file name 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 file name.

(. , .)s/*RE***/***replacement***/** or
(. , .)s/*RE***/***replacement***/g**

The substitute command searches each addressed line for an occurrence of the specified RE. In each line in which a match is found, all (non-overlapped) 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. It is an error for the substitution to fail on *all* addressed lines. Any character other than space or new-line may be used instead of / to delimit the RE and the *replacement;* **.** is left at the last line on which a substitution occurred. See also the last 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 new-line character into it. The new-line 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*

This command acts just like the **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

The **undo** command nullifies the effect of 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*

The write command writes 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 file name is *not* changed unless *file* is the very first file name mentioned since ed was invoked. If no file name is given, the currently-remembered file name, 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 file name.

($)=

The line number of the addressed line is typed; **.** is unchanged by this command.

!*shell command*

The remainder of the line after the **!** is sent to the UNIX System shell (*sh*(1)) to be interpreted as a command. Within the text of that command, the unescaped character **%** is replaced with the remembered file name; 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**)<new-line>

An address alone on a line causes the addressed line to be printed. A new-line alone is equivalent to **.+1p**; it is useful for stepping forward through the buffer.

If an interrupt signal (ASCII DEL or BREAK) 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 new-line. Files (for example, **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 (for example, /) would be the last character before a new-line, 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?
```

Restrictions

A **!** command cannot be subject to a **g** or a **v** command.
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. For further information, see sh(1).
The sequence **\n** in a RE does not match a new-line character.
The **l** command mishandles DEL.

Diagnostics

?	for command errors.
?*file*	for an inaccessible file.
	(use the help and Help 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 at this point will take effect. The – command-line option inhibits this feature.

Files

/tmp/e#	temporary; # is the process number.
ed.hup	work is saved here if the terminal is hung up.

See Also

grep(1), sed(1), sh(1), stty(1)

Name

env – set environment for command execution

Syntax

env [–] [*name*=**value**] ... [*command args*]

Description

The `env` command obtains the current *environment*, modifies it according to its arguments, then executes the command with the modified environment. Arguments of the form *name=value* are merged into the inherited environment before the command is executed. The – flag 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.

See Also

sh(1), environ(5)

Name

error – analyze and disperse compiler error messages

Syntax

error [**–n**] [**–s**] [**–q**] [**–v**] [**–t** *suffixlist*] [**–I** *ignorefile*] [*name*]

Description

The `error` command analyzes and optionally disperses the diagnostic error messages produced by a number of compilers and language processors to the source file and line where the errors occurred. It permits error messages and source code to be viewed simultaneously without using multiple windows in a screen editor.

The `error` command looks at the error messages, either from the specified file *name* or from the standard input. It attempts to determine the following: which language processor produced each error message, to which source file and line number the error message refers, and if the error message is to be ignored or not. It also inserts the error message into the source file as a comment on the line preceding the one where the error occurred.

Error messages that cannot be categorized by language processor or content are not inserted into any file, but are sent to the standard output. The `error` command touches source files only after all input has been read. By specifying the **–q** query option, the user is asked to confirm any potentially dangerous (such as touching a file) or verbose action.

If the **–t** touch option and associated suffix list is given, `error` restricts itself to touching only those files with suffixes in the suffix list. Error also can be asked (by specifying **–v**) to invoke `vi(1)` on the files in which error messages were inserted; this prevents the need to remember the names of the files with errors.

The `error` command is intended to be run with its standard input connected via a pipe to the error message source. Some language processors put error messages on their standard error file; others put their messages on the standard output. Hence, both error sources should be piped together into `error`. For example, when using the *csh* syntax,

```
make -s lint |& error -q -v
```

analyzes all the error messages produced by whatever programs `make` runs when making lint.

The `error` command knows about the error messages produced by the following: `make, cc, cpp, ccom, as, ld, lint, pi, pc` and `f77`. The `error` command knows a standard format for error messages produced by the language processors, so it is sensitive to changes in these formats. For all languages except *Pascal,* error messages are restricted to be on one line. Some error messages refer to more than one line in more than one file. The `error` command duplicates the error message and inserts it at all of the places referenced.

The `error` command does one of six things with error messages.

synchronize — Some language processors produce short errors describing which file it is processing. The `error` command uses these to determine the file name for languages that don't include

the file name in each error message. These synchronization messages are consumed entirely by `error`.

discard
: Error messages from `lint` that refer to one of the two `lint` libraries, /usr/lib/llib-lc and /usr/lib/llib-port are discarded, to prevent accidently touching these libraries. Again, these error messages are consumed entirely by `error`.

nullify
: Error messages from `lint` can be nullified if they refer to a specific function, which is known to generate diagnostics which are not interesting. Nullified error messages are not inserted into the source file, but are written to the standard output. The names of functions to ignore are taken from either the file named .errorrc in the users's home directory, or from the file named by the **–I** option. If the file does not exist, no error messages are nullified. If the file does exist, there must be one function name per line.

not file specific
: Error messages that can't be discerned are grouped together, and written to the standard output before any files are touched. They will not be inserted into any source file.

file specific
: Error message that refer to a specific file, but to no specific line, are written to the standard output when that file is touched.

true errors
: Error messages that can be intuited are candidates for insertion into the file to which they refer.

Only true error messages are candidates for inserting into the file they refer to. Other error messages are consumed entirely by `error` or are written to the standard output. The `error` command inserts the error messages into the source file on the line preceding the line the language processor found in error. Each error message is turned into a one line comment for the language, and is internally flagged with the string "###" at the beginning of the error, and "%%%" at the end of the error. This makes pattern searching for errors easier with an editor, and allows the messages to be easily removed.

In addition, each error message contains the source line number for the line to which the message refers. A reasonably formatted source program can be recompiled with the error messages still in it, without having the error messages themselves cause future errors. For poorly formatted source programs in free format languages, such as C or Pascal, it is possible to insert a comment into another comment, which can wreak havoc with a future compilation. To avoid this, programs with comments and source on the same line should be formatted so that language statements appear before comments.

The `error` command catches interrupt and terminate signals, and if in the insertion phase, terminates what it is doing.

Options

Options available with `error` are the following:

–I *ignorefile*
: Ignore the functions listed in the specified file (next argument).

–n Does not touch files and sends error messages to the standard output.

–q Prompts before touching the source file. A ''y'' or ''n'' to the question is necessary to continue. Absence of the **–q** option implies that all referenced files (except those referring to discarded error messages) are to be touched.

–S Shows error in unsorted order from the error file.

–s Displays *statistics* for each error type.

–T Terse output.

–t *suffixlist* Does not touch those files that match the specified suffix. The suffix list is dot separated, and ''*'' wildcards work. Thus the suffix list:

".c.y.foo*.h"

allows **error** to touch files ending with ''.c'', ''.y'', ''.foo*'' and ''.h''.

–v Invokes the `vi` editor on each file that had been touched.

Restrictions

Opens the teletype directly to do user querying.

Source files with links make a new copy of the file with only one link to it.

Changing a language processor's format of error messages may cause `error` to not understand the error message.

The `error` command, since it is purely mechanical, does not filter out subsequent errors caused by 'floodgating' initiated by one syntactically trivial error.

Pascal error messages belong after the lines affected (error puts them before). The alignment of the '|' marking the point of error is also disturbed by `error`.

The `error` command was designed for work on CRT's at reasonably high speed. It does not work as well on slow speed terminals, and has never been used on hard-copy terminals.

Files

`~/.errorrc` function names to ignore for *lint* error messages

`/dev/tty` user's teletype

Name

ex, edit – text editor

Syntax

ex [–] [**–v**] [**–x**] [**–t** tag] [**–r**] [*+command*] [**–l**] *name...*
edit [ex options]

Description

The ex editor is the root of a family of editors: edit, ex and vi. The ex editor is a superset of ed, with the most notable extension being a display-editing facility. Display-based editing is the focus of vi.

The *name* argument indicates the files to be edited.

Options

– Suppresses all interactive-user feedback. This option is useful in processing editor scripts in command files.

–v Equivalent to using vi rather than ex.

–t Equivalent to an initial *tag* command, that is, editing the file containing the *tag* and positioning the editor at its definition.

–r Used to recover after an editor or system crash. It recovers by retrieving the last saved version of the named *file*. If no *file* is specified, it displays a list of saved files.

–R Sets the read-only option at the start.

+command
Indicates that the editor should begin by executing the specified command. If the command is omitted, it defaults to **$**, positioning the editor at the last line of the first file, initially. Other useful commands here are scanning patterns of the form *+/pattern* or line numbers.

–l Sets up for LISP. That is, it sets the *showmatch* and *lisp* options.

NOTE

The **–x** option is available only if the Encryption layered product is installed.

–x Causes ex to prompt for a key. The key is used to encrypt and decrypt the contents of the file. If the file contents have been encrypted with one key, you must use the same key to decrypt them.

Restrictions

The undo command causes all marks to be lost on lines changed and then restored if the marked lines were changed.

The undo command does not clear the buffer modified condition.

The *z* command prints a number of logical rather than physical lines. More than a screenful of output may result if long lines are present.

File input/output errors does not print a name if the command line minus sign (–) option is used.

There is no easy way to do a single scan ignoring case.

The editor does not warn you if you place text in named buffers and do not use it before exiting the editor.

Null characters are discarded from input files, and cannot appear in output files.

Files

/usr/lib/ex?.?recover	recover command
/usr/lib/ex?.?preserve	preserve command
/etc/termcap	terminal capabilities
~/.exrc	editor startup file
/tmp/Ex*nnnnn*	editor temporary
/tmp/Rx*nnnnn*	named buffer temporary
/usr/preserve	preservation directory

See Also

awk(1), ed(1), grep(1), sed(1), vi(1), termcap(5), environ(7)
Edit: A Tutorial and the *Ex Reference Manual*
ULTRIX Supplementary Documents Vol. I: General User

Name

expand, unexpand – expand tabs to spaces, and vice versa

Syntax

expand [–*tabstop*] [–*tabn...*] [*file...*]
unexpand **[–a]** [*file...*]

Description

The expand command processes the named files or the standard input writing the standard output with tabs changed into blanks. Backspace characters are preserved into the output and decrement the column count for tab calculations. The expand command is useful for pre-processing character files (before sorting, looking at specific columns, and so forth) that contain tabs.

If a single *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.

The unexpand command 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** option is given, then tabs are inserted whenever they would compress the resultant file by replacing two or more characters.

Options

–# Sets tabstops the specified number of spaces (#) apart.

–a When used with unexpand, compresses file by inserting tabs for two or more spaces.

Name

expr – evaluate expressions

Syntax

expr *arg...*

Description

The arguments are taken as an expression. After evaluation, the result is written on the standard output. Each token of the expression is a separate argument.

The operators and keywords are listed below. The list is in order of increasing precedence, with equal precedence operators grouped.

expr | *expr* — Yields the first *expr* if it is neither null nor 0. Otherwise yields the second *expr*.

expr & *expr* — Yields the first *expr* if neither *expr* is null or 0. Otherwise yields 0.

expr relop expr — The *relop* is one of < <= = != >= > and yields 1 if the indicated comparison is true, '0' if false. The comparison is numeric if both *expr* are integers, otherwise lexicographic.

expr + *expr*
expr - *expr* — Yields addition or subtraction of the arguments.

expr * *expr*
expr / *expr*
expr % *expr* — Yields multiplication, division, or remainder of the arguments.

expr : *expr* — The matching operator compares the string first argument with the regular expression second argument; regular expression syntax is the same as that of `ed(1)`. The `\(...\)` pattern symbols can be used to select a portion of the first argument. Otherwise, the matching operator yields the number of characters matched ('0' on failure).

(*expr*) — parentheses for grouping.

Examples

The first example adds 1 to the Shell variable *a*:

```
a=`expr $a + 1`
```

The second example finds the file name part (least significant part) of the pathname stored in variable *a*,

```
expr $a : '.*/\(.*\)' '|' $a
```

Note the quoted Shell metacharacters.

Diagnostics

The `expr` command returns the following exit codes:

0 The expression is neither null nor '0'.

1 The expression is null or '0'.

2 The expression is invalid.

See Also

ed(1), sh(1), test(1)

Name

extract – interactive string extract and replace

Syntax

extract [**–i** *ignorefile*] [**–m** *prefix*] [**–n**] [**–p** *patternfile*] [**–s** *string*]
[**–u**] *source-program...*

Description

The extract command interactively extracts text strings from source programs. The extract command replaces the strings it extracts with calls to the catgets function. The command also writes the string it extracts to a source message catalog. You use this command to replace hard-coded messages in your program source file with calls to the catgets command and create a source message catalog. At run time, the program reads the message text from the message catalog. By storing messages in a message catalog, instead of in your program, you allow the text of messages to be translated to a new language or modified without the source program being changed.

In the *source-program* argument, you name one or more source programs from which you want messages extracted. The extract command does not extract messages from source programs included using the #include directive. Therefore, you might want to name a source program and all the source programs it includes on a single extract command line.

You can create a patterns file (as specified by (*patternfile*) to control how the extract command extracts and replaces text. The patterns file is divided into several sections, each of which is identified by a keyword. The keyword must start at the beginning of a new line, and its first character must be a dollar sign ($). Following the identifier, you specify a number of patterns. Each pattern begins on a new line and follows the regular expression syntax you use in the regex(3) routine. For more information on the patterns file, see the patterns(5int) reference page.

In addition to the patterns file, you can create a file that indicates strings that extract ignores. Each line in this ignore file contains a single string to be ignored that follows the syntax of the regex(3) routine.

When you invoke the extract command, it reads the patterns file and the file that contains strings it ignores. You can specify a patterns file and an ignore file on the extract command line. Otherwise, the extract command matches all strings and uses a default patterns file.

When you run extract, it displays three windows on your terminal. The first window contains the program source code. The string that matches a string in the patterns file is displayed in reverse video.

The second window displays the contents of the source message catalog that the extract command is creating.

The third window contains a list of the commands that are available. The extract command displays the current command in reverse video. You can execute the current command by pressing the RETURN key. Select another command by typing the first letter in the command name and pressing the RETURN key. The extract command is not sensitive to the case of letters, so you can use uppercase or

lowercase letters to issue commands.

You can use the following commands to control how `extract` treats the string displayed in the first window:

EXTRACT Extract the string into the catalog file and rewrite the source using the rewrite string in the patterns file.

DUPLICATE If the string has been encountered previously, rewrite the source program using the same message number as before. The `extract` command need not add the message to the source message catalog again, so this command saves space in catalogs.

IGNORE Ignore this and all subsequent occurrences of this string during this interactive session. This command does not add the string to the ignore file.

PASS Pass by (ignore) this occurrence of this particular string.

ADD Ignore this and all subsequent occurrences of this string during this interactive session. Add the string to the ignore file.

COMMENT Add the comment you enter to the source message catalog. The `extract` command prompts you to be sure the comment you entered is correct. You answer the prompt by typing ''y,'' n, or q, without pressing the RETURN key.

QUIT Quit from the interactive session. The `extract` command prompts you to be sure you want to quit. Answer ''y'' or ''n'' to the prompt, without pressing the return key.

The output files that `extract` creates up to this point are not removed by this command. However, the files contain only the result of the string extractions that occurred before you issued the QUIT command.

HELP Display a description of all the `extract` commands.

The `extract` command creates to files in your current working directory. The command creates a new version of the source program that contains calls to the `catgets` function, instead of hard-coded messages. The new version of the source program has the same name as the input source program, with the prefix ''nl_''. For example, if the input source program is named `update.c`, the output source program is named `nl_update.c`

In addition to a new source program, the `extract` command creates a source message catalog. The source message catalog contains the text for each message extracted from your input source program. The `extract` command names the file by appending ''.msf'' to the name of the input source program. For example, the source message catalog for the `update.c` source program is named `update.msf`. You can use the source message catalog as input to the `gencat` command.

Options

-i Ignore text strings specified in *ignorefile* . By default, the `extract` command searches for *ignorefile* in the current working directory, your home directory, and `/usr/lib/intln`.

If you omit the `-i` option, `extract` recognizes all strings specified in the patterns file.

-m Add *prefix* to message numbers in the output source program and source message catalog. You can use this prefix as a mnemonic. You must process source message catalogs that contain message number prefixes using the `gencat -h` option.

-n Create a new source message catalog for each input source program. By default, if you specify more than one input source program on the `extract` command line, the command creates one source message catalog for all the input source programs.

-p Use *patternfile* to match strings in the input source program. By default, the command searches for the pattern file in the current directory, your home directory and finally `/usr/lib/intln`.

If you omit the `-p` option, the `extract` command uses a default patterns file that is stored in `/usr/lib/intln/patterns`.

-s Write *string* at the top of the source message catalog. If you omit the `-s` option, `extract` uses the string specified in the `$CATHEAD` section of the patterns file.

-u Use a `message` file produced by a previous run of `strextract`. This file contains details of all the strings which matched the pattern file along with file offsets and line numbers. By default `strextract` is run and its output is used to drive `extract`.

Restrictions

Given the current syntax of the patterns file, you cannot cause `extract` to ignore strings in comments that are longer than one line.

You can specify only one rewrite string for all classes of pattern matches.

The `extract` command does not extract strings from files you include with the `#include` directive. You must run the `extract` commands on these files separately.

Your terminal screen must contain at least 80 columns and 24 lines for `extract` to display its three windows.

The `extract` command does not recognize strings that extend beyond one line.

Examples

The following example shows the commands you issue to run the `extract` command, create a message catalog from the source message catalog, and compile the output source program:

```
% extract -i newignore -p c_patterns remove.c
% gencat remove.cat remove.msf
% vi nl_remove.c
% cc nl_remove.c
```

In this example, the `extract` command uses the `newignore` file to determine which strings to ignore. The command uses the `c_patterns` file to determines which strings to match. The input source program is named `remove.c`.

In response to this command, `extract` creates the source message catalog `remove.msf` and the output source program `nl_remove.c`.

You must edit `nl_remove.c` to include the appropriate `catopen` and `catclose` function calls.

The `gencat` command creates a message catalog and the `cc` command creates an executable program.

See Also

intro(3int), gencat(1int), strextract(1int), strmerge(1int), regex(3), catopen(3int), catgets(3int), patterns(5int)
Guide to Developing International Software

eyacc (1)

Name

eyacc – modified yacc allowing much improved error recovery

Syntax

eyacc [–**v**] [*grammar*]

Description

The eyacc command is an old version of yacc(1), which produces tables used by the Pascal system and its error recovery routines. The eyacc command fully enumerates test actions in its parser when an error token is in the look-ahead set. This prevents the parser from making undesirable reductions when an error occurs before the error is detected. The table format is different in eyacc than it was in the old yacc, as minor changes had been made for efficiency reasons.

See Also

yacc(1)

Name

file – determine file type

Syntax

file [**–c**] [**–f** ffile] [**–m** mfile] *filename* ...

Description

The `file` command performs a series of tests on each *filename* argument in an attempt to classify it. If an argument appears to be ASCII, the `file` command examines the first 1024 bytes and tries to guess its language.

For character special files, part of this classification is information about which devices the system shows as active. In particular, device-specific information such as controller type and unit, device type and unit, and status (offline, write locked, density, errors) is returned. The general categories currently implemented are disk, tape, and terminal devices. The supported terminal devices include Local Area Terminals (LAT) but not Local Area Network (LAN) pseudo-terminals.

The `file` command uses the file `/usr/lib/file/magic` to identify files that have some sort of *magic number*. A *magic number* is any numeric or string constant that identifies the file containing the constant. Commentary at the beginning of `/usr/lib/file/magic` explains its format.

Options

–c Checks the magic file for format errors by printing the internal representation of the magic file. No file typing is done under **–c.**

–f Interprets the following argument to be a file containing the names of the files to be examined.

–m Instructs *file* to use an alternate magic file.

Restrictions

It often does a poor job of distinguishing C programs, shell scripts, English text, and ASCII text.

It does not recognize many programming languages, including Modula, Pascal, and Lisp.

Files

/usr/lib/file/magic

See Also

magic(5)

Name

find – find files

Syntax

find [*options*] *pathname-list expression*

Description

The `find` command 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. 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*.

Options

–atime *n* Tests true if the file has been accessed in *n* days.

–cpio *output* Writes current file on *output* in the format (5120-byte records) specified in the `cpio`(5) reference page. The *output* can be either a file or tape device. If *output* is a tape device the `cpio` B key must be used to read data from the tape.

–ctime *n* Tests true if the file has been changed in *n* days.

–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 (that is, postorder instead of preorder). This can be useful when `find` is used with `cpio` to transfer files that are contained in directories without write permission.

–exec command Tests true if specified command returns a 0 on exit. The end of the command must be punctuated by an escaped semicolon. A command argument '{}' is replaced by the current pathname.

–group gname Tests true if group ID matches specified group name.

–inum *n* Tests true if the file has inode number *n*.

–links *n* Tests true if the file has *n* links.

–mount Tests true if the current file is on the same file system as the current starting pathname.

–mtime *n* Tests true if the file has been modified in *n* days.

–name filename Tests true if the *filename* argument matches the current file name. Normal Shell argument syntax may be used if escaped (watch out for '[', '?' and '*').

–newer file Tests true if the current file has been modified more recently than the argument *file*.

–ok command Executes specified command on standard output, then standard input is read and command executed only upon response **y**.

–perm onum Tests true if file has specified octal number. For further

information, see chmod(1). If *onum* is prefixed by a minus sign, more flag bits (017777) become significant and the flags are compared: *(flags&onum)==onum*. For further information, see stat(2).

–print Prints current pathname.

–size *n* Tests true if the file is *n* blocks long (512 bytes per block).

–type *c* Tests true if file is *c* type (*c* = **b,** block special file: **c,** character special file: **d,** directory: **f,** plain file: **l,** symbolic link: **p,** type port: **s,** type socket).

–user uname Tests true if file owner is login name or numeric user ID.

The primaries may be combined using the following operators (in order of decreasing precedence):

1) A parenthesized group of primaries and operators (parentheses are special to the Shell and must be escaped).
2) The negation of a primary ('!' is the unary *not* operator).
3) Concatenation of primaries (the *and* operation is implied by the juxtaposition of two primaries).
4) Alternation of primaries ('**–o**' is the *or* operator).

Examples

To remove all files named 'a.out' or '*.o' that have not been accessed for a week:

```
find / \( -name a.out -o -name '*.o' \) \
-atime +7 -exec rm {} \;
```

To find all files on the root file system type:

```
find / -mount -print
```

To write all the files on the root file system to tape:

```
find / -mount -print -cpio /dev/rmt?h
cpio -iBvt < /dev/rmt?h
```

To find all the mount points on the root file system type:

```
find / ! -mount -print
```

Files

/etc/passwd
/etc/group

See Also

cpio(1), cpio(5), sh(1), test(1), fs(5)

Name

finger – print user finger information

Syntax

finger [*options*] [*name...*]

Description

By default, `finger` lists the login name, full name, terminal name and write status, idle time, login time, and office location and phone number for each current ULTRIX user. Terminal write status is noted as an asterisk (*) before the terminal name if write permission is denied. Idle time is given in minutes if the listing shows a single integer, hours and minutes if a colon (:) 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. (Account names as well as first and last names of users are accepted.) This format is multi-line, and includes all the information described above as well as the user's home directory and login shell. Additionally, it displays the information contained in the files .plan and .project, both of which are located in the user's home directory. If no list is given, all the people currently logged in are shown.

The `finger` command may be used to look up users on a remote machine. The format is to specify the user as user@host. If the user name is left off, the standard format listing is provided on the remote machine.

Options

–b Displays a briefer long form list of users.

–f Disables printing of headers for short and quick outputs.

–h Suppresses printing of the .project file.

–i Displays list of users with idle times.

–l Displays output in long format.

–m Matches arguments only on user name.

–p Suppresses printing of the .plan file.

–q Displays list of users.

–s Displays output in short format.

–w Displays narrow short format of specified users.

Restrictions

Only the first line of the .project file is printed.

The user's `.plan` or `.project` file cannot be a link to another file. If either of these files is something other than a regular file, it will be ignored.

Files

`/etc/utmp`	Who file
`/etc/passwd`	User information
`/usr/adm/lastlog`	Last login times
`~/.plan`	Plans
`~/.project`	Projects

See Also

chfn(1), w(1), who(1), fingerd(8c)

Name

fmt – simple text formatter

Syntax

fmt [*name*...]

Description

The fmt command 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.

The fmt command 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 (for example, vi) the command

```
!}fmt
```

will reformat a paragraph, evening the lines.

Restrictions

The program was designed to be simple and fast – for more complex operations, the standard text processors are likely to be more appropriate.

See Also

mail(1), nroff(1)

Name

fold – fold long lines for finite width output device

Syntax

fold [**–b**] [**–s**] [**–w** *width* or *–width*] [*file...*]

Description

The `fold` command is a filter which folds the contents of each specified *file*, or the standard input if no *file* is specified, breaking the lines to have maximum width *width*.

Options

–b Causes each '<backspace>' in a line to be interpreted as decrementing the line length by one.

–s Breaks the line on the last <blank> character found before the specified length. If none are found the line breaks at the specified length.

–w *width* or *–width*
Specify the maximum line width, in bytes. The default value is 80. The *width* should be a multiple of 8 if tabs are present, or the tabs should be expanded using `expand`(1) before coming to `fold`.

Restrictions

The `fold` command may interfere with underlining.

Return Value

The `fold` command returns zero (0) on successful completion.

See Also

expand(1)

Name

folder – set folder or display current foldername

Syntax

folder [*+foldername*] [*msg*] **[–all] [–fast] [–nofast] [–header] [–noheader] [–pack] [–nopack] [–recurse] [–norecurse] [–total] [–nototal] [–print] [–noprint] [–list] [–nolist] [–push] [–pop] [–help]**

Description

The `folder` command lets you set the current folder or display its name and its contents. It can also be used to manage the folder stack. If you use the `folder` command without a *+foldername* argument, the contents of the current folder will be displayed on the screen.

If you use `folder` with the *+foldername* argument, the specified folder will be set to be the current folder.

If you use `folder` with the *msg* argument, it will set the specified message to be current.

The display is identical whether you set the folder or display the contents of the current folder. The following example shows the type of display that is produced. The display lists the current folder, the number of messages in it, the range of the messages (low–high), and the current message within the folder, and will flag extra files if they exist.

```
inbox+ has  16 messages    (  3- 22); cur=  15
```

If a *+foldername* and/or *msg* argument are specified, they will become the current folder and/or message.

Options

Specifying **–all** will produce a line for each folder in your MH directory, sorted alphabetically. This is identical to the effect that is obtained if you specify `folders`. The display that is obtained is illustrated in the following example.

```
Folder       # of messages (range); cur  msg  (other files)
V2.3    has   3 messages    (1-3).
adrian  has  20 messages   (1-20);    cur=  2.
brian   has  16 messages   (1-16).
chris   has  12 messages   (1-12).
copylog has 242 messages (1- 242);   cur= 225.
inbox+  has  73 messages (1- 127);   cur= 127.
int     has   4 messages    (1-4);     cur= 2    (others).
jack    has  17 messages   (1-17);   cur=  17.


            TOTAL= 387 messages in 8 folders.
```

The plus sign (+) after inbox indicates that it is the current folder. The folder `int` has (others) after the description of the folder. This indicates that the folder `int` contains files which are not messages. These files may either be sub-folders, or files that do not belong under the MH file naming scheme.

You can get the same effect by specifying `folders` instead of `folder`–all (see `folders`(1mh).)

The header is output if either an `-all` or a `-header` switch is specified. It is suppressed by `-noheader`. The `-total` switch will produce only the summary line. If you select the `-nototal` option, the summary line will be suppressed but the rest of the information about the folders will be displayed.

If `-fast` is given, only the folder name will be listed. This is faster because the folders need not be read.

The `-pack` switch will compress the message names in a folder, removing holes in message numbering.

The `-recurse` switch will list each folder recursively. Use of this option effectively defeats the speed enhancement of the `-fast` option, since each folder must be searched for subfolders. Nevertheless, the combination of these options is useful.

If you specify a *+folder* that does not exist, you will be asked whether you want to create it. This is a good way to create an empty folder for later use. The following example shows how you can create a sub folder in the folder +test using this method.

```
% folder +test/testtwo
Create folder "/usr/username/Mail/test/testwo"? y
        test/testtwo+ has no messages.
```

See `refile`(1mh) for more details of sub folders.

The `-push`, `-pop`, and – options can be used to manage the folder stack.

The `-push` switch directs `folder` to push the current folder onto the folder-stack, and make the `+folder` argument into the current folder. If `+folder` is not given, the current folder and the top of the folder-stack are exchanged. This corresponds to the `pushd` operation in the `Cshell` (see `csh`(1)).

The `-pop` switch directs `folder` to discard the top of the folder-stack, after setting the current folder to that value. No *+folder* argument is allowed. This corresponds to the `popd` operation in the `Cshell` (see `csh`(1)) The `-push` switch and the `-pop` switch are mutually exclusive: the last occurrence of either one overrides any previous occurrence of the other.

The `-list` switch directs `folder` to list the contents of the folder-stack. No *+folder* argument is allowed. After a successful `-push` or `-pop`, the `-list` action is taken. This corresponds to the **dirs** operation in the Cshell. The defaults for `folder` are:

> *+foldername* defaults to the current folder
> *msg* defaults to none
> `-nofast`
> `-noheader`
> `-nototal`
> `-nopack`
> `-norecurse`
> `-print` is the default if `-list`, `-push` or `-pop` are specified.

Files

`$HOME/.mh_profile`	The user profile

Profile Components

`Path:`	To determine your MH directory
`Current-Folder:`	To find the default current folder
`Folder-Protect:`	To set mode when creating a new folder
`Folder-Stack:`	To determine the folder stack
`lsproc:`	Program to list the contents of a folder

See Also

csh(1), refile(1mh), mhpath(1mh)

Name

folders – list folders and contents

Syntax

folders [*folder*] [*msg*] **[–fast] [–nofast] [–header] [–noheader] [–pack] [–nopack] [–recurse] [–norecurse] [–total] [–nototal] [–print] [–noprint] [–list] [–nolist] [–push] [–pop] [–help]**

Description

The `folders` command lets you display the names of your folders and the number of messages that they each contain.

When you use `folders`, the display contains a line for each folder in your MH directory, sorted alphabetically. This is illustrated in the following example.

```
Folder         # of messages (range); cur  msg  (other files)
V2.3    has   3 messages    (1-3).
adrian  has  20 messages   (1-20);    cur=  2.
brian   has  16 messages   (1-16).
chris   has  12 messages   (1-12).
copylog has 242 messages (1- 242);   cur= 225.
inbox+  has  73 messages (1- 127);   cur= 127.
int     has   4 messages    (1-4);     cur= 2    (others).
jack    has  17 messages   (1-17);   cur=  17.


            TOTAL= 387 messages in 8 folders.
```

The plus sign (+) after inbox indicates that it is the current folder. The information about the `int` folder includes the term (others). This indicates that the folder `int` contains files which are not messages. These files may either be sub-folders, or files that do not belong under the MH file naming scheme.

In all respects, the effect of using `folders` is identical to the effect of using **folder –all**. See `folder`(1mh) for details.

If you use `folders` with the *+foldername* argument, `folders` will display all the subfolders within the nominated folder, as shown in the following example. See `refile`(1mh) for more details of sub folders.

```
% folders +test
Folder            # of messages (  range  ); cur  msg  (other files)
test+ has           18 messages (   1-  18);            (others).
test/testone has     1 message  (   1-   1).
test/testtwo has    no messages.

       TOTAL=   19 messages in 3 folders.
```

If you specify a folder, that folder will become the current folder.

The remainder of the options work as they do for `folder`–all. See `folder`(1mh) for details.

The defaults for `folders` are:

+foldername defaults to all
msg defaults to none

```
-nofast
-noheader
-nototal
-nopack
-norecurse
```

Restrictions

You cannot have more than 100 folders in any one level.

Files

`$HOME/.mh_profile`	The user profile

Profile Components

`Path:`	To determine your MH directory
`Current-Folder:`	To find the default current folder
`Folder-Protect:`	To set mode when creating a new folder
`Folder-Stack:`	To determine the folder stack
`lsproc:`	Program to list the contents of a folder

See Also

csh(1), refile(1mh), mhpath(1mh)

Name

forw – forward messages

Syntax

forw [+*folder*] [*msgs*] **[–annotate] [–noannotate] [–draftfolder FI+folder] [–draftmessage** *msg*] **[–nodraftfolder] [–editor** *editorname*] **[–noedit] [–filter** *filterfile*] **[–form** *formfile*] **[–format] [–noformat] [–inplace] [–noinplace] [–whatnowproc** *program*] **[–nowhatnowproc] [–digest** *list*] **[–issue** *number*] **[–volume** *number*] **[–help]**

Description

Use `forw` to send one or more messages on to recipients who were not the original addressees. A message header is added to the message(s) to be forwarded and the message is encapsulated. Forwarded messages appear to originate from the forwarder and not the sender of the original message. In this respect `forw` is different from `dist`. The other difference between `forw` and `dist`, is that you can add your own message to a forwarded message with `forw`.

An editor is invoked as in `comp`, and after editing is complete, you are prompted before the message is sent.

You can forward several messages at once by specifying the message numbers separated by spaces. The following example would concatenate messages 3, 5 and 7 and forward them as one message.

```
forw 3 5 7
```

You can also forward a number of messages by specifying a range. The following example would forward messages 3, 4, 5, 6, 7 as one message. Note that there are no spaces when you specify a range of messages.

```
forw 3-7
```

Options

The default message form contains the following elements:

```
To:
cc:
Subject:
--------
```

If the file named `forwcomps` exists in your MH directory, it will be used instead of this form. The file specified by **–form** *formfile* will be used if given.

If the draft file exists, you cannot normally forward another message until you have cleared the draft file. This is because `forw` uses the draft file to compose the forwarded message. If you attempt to do this, `forw` will ask you what you want to do. Press <RETURN> to see the following options.

A reply of `quit` will abort `forw`, leaving the draft intact; `replace` will replace the existing draft with a blank skeleton; and `list` will display the draft.

If you set up the draftfolder: drafts line in your .mh_profile, forw will forward whichever message(s) you choose, without endangering any unsent messages (see mh-profile(5mh)).

If the -annotate switch is given, each message being forwarded will be annotated with the lines

```
Forwarded: date
Forwarded: addrs
```

where each address list contains as many lines as required. This annotation will be done only if the message is sent directly from forw. If the message is not sent immediately from forw, comp -use may be used to re-edit and send the constructed message, but the annotations will not be added (see comp(1mh)). The -inplace switch causes annotation to be done in place in order to preserve links to the annotated message.

When forw is told to annotate the messages it forwards, it does not annotate them until the draft is successfully sent. If you choose to push at the whatnow? prompt instead of send, it is possible to confuse forw by re-ordering the folder: For example, by using

```
folder -pack
```

before the message is successfully sent. The functions dist and repl do not have this problem.

You can specify the editor that you want to use to edit your forwarded message with the -editor option. You can suppress editing altogether with the -noedit option.

Although forw uses the -form *formfile* switch to direct it how to construct the beginning of the draft, the -filter *filterfile*, -format, and -noformat switches direct forw as to how each forwarded message should be formatted in the body of the draft.

If -noformat is specified, then each forwarded message is output exactly as it appears. If -format or -filter *filterfile* is specified, then each forwarded message is filtered (re-formatted) prior to being output to the body of the draft. The *filterfile* for forw should be a standard form file for mhl, as forw will invoke mhl to format the forwarded messages. The default message filter that you get with -format is:

```
width=80,overflowtext=,overflowoffset=10
leftadjust,compress,compwidth=9
Date:formatfield="%<(nodate{text})%|%(tws{text})%>"
From:
To:
cc:
Subject:
:
body:nocomponent,overflowoffset=0,noleftadjust,nocompress
```

If the file named mhl.forward exists in the user's MH directory, it will be used instead of this form. In either case, the file specified by -filter *filterfile* will be used if given. To summarize: -noformat will reproduce each forwarded message exactly, -format will use mhl and a default *filterfile*, mhl.forward, to format each forwarded message, and -filter *filterfile* will use the named filterfile to format each forwarded message with mhl.

Each forwarded message is separated with an encapsulation boundary so that when received, the message is suitable for expanding with burst(1mh).

If you use prompter as your editor, you can specify prompter's -prepend switch in the mh_profile file. If you do this any commentary text is entered before the forwarded messages. See prompter(1mh) for details of the other prompter options.

Normally forw uses the draft file, or drafts folder if you have one set up, to compose the forwarded message in. You can make forw compose the message to be forwarded in alternative locations by specifying the *+foldername* or *msg* arguments. See comp(1mh) for details.

When you exit from the editor, forw invokes the whatnow program. See whatnow(1mh) for details of the available options. The invocation of this program can be inhibited by using the -nowhatnowproc switch.

The -digest *list*, -issue *number* and -volume *number* switches implement a digest facility for MH.

The following defaults are valid:

> *+foldername* defaults to the current folder *msgs* defaults to the current message
> -noannotate
> -nodraftfolder
> -noformat
> -noinplace

If the whatnowproc is whatnow, then forw uses its own built-in whatnow; it does not actually run the whatnow program. Hence, if you define your own whatnowproc, do not call it whatnow since forw will not run it.

Files

/usr/new/lib/mh/forwcomps	The message skeleton
<mh-dir>/forwcomps	An alternative message skeleton
/usr/new/lib/mh/digestcomps	The message skeleton if -digest is given
or <mh-dir>/digestcomps	Rather than the standard skeleton
/usr/new/lib/mh/mhl.forward	The message filter
or <mh-dir>/mhl.forward	Rather than the standard filter
$HOME/.mh_profile	The user profile
<mh-dir>/draft	The draft file

Profile Components

Path:	To determine your MH directory
Current-Folder:	To find the default current folder
Draft-Folder:	To find the default draft–folder
Editor:	To override the default editor
Msg-Protect:	To set mode when creating a new message (draft)
fileproc:	Program to refile the message
mhlproc:	Program to filter messages being forwarded
whatnowproc:	Program to ask the "What now?" questions

See Also

comp(1mh), dist(1mhs), refile(1mh), repl(1mh), send(1mh), whatnow(1mh)
Proposed Standard for Message Encapsulation (RFC 934)

Name

from – identifies sender of mail

Syntax

from [**–f** *mailbox*] [**–s** *sender*]

Description

The `from` command prints out the mail header lines in a mailbox file to show you who has sent mail to you.

Options

–f *mailbox*	Uses specified file instead of your normal mail file. If this option is used, but file argument is not specified, read your mbox file.
–s *sender*	Prints mail headers for mail sent by specified sender.

Files

/usr/spool/mail/*

See Also

mail(1)

Name

fsplit – split a multi-routine Fortran file into individual files

Syntax

fsplit [**–e** *efile...*] [*file*]

Description

The `fsplit` command takes as input either a file or standard input containing Fortran source code. It attempts to split the input into separate routine files of the form *name.f,* where *name* is the name of the program unit (for example, function, subroutine, block data or program). The name for unnamed block data subprograms has the form *blkdtaNNN.f* where NNN is three digits and a file of this name does not already exist. For unnamed main programs the name has the form *mainNNN.f.* If there is an error in classifying a program unit, or if *name.f* already exists, the program unit is put in a file of the form *zzzNNN.f* where *zzzNNN.f* does not already exist.

Normally each subprogram unit is split into a separate file.

Options

– e *efile* Splits only specified subprogram units into separate files.

Examples

The following example splits readit and doit into separate files:

```
fsplit -e readit -e doit prog.f
```

Restrictions

The `fsplit` command assumes the subprogram name is on the first noncomment line of the subprogram unit. Nonstandard source formats may confuse `fsplit`.

It is hard to use **–e** for unnamed main programs and block data subprograms since you must predict the created file name.

Diagnostics

If names specified using the **–e** option are not found, a diagnostic is written to *standard error*.

Name

ftp – file transfer program

Syntax

ftp [–v] [–d] [–i] [–n] [–g] [host]

Description

The `ftp` command 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 the client host is specified on the command line, `ftp` immediately attempts to establish a connection to an FTP server on that host; otherwise, `ftp` enters its command interpreter and awaits instructions from the user. While `ftp` is awaiting commands from the user, it provides the user with the prompt ftp>. The following commands are recognized by `ftp`:

! Invokes a shell on the local machine.

$ *macro-name* [*args*]
: Executes the macro *macro-name* that was defined with the **macdef** command. Arguments are passed to the macro unglobbed.

account [*passwd*]
: Supplies a supplemental password required by a remote system for access to resources once a login has been successfully completed. If no argument is included, the user is prompted for an account password in a non-echoing input mode.

append *local-file* [*remote-file*]
: Appends a local file to a file on the remote machine. If *remote-file* is not specified, the local file name is used in naming the remote file. File transfer uses the current settings for *type*, *format*, *mode*, and *structure*.

ascii Sets the file transfer *type* to network ASCII. This is the default type.

bell Arranges for a bell to sound after each file transfer command is completed.

binary Sets the file transfer *type* to support binary image transfer.

bye Terminates the FTP session with the remote server and exits `ftp`.

case Toggles the remote computer's file name case mapping during **mget** commands. When **case** is on (default is off), the remote computer's file names are written in the local directory with all letters in upper case mapped to lower case.

cd *remote-directory*
: Changes the working directory on the remote machine to *remote-directory*.

cdup Changes the remote machine working directory to the parent of the current remote machine working directory.

close Terminates the FTP session with the remote server and returns to the command interpreter.

cr Toggles the 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*
Deletes the file *remote-file* on the remote machine.

debug [*debug-value*]
Toggles the 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 q-->.

dir [*remote-directory*] [*local-file*]
Prints a listing of the directory contents in the directory, *remote-directory*, and, optionally, places 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, output comes to the terminal.

disconnect A synonym for **close**.

form *format*
Sets the file transfer *form* to *format*. The default format is file.

get *remote-file* [*local-file*]
Retrieves the *remote-file* and stores it on the local machine. If the local file name is not specified, it is given the same name it has on the remote machine. The current settings for *type*, *form*, *mode*, and *structure* are used while transferring the file.

hash Toggles the hash-sign (#) printing for each data block transferred. The size of a data block is 1024 bytes.

glob Toggles filename expansion for `mdelete`, `mget`, and `mput`. If globbing is turned off with `glob`, the file name arguments are taken literally and not expanded. Globbing for `mput` is done as in `csh`(1). For `mdelete` and `mget`, each remote file name 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 entering: mls *remote-files*. Neither `mget` nor `mput` is meant to transfer entire directory subtrees of files. That can be done by transferring a `tar`(1) archive of the subtree (in binary mode).

lcd [*directory*]
Changes the working directory on the local machine. If no *directory* is specified, the user's home directory is used.

ls [*remote-directory*] [*local-file*]
Prints 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, the output is sent to the terminal.

macdef *macro-name*
Defines 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 dollar signs ($) and backslashes (\) as special characters. A dollar sign ($) followed by a number (or numbers) is replaced by the corresponding argument on the macro invocation command line. A dollar sign ($) followed by an i signals the 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 backslash (\) followed by any character is replaced by that character. Use the backslash (\) to prevent special treatment of the dollar sign ($).

mdelete *remote-files*
Deletes the specified files on the remote machine. If globbing is enabled, the specification of remote files will first be expanded using **ls.**

mdir *remote-files local-file*
Obtains a directory listing of multiple files on the remote machine and places the result in *local-file*.

mget *remote-files*
Retrieves the specified files from the remote machine and places them in the current local directory. If globbing is enabled, the specification of remote files will first be expanding using `ls`.

mkdir *directory-name*
Makes a directory on the remote machine.

mls *remote-files local-file*
Obtains an abbreviated listing of multiple files on the remote machine and places the result in *local-file*.

mode [*mode-name*]
Sets the file transfer *mode* to *mode-name*. The default mode is the stream mode.

mput *local-files*
Transfers multiple local files from the current local directory to the current working directory on the remote machine.

nmap [*inpattern outpattern*]
Sets or unsets 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 which are issued without a specified remote target filename. If arguments are specified, local filenames are mapped during **mget** commands and **get** commands which are 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 a backslash (\) to prevent this special treatment of the dollar sign ($) 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 file name mydata.data, $1 has the value mydata, and $2 has 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 replace 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] yields 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 backslash (\) to prevent special treatment of the dollar sign ($), left bracket ([), right bracket (]), and comma (,).

ntrans [*inchars* [*outchars*]]

Sets or unsets 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 which are issued without a specified remote target filename. If arguments are specified, characters in local filenames are translated during **mget** commands and **get** commands which are 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 file name.

open *host* [*port*]

Establishes a connection to the specified *host* FTP server. If an optional port number is supplied, `ftp` attempts to contact an FTP server at that port. If the *auto-login* option is on (default), `ftp` automatically attempts to log the user in to the FTP server (see below).

prompt Toggles interactive prompting. Interactive prompting occurs during multiple file transfers to allow the user to retrieve or store files selectively. If prompting is turned off (default), any *mget* or *mput* transfers all files.

proxy *ftp-command*

Executes 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. Type 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 auto-login 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

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*]
Stores a local file on the remote machine. If *remote-file* is unspecified, the local file name is used in naming the remote file. File transfer uses the current settings for *type*, *format*, *mode*, and *structure*.

pwd Prints the name of the current working directory on the remote machine.

quit A synonym for bye.

quote *arg1 arg2 ...*
Sends the arguments that are specified, verbatim, to the remote FTP server. A single FTP reply code is expected in return.

recv *remote-file* [*local-file*]
A synonym for get.

remotehelp [*command-name*]
Requests help from the remote FTP server. If a *command-name* is specified it is supplied to the server as well.

rename [*from*] [*to*]
Renames the file *from* on the remote machine, to the file *to*.

reset Clears the reply queue. This command re-synchronizes command/reply sequencing with the remote ftp server. If the remote server violates the ftp protocol, resynchronization may be necessary.

rmdir *directory-name*
Deletes a directory on the remote machine.

runique Toggles 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 Toggles the use of PORT commands. By default, `ftp` attempts to use a PORT command when establishing a connection for each data transfer. If the PORT command fails, `ftp` uses the default data port. When the use of PORT commands is disabled, no attempt is made to use PORT commands for each data transfer. This is useful for certain FTP implementations which do ignore PORT commands but, incorrectly, indicate that they have been accepted.

status Shows the current status of `ftp`.

struct [*struct-name*]
Sets the file transfer *structure* to *struct-name*. By default the file structure is used.

sunique Toggles storing of files on a remote machine under unique file names. The remote ftp server must support the ftp protocol STOU command for successful completion of this command. The remote server reports the unique name. Default value is off.

tenex Sets the file transfer type to that needed to talk to TENEX machines.

trace Toggles packet tracing.

type [*type-name*]
Sets 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*]
Identifies the user to the remote FTP server. If the password is not specified and the server requires it, `ftp` disables the local echo and then prompts the user for it. If an account field is not specified, and the FTP server requires it, the user is prompted for it also. Unless `ftp` is invoked with auto-login disabled, this process is done automatically on initial connection to the FTP server.

verbose Toggles the verbose mode. In verbose mode, all responses from the FTP server are displayed to the user. In addition, if verbose is on, statistics regarding the efficiency of a file transfer are reported when the transfer is complete. By default, verbose is on.

? [*command*]
A synonym for help.

Command arguments which have embedded spaces may be quoted with quotation (") marks.

Aborting A File Transfer

To abort a file transfer, use the terminal interrupt key (usually <CTRL/C>). Sending transfers are halted immediately. Receiving transfers are halted by sending a 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 appears when the remote server has completed sending the requested file.

The terminal interrupt key sequence is 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 ABOR processing, 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) Standard input is used for reading and standard output is used for writing when the file name is specified by a minus sign (–).
2) If the first character of the file name is a vertical line (|), the remainder of the argument is interpreted as a shell command. The `ftp` command 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, as in ``"| ls -lt"''. A particularly useful example of this mechanism is: ``dir |more''.
3) If globbing is enabled, local file names are expanded according to the rules used in the `csh`(1) (compare to the `glob` command). If the `ftp` command expects a single local file, such as **put,** only the first filename generated by the globbing operation is used.
4) For **mget** commands and **get** commands with unspecified local file names, the local filename is the remote filename and can 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 file names, the remote filename is the local filename and may be altered by a **ntrans** or **nmap** setting. The resulting filename can then be altered by the remote server if **sunique** is on.

File Transfer Parameters

Many parameters can affect a file transfer. The *type* can be ascii, image (binary), ebcdic, or local byte size (for PDP-10's and PDP-20's generally). The `ftp` command supports the ascii and image types of file transfer and local byte size 8 for tenex mode transfers.

The `ftp` command supports only the default values for the remaining file transfer parameters: *mode*, *form*, and *struct*.

The .netrc File

The .netrc file contains login and initialization information used by the auto-login process. It resides in the user's home directory. The following tokens are recognized; they may be separated by spaces, tabs, or new-lines:

machine *name*
: Identifies a remote machine name. The auto-login 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* Identifies a user on the remote machine. If this token is present, the auto-login process initiates a login using the specified name.

password *string*
Supplies a password. If this token is present, the auto-login process supplies 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, and if the .netrc is readable by anyone other than the user, `ftp` aborts the auto-login process.

account *string*
Supplies an additional account password. When this token is present, the auto-login process supplies the the remote server with an additional account password if the remote server requires it. If it does not, the auto-login process initiates an ACCT command.

macdef *name*
Defines a macro. This token functions like the `ftp` **macdef** command. A macro is defined with a specified name; its contents begin with the next .netrc line and continue until a null line (consecutive new-line characters) is encountered. If a macro named *init* is defined, it is automatically executed as the last step in the auto-login process.

Options

–d Enables debugging.

–g Disables file name expansion.

–i Disables interactive prompting during multiple file transfers.

–n Disables autologin during an initial connection. If auto-login is enabled, `ftp` will check the .netrc file in the user's home directory for an entry describing an account on the remote machine. If no entry exists, `ftp` will use the login name on the local machine as the user identity on the remote machine, prompt for a password and, optionally, an account with which to login.

–v Displays all responses from the remote server as well as all data transfer statistics.

Restrictions

Correct execution of many commands depends on proper behavior by the remote server.

The `ftpd` server prevents the unauthorized users listed in the /etc/ftpusers file from transferring files.

An error in the treatment of carriage returns in the 4.2BSD UNIX ascii-mode transfer code has been corrected. This correction may result in incorrect transfers of binary files to and from 4.2BSD servers using the ascii type. Avoid this problem by using the binary image type.

Files

/etc/ftpusers Contains the list of unauthorized users

See Also

services(5), ftpd(8c), inetd(8c), syslog(8)

Name

gcore – get core images of running processes

Syntax

gcore *process-id...*

Description

The `gcore` command creates a core image of each specified process, suitable for use with `adb`(1) or `dbx`(1).

Restrictions

Paging activity that occurs while `gcore` is running may cause the program to become confused. For best results, the desired processes should be stopped.

Files

core.<process-id> core images

Name

gencat – generate a formatted message catalog

Syntax

gencat [-h *hdrfile*] *catfile msgfile ...*

Description

The `gencat` command takes one or more message source files and either creates a new catalog or merges new message text into an existing catalog. You should use the extension `.msf` for message text files (for example, `msgfile.msf`) and the extension `.cat` for catalogs (for example, `catfile.cat`) to process files with the `gencat` command.

In some cases, a formatted message catalog exists that has the same name the one that `gencat` is creating. When this occurs, `gencat` merges the messages from the source message catalogs into this existing formatted message catalog. The command merges the source message catalogs into the formatted message catalog in the same manner as it merges a group of source message catalogs. If a source message catalog contains the same set number or message number as a set or message in the formatted message catalog, the source message catalog set or message has precedence. For example, if both the source and formatted message catalogs contain a message number 25, the message text for message 25 in the source message catalog replaces the message text in the formatted message catalog. Thus, when source message catalogs are merged with formatted message catalogs, the formatted catalogs are updated.

The `-h` option indicates that the source message file contains set and message mnemonics instead of numeric identifiers. This option causes the `gencat` command to create the header file, *hdrfile*. The header file contains C preprocessor directives that control the mapping between set and message labels specified in the source catalogs and the set and message numbers written to the formatted catalog. If you use message labels in your program, you must include the header file `gencat` creates. Use the `#include` directive to include the header file. When you specify the `-h` option, `gencat` does not merge source message catalogs with existing formatted message catalogs. If a formatted message catalog exists, the `gencat` command writes over it with the new message catalog. If no formatted message catalog exists, `gencat` creates one.

For information on the source format for messages files, see the *Guide to Developing International Software.*

Options

-h Generate a header file that maps set and message labels in the formatted message catalog to set and message labels in source message catalogs.

The `gencat` command ignores this option if you use set and message numbers, rather than set and message labels.

You must include the header file *hdrfile* in your C program if you want to use set and message labels.

Restrictions

Source message catalogs you want to format using `gencat` can contain either set and message numbers or set and message labels, but not both.

Numeric message source files are guaranteed portable between X/Open systems.

See Also

intro(3int), extract(1int), trans(1int), catgets(3int), catopen(3int)
Guide to Developing International Software

Name

get – get a copy of SCCS file

Syntax

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

Description

The `get` command generates an ASCII text file from each named SCCS file according to the specifications given by its options. The options, which begin with –, can be specified in any order, but all options apply to all named SCCS files. If a directory is named, `get` 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; 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* whose name is derived from the SCCS file name by simply removing the leading **s.**; (see also *FILES*, below).

Each of the options is explained below as though only one SCCS file is to be processed, but the effects of any options applies independently to each named file.

Options

–r*SID* Indicates specified delta version number. Table 1 below 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** option is also used), as a function of the SID specified.

–c*cutoff* The *cutoff* is a date-time in the following form:

```
YY[MM[DD[HH[MM[SS]]]]]
```

No changes (deltas) to the SCCS file that 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 two 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**".

–e Gets specified delta version for edit. The **–e** option used in a `get` for a particular version (SID) of the SCCS file prevents further `gets` 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 SCCS front end processor is used, the command `get–e` is replaced by **edit.**

If the *g-file* generated by `get` with an **–e** option is accidentally ruined while being edited, it may be regenerated by re-executing the `get`

command with the **–k** option in place of the **–e** option.

SCCS file protection specified by the ceiling, floor, and authorized user list stored in the SCCS file are enforced when the **–e** option is used. For further information, see `admin(1)`.

–b Gets delta from new branch and must be used with **-e** option. This option is ignored if the **b** flag is not present in the file or if the retrieved delta is not a leaf delta. For further information, see `admin(1)`. 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* Includes specified list of deltas. The *list* has the following syntax:

<list> ::= <range> | <list> , <range>
<range> ::= SID | SID – SID

SID, may be in any form shown in the ''SID Specified'' column of Table 1. Partial SIDs are interpreted as shown in the ''SID Retrieved'' column of Table 1.

–x*list* Excludes specified list of deltas. See the **–i** option for the *list* format.

–k Does not expand ID keywords. The **–k** option is implied by the **–e** option.

–l Writes a delta summary to 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 **FILES** for the format of the *l-file*.

–p Writes text to stdout. No *g-file* is created. All output which normally goes to the standard output goes to file descriptor 2 instead, unless the **–s** option is used, in which case it disappears.

–s Suppresses all messages, except those for fatal errors. However, fatal error messages, which always go to file descriptor 2, remain unaffected.

–m Precedes each text line with delta version number. The format is: SID, followed by a horizontal tab, followed by the text line.

–n Precedes each text line with identification keyword. The format is: %M% value, followed by a horizontal tab, followed by the text line. When both the **–m** and **–n** option are used, the format is: %M% value, followed by a horizontal tab, followed by the **–m** option 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 Gets most recently created (top) delta. For example, **–r1**), or release and level, for example, **–r1.2**).

–u Sets the time of the g–file to the time of the s–file. This results in a g-file with a time equal to the last delta. This is useful for build

scripts which extract all files from the SCCS database and then do a make.

–a*seq-no.* Retrieves the specified delta sequence number. For further information, see `sccsfile(5)`. This option is used by the `comb` command. It is not a generally useful option, and users should not use it. If both the **–r** and **–a** option are specified, the **–a** option is used. Care should be taken when using the **–a** option in conjunction with the **–e** option, as the SID of the delta to be created may not be what one expects. The **–r** option can be used with the **–a** and **–e** option 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** option 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 file name is printed (preceded by a new-line) before it is processed. If the **–i** option is used included deltas are listed following the notation ''Included''; if the **–x** option is used, excluded deltas are listed following the notation ''Excluded''.

The SCCS identification strings are defined in the following table:

SID* Specified	–b Option 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, in that order; 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 (that is, maximum branch number plus one) of level L within release R. Note that if the SID specified is of the form R.L, R.L.B, or R.L.B.S, each of the specified components must exist.

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

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

Successor.

† The **–b** option is effective only if the **b** flag is present in the file. An entry of – means "irrelevant". For further information, see admin(1).

‡ 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

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

%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% The time the newest applied delta was created (HH:MM:SS).

%Y% Module type: value of the **t** flag in the SCCS file For further information, see admin(1).

%F% SCCS file name.

%P% Fully qualified SCCS file name.

%Q% The value of the **q** flag in the file. For further information, see admin(1).

%C% Current line number. This keyword is intended for identifying output program messages such as "this shouldn't 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 UNIX program files. %W% = %Z%%M%<horizontal-tab>%I%

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

Restrictions

If the user has write permission in the directory containing the *g-files,* but the real user does not, then only one file can be named when the **–e** option is used.

Diagnostics

See `sccshelp(1)` for explanations.

Files

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 file name is formed from the SCCS file name: the last component of all SCCS file names must be of the form s.*module-name*, 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 file names would be **xyz.c**, **l.xyz.c**, **p.xyz.c**, and **z.xyz.c**, respectively.

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

The *l-file* contains a table showing which deltas were applied in generating the retrieved text. The *l-file* is created in the current directory if the **–l** option 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 wasn't applied and ignored;
   * if the delta wasn't applied and wasn't ignored.
c. A code indicating a ''special'' reason why the delta was or was not applied:
   ´I': Included.
   ´X': Excluded.
   ´C': Cut off (by a **–c** option).
d. Blank.
e. SCCS identification (SID).
f. Tab character.
g. Date and time (in the form YY/MM/DD HH:MM:SS) of creation.
h. Blank.
i. Login name of person who created delta.

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

The *p-file* is used to pass information resulting from a `get` with an **–e** option along to delta. Its contents are also used to prevent a subsequent execution of `get` with an **–e** option for the same SID until delta is executed or the joint edit flag, **j**, see `admin`(1), is set in the SCCS file.

The *p-file* is created in the directory containing the SCCS file and the effective user must have write permission in that directory. Its mode is 644 and it is owned by the effective user.

The format of the *p-file* is the following: the gotten SID, followed by a blank, followed by the SID that the new delta will have when it is made, followed by a blank, followed by the login name of the real user, followed by a blank, followed by the date-time the `get` was executed, followed by a blank and the **–i** option if it was present, followed by a blank and the **–x** option if it was present, followed by a new-line. 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 (two bytes) process ID of the command `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 for the *p-file* apply for the *z-file*. The *z-file* is created mode 444.

See Also

admin(1), delta(1), prs(1), sccs(1), sccsfile(5), sccshelp(1), what(1)
Guide to the Source Code Control System

Name

getopt – parse command options

Syntax

set – – `getopt optstring $*`

Description

The `getopt` command breaks up options in command lines for easy parsing by Shell procedures and checks for legal options. The **optstring** option letters are recognized if a letter is followed by a colon, the option expects an argument which may or may not be separated from it by white space. For further information, see `getopt`(3c).

The special option, specified by two minus signs (– –), delimits the end of the options. If the delimiters are used explicitly, `getopt` recognizes it; otherwise, `getopt` generates it. In either case, `getopt` places the delimiter at the end of the options. The positional parameters ($1 $2 ...) of the shell are reset so that each option is preceded by a single minus sign (–) and is in its own positional parameter; each option argument is also parsed into its own positional parameter.

Examples

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

```
#!/bin/sh5
set -- `getopt abo: $*`
if [ $? != 0 ]
then
       echo $USAGE
       exit 2
fi
for i in $*
do
       case $i in
       -a | -b)     FLAG=$i; shift;;
       -o)    OARG=$2; shift 2;;
       --)    shift; break;;
       esac
done
```

This code accepts any of the following as equivalent:

```
cmd -aoarg file file
cmd -a -o arg file file
cmd -oarg -a file file
cmd -a -oarg -- file file
```

Diagnostics

The `getopt` command prints an error message on the standard error when it encounters an option letter not included in *optstring*.

See Also

sh5(1), getopt(3)

Name

gprof – display call graph profile data

Syntax

gprof [*options*] [*a.out* [*gmon.out...*]]

Description

The `gprof` command produces an execution profile of C, Pascal, or Fortran77 programs. The effect of called routines is incorporated in the profile of each caller. The profile data is taken from the call graph profile file (gmon.out default) which is created by programs which are compiled with the **–pg** option of `cc`, `pc`, and `f77`. That option also links in versions of the library routines which are compiled for profiling. The symbol table in the named object file (a.out default) is read and correlated with the call graph profile file. If more than one profile file is specified, the `gprof` output shows the sum of the profile information in the given profile files.

First, a flat profile is given, similar to that provided by `prof`(1). This listing gives the total execution times and call counts for each of the functions in the program, sorted by decreasing time.

Next, these times are propagated along the edges of the call graph. Cycles are discovered, and calls into a cycle are made to share the time of the cycle. A second listing shows the functions sorted according to the time they represent including the time of their call graph descendents. Below each function entry is shown its (direct) call graph children, and how their times are propagated to this function. A similar display above the function shows how this function's time and the time of its descendents is propagated to its (direct) call graph parents.

Cycles are also shown, with an entry for the cycle as a whole and a listing of the members of the cycle and their contributions to the time and call counts of the cycle.

Options

The following options are available:

–a Suppresses the printing of statically declared functions. If this option is given, all relevant information about the static function (for example, time samples, calls to other functions, calls from other functions) belongs to the function loaded just before the static function in the `a.out` file.

–b Suppresses the printing of a description of each field in the profile.

–c The static call graph of the program is discovered by a heuristic which examines the text space of the object file. Static-only parents or children are indicated with call counts of 0.

–E *name* Suppresses the printing of the graph profile entry for routine *name* (and its descendants) as **–e**, above, and also excludes the time spent in *name* (and its descendants) from the total and percentage time computations. (For example, **–E** *mcount* **–E** *mcleanup* is the default.)

–e *name*
: Suppresses the printing of the graph profile entry for routine *name* and all its descendants More than one **–e** option may be given. Only one *name* may be given with each **–e** option.

–F *name*
: Prints the graph profile entry of only the routine *name* and its descendants (as **–f,** above) and also uses only the times of the printed routines in total time and percentage computations. More than one **–F** option may be given. Only one *name* may be given with each **–F** option. The **–F** option overrides the **–E** option.

–f *name*
: Prints the graph profile entry of only the specified routine *name* and its descendants. More than one **–f** option may be given. Only one *name* may be given with each **–f** option.

–s
: Produces a profile file *gmon.sum* is produced which represents the sum of the profile information in all the specified profile files. This summary profile file may be given to subsequent executions of gprof (probably also with a **–s**) to accumulate profile data across several runs of an *a.out* file.

–z
: Displays routines which have zero usage (as indicated by call counts and accumulated time). This is useful in conjunction with the **–c** option for discovering which routines were never called.

Restrictions

Beware of quantization errors. The granularity of the sampling is shown, but remains statistical at best. We assume that the time for each execution of a function can be expressed by the total time for the function divided by the number of times the function is called. Thus the time propagated along the call graph arcs to parents of that function is directly proportional to the number of times that arc is traversed.

Parents which are not themselves profiled have the time of their profiled children propagated to them, but they appear to be spontaneously invoked in the call graph listing, and do not have their time propagated further. Similarly, signal catchers, even though profiled, appear to be spontaneous (although for more obscure reasons). Any profiled children of signal catchers should have their times propagated properly, unless the signal catcher was invoked during the execution of the profiling routine, in which case all is lost.

The profiled program must call `exit(2)` or return normally for the profiling information to be saved in the `gmon.out` file.

Files

`a.out`	the name list and text space.
`gmon.out`	dynamic call graph and profile.
`gmon.sum`	summarized dynamic call graph and profile.

See Also

cc(1), prof(1), profil(2), monitor(3)

Name

graph – draw a graph

Syntax

graph [*option...*]

Description

The `graph` command with no options takes pairs of numbers from the standard input as abscissas and ordinates of a graph. Successive points are connected by straight lines. The graph is encoded on the standard output for display by the `plot`(1g) filters.

If the coordinates of a point are followed by a nonnumeric 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 new lines.

A legend indicating grid range is produced with a grid unless the **–s** option is present.

If a specified lower limit exceeds the upper limit, the axis is reversed.

Options

–a Supplies abscissas automatically and uses next two arguments to set spacing and starting point. Spacing is given by the next argument (default 1). A second optional argument is the starting point for automatic abscissas (default 0 or lower limit given by **–x**).

–b Breaks graph after each label in the input.

–c Uses specified string (next argument) as label.

–g Uses specified number (next argument) in setting up grid style: 0 (no grid), 1 (frame with ticks), and 2 (full grid). Default is 2.

–h Uses specified number (next argument) as fraction of space for height.

–l Uses specified string (next argument) as graph label.

–m Uses specified number (next argument) in setting up line mode: 0 (disconnected) and 1 (connected). Default is 1.

–r Uses specified number (next argument) as fraction of space to right before plotting.

–s Saves screen (no erase) before plotting.

–t Transposes vertical and horizontal axes.

–u Uses specified number (next argument) as fraction of space to move up before plotting.

–w Uses specified number (next argument) as fraction of space for width.

–x [**l**] Determines x axis logarithmically. Next two arguments after **l** determine lower and upper *x* limits respectively. The third argument determines grid spacing on *x* axis.

–y [**l**] Same as **x** but for **y** axis.

Restrictions

The `graph` command stores all points internally and drops those for which there is not room.

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

Logarithmic axes may not be reversed.

See Also

plot(1g), spline(1g)

Name

grep, egrep, fgrep – search file for regular expression

Syntax

grep [*option...*] *expression* [*file...*]

egrep [*option...*] [*expression*] [*file...*]

fgrep [*option...*] [*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.

The `grep` command patterns are limited regular expressions in the style of `ex`(1), which uses a compact nondeterministic algorithm. The `egrep` command patterns are full regular expressions. The `egrep` command uses a fast deterministic algorithm that sometimes needs exponential space. The `fgrep` command patterns are fixed strings. The `fgrep` command is fast and compact.

In all cases the file name is shown if there is more than one input file. Take care when using the characters $ * [^ | ( ) and \ in the *expression* because they are also meaningful to the Shell. It is safest to enclose the entire *expression* argument in single quotes ´ ´.

The `fgrep` command searches for lines that contain one of the (new line-separated) *strings*.

The `egrep` command accepts extended regular expressions. In the following description 'character' excludes new line:

> A \ followed by a single character other than new line matches that character.
>
> The character ^ matches the beginning of a line.
>
> The character $ matches the end of a line.
>
> A **.** (dot) matches any character.
>
> A single character not otherwise endowed with special meaning matches that character.
>
> A string enclosed in brackets [ ] matches any single character from the string. Ranges of ASCII character codes may be abbreviated as in 'a–z0–9'. A] may occur only as the first character of the string. A literal – must be placed where it can't be mistaken as a range indicator.
>
> A regular expression followed by an * (asterisk) matches a sequence of 0 or more matches of the regular expression. A regular expression followed by a + (plus) matches a sequence of 1 or more matches of the regular expression. A regular expression followed by a ? (question mark) matches a sequence of 0 or 1 matches of the regular expression.
>
> Two regular expressions concatenated match a match of the first followed by a match of the second.

Two regular expressions separated by | or new line match either a match for the first or a match for the second.

A regular expression enclosed in parentheses matches a match for the regular expression.

The order of precedence of operators at the same parenthesis level is the following: [ ], then *+?, then concatenation, then | and new line.

Options

–b Precedes each output line with its block number. This is sometimes useful in locating disk block numbers by context.

–c Produces count of matching lines only.

–e *expression* Uses next argument as expression that begins with a minus (–).

–f *file* Takes regular expression (**egrep**) or string list (**fgrep**) from *file*.

–i Considers upper and lowercase letter identical in making comparisons (`grep` and `fgrep` only).

–l Lists files with matching lines only once, separated by a new line.

–n Precedes each matching line with its line number.

–s Silent mode and nothing is printed (except error messages). This is useful for checking the error status (see DIAGNOSTICS).

–v Displays all lines that do not match specified expression.

–w Searches for an expression as for a word (as if surrounded by '\<' and '\>'). For further information, see `ex(1)`, `grep` only.

–x Prints exact lines matched in their entirety (`fgrep` only).

Restrictions

Lines are limited to 256 characters; longer lines are truncated.

Diagnostics

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

See Also

ex(1), sed(1), sh(1)

groups(1)

Name

groups – show group memberships

Syntax

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 you do belong to the group by which it is owned, you are granted group access to the file.

When a new file is created it is given the group specifications of the directory in which it resides.

Files

`/etc/passwd`
`/etc/group`

See Also

setgroups(2)

Name

head – give first few lines

Syntax

head [–*count*] [*file*...]

Description

This filter gives the first *count* lines of each of the specified files, or of the standard input. The default number of lines displayed is 10, if *count* is omitted.

See Also

tail(1)

hostid(1)

Name

hostid – set or print identifier of current host system

Syntax

hostid [*identifier*]

Description

The `hostid` command prints 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. The super-user can set the hostid by giving a hexadecimal argument. This is usually done in the startup script `/etc/rc.local`.

See Also

gethostid(2), sethostid(2)

Name

hostname – print system name

Syntax

hostname [*nameofhost*]

Description

The `hostname` command prints the name of the current host, as given before the ''login'' prompt. The super-user can set the hostname by giving an argument; this is usually done in the startup script `/etc/rc.local`.

See Also

gethostname(2), sethostname(2)

Name

ic – compiler for language support database

Syntax

ic [**–D***name=def*] [**–U***name*] [**–I***dir*] [**–v**] [**–o** *output*] [*source*]

Description

The command `ic` generates a binary international database from a database language source file. The command either accepts its input from the file *source* or from the standard input, if you do not specify *source* .

The name of the output file is the name of the codeset in the source file or the name you specify using the `-o` option.

For information on creating a database language source file, see the *Guide to Developing International Software*.

Options

–D Defines *name* to the C preprocessor. This option has the same effect as including the `#define name` directive at the head of your source file. The default *name* definition is 1.

–U Removes any initial preprocessor definition of *name* .

–I Causes the `ic` compiler to search the named directory for files you name in an `#include` directive.

–o Specifies the name you want `ic` to use for the output file. By default, the compiler uses the name of the codeset in the source file to name the output file.

–v Requests statistics on the number of simple and double letters in the source file, the number of tables in the source file, and the size of the output binary file.

Restrictions

The length of the table name modifier is limited to 44 characters.

Examples

The following command causes the `ic` compiler to compile the `GER_CH.8859.in` source file:

```
% ic -v GER_CH.8859.in
INTLINFO database GER_CH.8859:
        257 code table entries (256 simple/1 multi-byte).
          1 property table(s).
          1 collation table(s).
          1 string table(s).
          3 conversion tables: toascii, tolower, toupper.
5051 bytes total length.
```

The `ic` compiler searches for the `GER_CH.8859.in` file in the current working directory. The compiler writes compilation statistics to `stderr`, as requested by the `-v` option. The compiler creates a binary file, named `GER_CH.8859`, in the current

working directory.

Return Values

The `ic` compiler returns zero exit status for successful compilation; it returns nonzero status if it encounters errors that inhibit generation of a binary file.

Diagnostics

The `ic` compiler issues four types of messages. The following list describes each of the four types:

warning The compiler has detected syntax that may be in error, but does not adversely effect the binary file.

error nn The compiler has detected an error severe enough to inhibit the generation of a correct binary file.

fatal error The compiler has detected an error that makes it impossible to proceed with the compilation. This error most often occurs during compilation of the code table.

fatal bug This occurs when there are internal errors in the compiler. For example, this is generally produced when an incompatible source file is given as an input to `ic`.

Files

`/tmp/icXXXXXX` Temporary files

`/lib/cpp` C preprocessor

See Also

intro(3int), setlocale (3), environ (5int), lang (5int), nl_langinfo(5int)
Guide to Developing International Software

Name

iconv – international codeset conversion

Syntax

iconv [**–d**] **-f** *fromcodeset* **-t** *tocodeset* [*file...*]

Description

The `iconv` command converts the encoding of characters in its input from one codeset to another codeset. The *fromcodeset* argument specifies the codeset used to encode the data in the input; that is, it specifies the input codeset. The *tocodeset* argument specifies the codeset to which you want the input data converted; that is, it specifies the output codeset. The `iconv` command performs the conversion by reading rules from a conversion table you create. The command reads its input from standard input or from one or more files named on the command line. The command writes its output to standard output.

You define conversion rules in a conversion table. The conversion rules specify how `iconv` converts a particular character or group of characters, which are called tokens. The conversion table is a text file that contains two lists. In the left-hand list, you specify each token you want `iconv` to convert. In the right-hand list, you specify the token you want `iconv` to create in the output file. For example, if you issued the following command:

```
% iconv -fupper -tlower conversion_file
```

This command uses the conversion table located in the file `/usr/lib/intln/conv/upper_lower`, that specifies how to convert from an uppercase codeset to a lowercase codeset. The following shows part of the conversion table:

```
#
# Converts from uppercase to lowercase
#
#      Input token        Output token
#      -------------      ------------------
            A                    a
            B                    b
            C                    c
            D                    d
            E                    e
            F                    f
            G                    g
          .
          .
          .
            Z                    z
#
# Convert tabs to spaces using octal
#
          \011                    \040\040\040
#
# Convert the A umlaut to lowercase
#
             Ä                        ä
```

Each line in the conversion table must contain two strings, an input token and an output token. The tokens must be delimited with spaces or tabs. The backslash character (\ ) either causes the `iconv` command to recognize a character it normally ignores or introduces a three digit octal constant. All octal constants in the conversion table must contain three digits. Lines that begin with a hash symbol (#) are comments. The `iconv` command ignores comment lines and blank lines.

You name the conversion table file using the name of the input codeset, an underscore, and the name of the output codeset. For example, if your input codeset is ISO646 and your output codeset is ISO8859, you might name the conversion table file `646_8859`.

The `iconv` command searches for the conversion table file in the directory specified by the `${ICONV}/conv` pathname. If the ${ICONV} environment variable is undefined, the `iconv` command searches the `/usr/lib/intln/conv` directory.

The operation of the `iconv` command is 8-bit transparent.

Options

–d Deletes any characters that are omitted from the conversion table. By default, the `iconv` command sends characters that are omitted to the output file without modifying them.

–f Specifies the name of the input codeset.

–t Specifies the name of the output codeset.

Restrictions

The conversion table file name can contain no more than 255 characters. You may need to truncate the name of the input codeset or output codeset when you name the conversion table file.

Examples

The following shows an example of using the `iconv` command:

This command converts the data in `mydatafile` from ISO646 encoding to ISO8859 encoding. The `iconv` command reads the conversion table from the `${ICONV}/conv/646_8859` file. If the ${ICONV} environment variable is undefined, the `iconv` command uses the `/usr/lib/intln/conv/646_8859` file. If that file does not exist, the `iconv` command issues an error message and does not convert the data file. The `iconv` command writes the results of any conversion it performs to the file `newdatafile`

Files

`/usr/lib/conv/`*fromcodeset_tocodeset*
`${ICONV}/conv/`*fromcodeset_tocodeset*

See Also

environ(5int)
Guide to Developing International Software

id(1)

Name

id – print user and group ID and names

Syntax

id [*–gnru*]

Description

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

If multiple groups are supported by the underlying system, the supplementary group affiliations of the invoking process are also written.

When no options are specified, the standard format of output produced by `id` is

<real user id>, <user-name>, <real group id>, <group-name>

Options

–g Outputs only the group ID. The default format is **%d\n**. This may be modified by the **–n** option. The default group ID is the effective group ID; this may be modified by the **–r** option.

–n Outputs the name in the format **%s\n** instead of the numeric ID when the **–u** or **–g** options are used.

–r Outputs the real ID instead of the effective ID when the **–u** or **–g** options are used. There is no option to produce a list of supplementary group IDs alone.

–u Outputs only the user ID. The default output format is **%d0fP**. This may be modified with the **–n** option. The default user ID is the effective user ID; this may be modified by the **-r** option.

See Also

logname(1), getuid(2)

Name

inc – incorporate new mail

Syntax

inc [*+foldername*] **[–audit** *audit–file*] **[–noaudit] [–changecur] [–nochangecur] [–form** *formatfile*] **[–format** *string*] **[–file** *name*] **[–silent] [–nosilent] [–truncate] [–notruncate] [–width** *columns*] **[–help]**

Description

Use `inc` to incorporate mail from your incoming mail drop into an `MH` folder. If `+folder` is not specified, the folder named `+inbox` in your `MH` directory will be used.

The new messages being incorporated are numbered sequentially starting with the next highest available number in the folder. If the specified (or default) folder does not exist, `inc` will ask you whether you want to create it. As the messages are processed, a `scan` listing of the new mail is produced. See `scan`(1mh) for details of the listing produced.

If your `mh_profile` contains a `Msg-Protect: nnn` entry, it will be used as the protection on the newly created messages, otherwise the `MH` default of 0600 will be used. This means that messages created will be read and write for the user only. During all operations on messages, this initially assigned protection will be preserved for each message, so `chmod` may be used to set a protection on an individual message, and its protection will be preserved thereafter.

Options

If the switch `-audit audit-file` is specified (usually as a default switch in the profile), then `inc` will append a header line and a line per message to the end of the specified audit-file with the format:

```
inc date
        <scan line for first message>
        <scan line for second message>
                        <etc.>
```

This is useful for keeping track of the volume and source of incoming mail.

Note that `inc` will incorporate even improperly formatted messages into your MH folder, inserting a blank line prior to the offending component and printing a comment identifying the bad message.

In all cases, your mail drop will be zeroed, unless the `-notruncate` switch is given.

If the profile entry `Unseen-Sequence` is present and non–empty, then `inc` will add each of the newly incorporated messages to each sequence named by the profile entry. This is similar to the `Previous-Sequence` profile entry supported by all `MH` commands which take `<msgs>` or `<msg>` arguments. Note that `inc` will not zero each sequence prior to adding messages.

The -form *formatfile*, and the -format *string*, and -width *columns* switches allow you to override the default output format of inc. See scan(1mh) for more details of these options.

By using the -file *name* switch, you can direct inc to incorporate messages from a file other than your maildrop. Note that the named file will not be zeroed, unless the -truncate switch is given.

If the environment variable $MAILDROP is set, then inc uses it as the location of your maildrop instead of the default. However the -file *filename* switch overrides this. If this variable is not set, then inc will consult the profile entry maildrop for this information. If the value found is not absolute, then it is interpreted relative to your MH directory. If the value is not found, then inc will look in the standard system location for your maildrop.

The -silent switch directs inc to be quiet and not ask any questions at all. This is useful for putting inc in the background and going on to other things.

The argument to the -format switch must be interpreted as a single token by the shell that invokes inc. Therefore, you should usually place the argument to this switch inside double–quotes.

The defaults for this command are:

+folder defaults to inbox
-noaudit
-changecur
-format defaults as described above
-nosilent
-truncate if -file name not given, -notruncate otherwise
-width defaulted to the width of the terminal

The folder into which messages are being incorporated will become the current folder. The first message incorporated will become the current message, unless the -nochangecur option is specified. This leaves the context ready for a show of the first new message.

Files

$HOME/.mh_profile	The user profile
/usr/new/lib/mh/mtstailor	tailor file
/usr/spool/mail/$USER	Location of mail drop

Profile Components

Path:	To determine your MH directory
Alternate-Mailboxes:	To determine your mailboxes
Folder-Protect:	To set mode when creating a new folder
Msg-Protect:	To set mode when creating a new message and audit–file
Unseen-Sequence:	To name sequences denoting unseen messages

See Also

chmod(1mh), mhmail(1mh), scan(1mh), mh-mail(5mh), post(8mh)

Name

indent – indent and format C program source

Syntax

indent *input* [*output*] [*flags*]

Description

The `indent` command is intended primarily as a C program formatter. Specifically, `indent` indents code lines, aligns comments, inserts spaces around operators where necessary and breaks up declaration lists as in ''int a,b,c;''.

The `indent` command does not break up long statements to make them fit within the maximum line length, but it does flag lines that are too long. Lines are broken so that each statement starts a new line, and braces appear alone on a line. Also, an attempt is made to line up identifiers in declarations.

The *flags* that can be specified follow. They can appear before or after the file names. If the *output* file is omitted, the formatted file is written back into *input* and a ''backup'' copy of *input* is written in the current directory. If *input* is named ''/blah/blah/file'', the backup file is named ''.Bfile''. If *output* is specified, `indent` checks to make sure it is different from *input*.

Options

The following options are used to control the formatting style imposed by `indent`:

–l*nnn* Determines maximum length of output line. The default is 75.

–c*nnn* Determines column in which comments start. The default is 33.

–cd*nnn* Determines column in which comments on declarations start. The default is for these comments to start in the same column as other comments.

–i*nnn* Determines number of spaces for one indentation level. The default is 4.

–dj,–ndj Causes declarations to be left justified. **–ndj** causes 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` reports when it splits one line of input into two or more lines of output, and it gives some size statistics at completion. The default is **–nv**.

–bc,–nbc Forces newline after each comma in a declaration. **–nbc** turns off this option. The default is **–bc**.

–d*nnn* Controls the placement of comments which are not to the right of code. Specifying **–d2** means that such comments are placed two indentation levels to the left of code. The default **–d0** lines up these comments with the code. See the section on comment indentation below.

–br,–bl Specifying **–bl** causes complex statements to be lined up in a space order. For example,

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

Specifying **–br** (the default) makes 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 login 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, always override profile switches. The profile file must be a single line of not more than 127 characters. The switches should be separated on the line by spaces or tabs.

Multiline expressions

The `indent` command does not break up complicated expressions that extend over multiple lines. However, it usually indents such expressions that have already been broken up correctly. Such an expression might look like the following:

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

Comments

The `indent` command recognizes the following four kinds of comments:

1) straight text
2) ''box'' comments
3) UNIX-style comments
4) comments that should be passed through unchanged

The comments are interpreted as follows:

''Box'' comments The `indent` command assumes that any comment with a dash immediately after the start of comment (i.e. ''/*–'') is a comment surrounded by a box of stars. Each line of such a comment is left unchanged, except that the first non-blank character of each successive line is lined up with the beginning slash of the first line. Box comments are 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 are treated similarly to box comments, except the first non-blank character on each line is lined up with the '*' of the ''/*''.

Unchanged comments — Any comment which starts in column 1 is 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* fits as many words (separated by blanks, tabs, or new lines) on a line as possible. Straight text comments are indented.

Comment indentation

Box, UNIX-style, and straight text comments may be indented. If a comment is on a line with code it is started in the ''comment column'', which is set by the **–cnnn** command line parameter. Otherwise, the comment is 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 is never be placed in column 1.) If the code on a line extends past the comment column, the comment is moved to the next line.

Restrictions

Does not know how to format ''long'' declarations.

Diagnostics

Diagnostic error messages, mostly to tell that a text line has been broken or is too long for the output line.

Files

.indent.pro profile file

Name

install – install binaries

Syntax

install [**–c**] [**–m** *mode*] [**–o** *owner*] [**–g** *group*] [**–s**] *binary destination*

Description

The *binary* is moved to *destination*. If *destination* already exists, it is removed before *binary* is moved. If the destination is a directory then *binary* is moved into the *destination* directory with its original file-name.

The `install` command refuses to move a file onto itself.

Options

–c Moves or copies binary to *destination.*

–g *group* Specifies a different group from group staff for *destination.* The *destination* is changed to group system; the **–g** *group* option may be used to specify a different group. The user must belong to the specified group and be the owner of the file or the superuser.

–m *mode* Specifies a different mode from the standard 755 for *destination.*

–o *owner* Specifies a different owner from owner root for *destination.* The *destination* is changed to current owner. The **–o** *owner* option may be used to specify a different owner, but only the superuser can change the owner.

–s Strips the binary after it is installed.

See Also

chgrp(1), chmod(1), cp(1), mv(1), strip(1), chown(8)

Name

invcutter – generate subset inventory files

Syntax

/usr/sys/dist/invcutter [**–d**] [**–f** *root-path*] [**–f** *version-code*]

Description

The `invcutter` command reads master inventory records from standard input. A subset inventory record is written to standard output for every record read from the input. The information contained in the output record is derived from the input record and the file attribute information in the file hierarchy rooted in the current directory.

Options

–d	Enable debugging. No useful diagnostics are printed.
–f *root-path*	Specify an alternate root path for finding file attribute information.
–v version-code	Specify a 3-digit version code for use in the version field of the output records. The default version code is *010.*

Restrictions

All input records must be sorted in ascending order on the pathname field.

Files described in an input record which exist as sockets in the file hierarchy are not processable.

If a file is described in an input record has a link count greater than 1, all other links to the file must be represented in the input.

Examples

The following command will generate inventory records for the master inventory entries in *PDS020.mi* containing version fields set to *020:*

```
invcutter -v 020 < PDS020.mi
```

Return Value

An exit status of 0 is returned if all goes well. An exit status of 1 is returned if an error occurs. See Diagnostics.

Diagnostics

"cannot chdir to *pathname* (*error-message*)"
The program cannot change directories to the *pathname* directory specified with the **–f** option. The *error-message* will provide additional information.

"sort error, record #*n*"
The *n*th input record is not in the correct sort order. All input records must be in ascending ASCII colating sequence on the pathname field.

"**cannot stat** *filename* (*error-message*)"
An error has occurred attempting to read the atributes of *filename*. The *error-message* explains exactly what happened.

"*pathname*: **illegal file type code 0140000**"
The file *pathname* is a socket. Sockets are not supported as valid file types for distribution.

"**unresolved nlink** *n: pathname*"
This indicates that file *pathname* in the master inventory is linked to *n* files which do not appear in the master inventory. Check inventory for validity with the `newinv` program.

"*n* **unresolved hard links**"
This is an informational message stating how many files were detected in the input inventory which had unresolved links.

See Also

newinv(1), stl_inv(5), stl_mi(5)
Guide to Preparing Software for Distribution on ULTRIX Systems

Name

iostat – report I/O statistics

Syntax

iostat [**–c**] [**–t**] [*disknames*] [*interval*] [*count*]

Description

The `iostat` command reports I/O statistics for terminals, disks and cpus. For terminals the number of input and output characters are counted. For disks the number of 512 byte blocks per second and number of transfers per second are displayed. For cpus, it provides the percentage of time the system has spent in user mode, in user mode running low priority (niced) processes, in system mode, and idling. On multiprocessor systems these cpu statistics represent a cumulative summary of all the cpus.

The optional *disknames* argument causes disk statistics to be displayed for the specified disks. If this argument is not specified then disk statistics will be displayed for the first 3 disks only.

The optional *interval* argument causes `iostat` to report once each *interval* seconds. The first report is for all time since a reboot and each subsequent report is for the last interval only.

The optional *count* argument restricts the number of reports.

Options

–c Displays the percentage of time each cpu spent in user mode, running low priority (nice'd) processes, in system mode, and idling.

–t Displays the number of characters read from and written to terminals.

Examples

This example will cause cpu and disk statistics for the 5 disks ra0, ra1, ra2, ra3, and ra4.

```
iostat ra0 ra1 ra2 ra3 ra4
```

This example will cause cpu, terminal, and disk statistics for ra0 to be displayed and updated every 2 seconds.

```
iostat -t ra0 2
```

Files

/dev/kmem
/vmunix

See Also

vmstat(1), cpustat(1)

ipcrm(1)

Name

ipcrm – remove a message queue, semaphore set

Syntax

ipcrm [*options*]

Description

The `ipcrm` command removes one or more specified messages, semaphores or shared memory identifiers.

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

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.

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

Syntax

ipcs [*options*]

Description

The ipcs command provides information about active, interprocess communication facilities, message queues, shared memory, and semaphores that are currently active in the system.

Options

The information is displayed in columns and is controlled by the following *options*:

–m Displays information about active shared memory segments

–q Displays information about active message queues

–s Displays information about active semaphores

If any of the options **–q, –m,** or –s are specified, information about only those indicated are printed. If none of these three is specified, information about all three are printed.

–a Uses all print *options* (shorthand notation for **–b, –c, –o, –p** and **–t**)

–b Displays the biggest allowable size information (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)

–C Uses the specified core file (next argument) in place of `/dev/kmem`

–c Displays creator's login name and group name

–N Uses the specified *namelist* (next argument) in place of `/vmunix`

–o Displays the outstanding usage information (number of messages in queue, size of each and number of processes attached to shared memory segments)

–p Displays the process ID 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)

–t Displays all time statistics (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)

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

T (all)
: Type of 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. The first two characters are interpreted as follows:

R
: If the 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 disappears when the last process attached to the segment detaches it.

C
: If the associated shared memory segment is to be clear 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 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 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.

Restrictions

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

Files

/vmunix	system namelist
/dev/kmem	memory
/etc/passwd	user names
/etc/group	group names

See Also

ipcrm(2), msgop(2), semop(2), shmop(2)

Name

join – join files

Syntax

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

Description

The `join` command compares a field in *file1 to a field in file2*. If the two fields match, the `join` command combines the line in *file1* that contains the field with the line in *file2* that contains the field. The command writes its output to standard output. If you specify a hyphen (-) in the *file1* argument, `join` compares standard input to the contents of *file2*.

The `join` command compares and combines the input files one line at a time. Each line in the input file contains one field that `join` uses to determine if two lines should be joined. This field is called the join field. By default, the `join` command uses the first field in each line as the join field. The command compares the join field in the first line of *file1* to the join field in the first line of *file2*. If the two fields match, the command joins the lines. The command then compares the join fields in the second line of both files, and so on.

In the input files, fields are separated by tab or space characters. The `join` command reads data from the first field until it encounters a tab or space character, which terminates the first field. By default, the command ignores tab and space characters, so the next character that is not a tab or space begins the second field. The second field is terminated by the tab or space that follows it, and the third field begins with the next character that is not a tab or space. The `join` command reads fields in this way until it encounters a new line character. Any number of tabs or spaces can separate two fields, and any number of newline characters can separate two lines.

Both *file1* and *file2* must be ordered in the collating sequence of the `sort`-b command on the fields that the two files are to be joined. By default, `join` uses the first field in each line and collates the same as `sort -b`.

To create output, the `join` command writes the join field, followed by the remaining fields in the line from *file1*, followed by the remaining fields in the line from *file2* to the output file. The following demonstrates how lines in the output appear by default:

```
join_field file1.field2 file1.field3 file1.field4 file2.field2 file2.field3
```

By default, the `join` command ignores lines that do not contain identical join fields. The command writes no output for these lines.

You can change how `join` creates output using command options. For example, you can cause the command to write output for lines that do not contain identical join fields. You can also specify a *list* using the `-o` option. In *list,* you supply a list of specifiers in the form *file.field,* where *file* is either 1 or 2 and *field* is the number of the field. For example, 1.2 specifies the second field in the first file and 2.4 specifies the fourth field in the second file. The following demonstrates how lines in the output appear if you use these two specifiers:

```
file1.field2 field2.field4
```

International Environment

LC_COLLATE If this environment variable is set and valid, `join` uses the international language database named in the definition to determine collation rules.

LC_CTYPE If this environment variable is set and valid, `join` uses the international language database named in the definition to determine character classification rules.

LANG If this environment variable is set and valid `join` uses the international language database named in the definition to determine collation and character classification rules. If LC_COLLATE or LC_CTYPE is defined their definition supercedes the definition of LANG.

Options

-a[*n*] Write lines that contain unmatched join fields to the output file. You can cause the command to write unmatched lines from only one file using *n*. If you specify 1 in *n*, `join` writes unmatched lines only from file 1. If you specify 2, `join` writes unmatched lines only from file 2.

If you omit the `-a` option, `join` writes no output for unmatched lines.

-e *s* Writes the string you specify in *s* to the output if you specify a nonexistent field in the *list* for the `-o` option. For example, if lines in file 2 contain only three fields, and you specify 2.4 in *list*, `join` writes *s* in place of the nonexistent field.

-j*n m* Defines field *m* in file *n* to be the join field. The `join` command compares the field you specify in the `-j` option to the default join field in the other file. If you omit *n*, the `join` command uses the *m*th field in both files.

-1 *m* Use the *m* th field in the first file as the join field. This option is equivalent to using `-j1` *m*.

-2 *m* Use the *m* field in the second file as the join field. This option is equivalent to using `-j2` *m*.

-o *list* Output the joined data according to *list*. The specifiers in *list* have the format *file.field*, where *file* is either 1 or 2 and *field* is the number of the field.

-t*c* Recognize the tab character *c*. The presence of *c* in a line is significant, both for comparing join fields and creating output.

Restrictions

If you specify the `-t` option, the `join` command collates the same as `sort` with no options.

Examples

Suppose that by issuing the following cat commands, you display the files shown in the example:

```
% cat file_1
apr     15
aug     20
dec     18
feb     05
% cat file_2
apr     06
aug     14
date
feb     15
```

Both files are sorted in ascending order.

If you issue the join command without options, the output appears as follows:

```
% join file_1 file_2
apr 15 06
aug 20 14
feb 05 15
```

The third line in each input file is not joined in the output because the join fields (date and dec) do not match.

To join the lines in these files and format the output so that the second field from each file appears first and the first (join) field appears second, issue the following command:

```
% join -o 1.2 1.1 2.2 2.1 file_1 file_2
15 apr 06 apr
20 aug 14 aug
05 feb 15 feb
```

To write lines that are unmatched to the output, issue the following command:

```
% join -a file_1 file_2
apr 15 06
aug 20 14
date
dec 18
feb 05 15
```

See Also

awk(1), comm(1), sort(1), sort5(1), environ(5int)

Name

kill – send a signal to a process

Syntax

kill [*–sig*] *processid...*
kill –l

Description

The `kill` command sends the TERM (terminate, 15) signal to the specified processes. If a signal name or number preceded by '–' is given as first argument, that signal is sent instead of terminate. For further information, see `sigvec`(2).

The terminate signal kills processes that do not catch the signal; 'kill –9 ...' is a sure kill, as the KILL (9) signal cannot be caught. By convention, if process number 0 is specified, all members in the process group (that is, processes resulting from the current login) are signaled. This works only if you use `sh`(1) and not if you use `csh`(1). To kill a process it must either belong to you or you must be superuser.

The process number of an asynchronous process started with '&' is reported by the shell. Process numbers can also be found by using `csh`(1). It allows job specifiers "%..." so process ID's are not as often used as `kill` arguments. See `csh`(1) for details.

Options

–l Lists signal names. The signal names are listed by 'kill –l', and are as given in /usr/include/signal.h, stripped of the common SIG prefix.

See Also

csh(1), ps(1), kill(2), sigvec(2)

Name

kits – generate setld format distribution kits

Syntax

/usr/sys/dist/kits *key-file input-path output-path* [*subset...*]

Description

The `kits` command produces subset images, inventories, and control files for an installation using the `setld` command. You need to know the key file which describes the product to be built, a hierarchy from which the component files to be kitted are to be taken, and a destination directory into which the kit information is to be placed.

The `kits` command produces a subset image and a .image file in the *output-path* directory for each subset. In the *instctrl* subdirectory of *output-path,* `kits` produces an inventory file and a control file. Any subset control program for the subset is transferred to *output-path/instctrl.* An *instctrl* directory is created if none existed.

Arguments

key-file
: The path name of the manufacturing key file which describes the product to be kitted. Unless optional *subset* arguments are specified, all subsets listed in the descriptor section of the *key-file* are kitted.

input-path
: The path name which specifies the top of a hierarchy of files. This hierarchy contains the files which are to be kitted into subsets.

output-path
: The name of the directory to be used to store the subset image and data files produced by the command.

subset...
: The names of individual subsets can be specified by optionally listing them on the command line. If they are specified, only those subsets will be kitted. The `kits` program assumes that all other subsets for the product have been kitted and that their images are in the directory specified by *output-path.* The key file specified must contain descriptors for each of the optional named subsets.

Restrictions

Any subset control programs to be provided with the kit must be located in a directory *scps* in the working directory where the `kits` program is invoked. If no subset control program is found for a subset, an empty one is created.

Examples

The following example shows the command used to produce a kit using key file *ULT400.k* in the current directory to package files from the hierarchy */var/kits/input* and place the results in */var/kits/output.*

```
kits ULT400.k /var/kits/input /var/kits/output
```

The next example shows the same usage, but specifies that only the ULTACCT400 subset is to be created.

```
kits ULT400.k /var/kits/input /var/kits/output ULTACCT400
```

Diagnostics

kits: ***key-file*** **not found**
The `kits` program was unable to find the *key-file* specified on the command line.

kits: ***input-path*** **not found**
The `kits` program was unable to find the specified *input-path.*

kits: ***output-path*** **not found**
The `kits` program was unable to find the specified *output-path.*

kits: cannot create instctrl directory.
The `kits` program cannot create an instctrl directory under *output-path.* Check that the user has write permission to *output-path.*

kits: ***key-file*** **format error**
One of the NAME, CODE, VERS, MI or ROOT values in the specified *key-file* is either missing or has a null value.

Inventory file ***pathname*** **not found**
The master inventory file *pathname* specified in the MI entry of the *key-file* cannot be found. Verify that the *pathname* is accessible from the current directory.

Generating media creation information...failed.
There are no records in the master inventory file for a subset which is being kitted. Check the master inventory file for correctness of content and format.

No such subset in ***key-file*** **subset** ***subset***
A subset name specified on the command line does not have a descriptor line in the *key-file.* Check the spelling of the subset name on the command line. Check the contents of the *key-file.*

compression failed. status = ***status***
The compression option was specified in the *key-file* and an attempt to compress a subset failed. This should not happen. Run the `kits` program once more.

Files

`ts.subset*`	temporary files.
`stderr`	log of subset packaging activity

See Also

invcutter(1), tarsets(1), stl_comp(5), stl_ctrl(5), stl_image(5), stl_inv(5), stl_key(5), stl_mi(5), stl_scp(5), setld(8)
Guide to Preparing Software for Distribution on ULTRIX Systems

Name

ksh, rksh – KornShell, a standard/restricted command and programming language

Syntax

ksh [±**aefhikmnoprstuvx**] [±**o** option] ... [**–c** string] [arg ...]
rksh [±**aefhikmnoprstuvx**] [±**o** option] ... [**–c** string] [arg ...]

Description

The ksh shell is a command and programming language that executes commands read from a terminal or a file. The rksh shell is a restricted version of the command interpreter *ksh*; it is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. See **Invocation** for the meaning of arguments to the shell.

Definitions

A metacharacter is one of the following characters:

; & ( ) | < > new-line 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 functions and 'named parameters'. A word is a sequence of characters separated by one or more non-quoted metacharacters.

A command is a sequence of characters in the syntax of the shell language. The shell reads each command and carries out the desired action either directly or by invoking separate utilities. A special command is a command that is carried out by the shell without creating a separate process. Except for documented side effects, most special commands can be implemented as separate utilities.

Commands

A simple-command is a sequence of blank separated words which may be preceded by a parameter assignment list. See **Environment** below. The first word specifies the name of the command to be executed. Except as specified below, 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(2) for a list of status values).

A pipeline is a sequence of one or more commands separated by |. The standard output of each command but the last 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 last 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** option of the special commands read and print described later. The symbol **&&** (||) causes the list following it to be executed only if the preceding pipeline returns a zero (non-zero) value. An arbitrary number of new-lines may appear in a list, instead of a semicolon, to delimit a command.

A command is either a simple-command or one of the following. Unless otherwise stated, the value returned by a command is that of the last simple-command executed in the command.

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**). 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 *word*s, each preceded by a number. If **in** *word* ... is omitted, then the positional parameters are used instead (see **Parameter Substitution** below). 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 *word*s, 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 file-name generation (see **File Name Generation** below).

if *list* **;then** *list* [**elif** *list* **;then** *list*] ... [**;else** *list*] **;fi**
: The *list* following **if** is executed and, if it returns a zero exit status, 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*)
: Execute *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 below.

{ *list*;} The *list* is simply executed. Note that unlike the metacharacters (and), { and } are 'reserved words' and must at the beginning of a line or after a ; in order to be recognized.

[[*expression*]]
Evaluates *expression* and returns a zero exit status when *expression* is true. See **Conditional Expressions** for a description of *expression.*

function *identifier* { *list* ;}
identifier () { *list* ;}
Define a function which is referenced by *identifier*. The body of the function is the *list* of commands between { and }. (See **Functions** below).

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 reserved words are only recognized as the first word of a command and when not quoted:

if then else elif fi case esac for while until do done { } function select time [[ ]]

Comments

A word beginning with # causes that word and all the following characters up to a new-line 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 non-special 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 above. The first word of each command in the replaced text, other than any that are in the process of being replaced, will be tested for 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 builtin commands but cannot be used to redefine the reserved words listed above. 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 scripts invoked by name, but must be reinitialized for separate invocations of the Shell (See **Invocation** below).

Aliasing is performed when scripts are read, not while they are executed. Therefore, for an alias to take effect the alias definition command has to be executed before the command which references the alias is read.

Aliases are frequently used as a short hand for full path names. An 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** option of the `set` command makes each referenced command name into a tracked alias.

The following 'exported aliases' are compiled into the shell but can be unset or redefined:

autoload='typeset –fu'
false='let 0'
functions='typeset –f'
hash='alias –t'
history='fc –l'
integer='typeset –i'
nohup='nohup '
r='fc –e –'
true=':'
type='whence –v'

Tilde Substitution

After alias substitution is performed, each word is checked to see if it begins with an unquoted ~. If it does, then the word up 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 $PWD and $OLDPWD respectively.

In addition, tilde substitution is attempted when the value of a 'variable assignment parameter' begins with a ~.

Command Substitution

The standard output from a command enclosed in parentheses preceded by a dollar sign ($()) or a pair of grave accents (` `) may be used as part or all of a word; trailing new-lines are removed. In the second (archaic) form, the string between the quotes is processed for special quoting characters before the command is executed. (See **Quoting**). The command substitution **$(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.

An arithmetic expression enclosed in double parenthesis preceded by a dollar sign ($(())) is replaced by the value of the arithmetic expression within the double parenthesis.

Process Substitution.

This feature is only available on versions of the operating system that support the `/dev/fd` directory for naming open files. Each command argument of the form <(*list*) or >(*list*) will run 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, then the file passed as an argument will contain the output of the *list* process. For example,

paste <(cut –f1 *file1*) **<(cut –f3** *file2*) **| tee >**(*process1*) **>**(*process2*)

cuts fields 1 and 3 from the files *file1* and *file2* respectively, **pastes** the results together, and sends it to the processes *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 a system pipe so programs that expect to lseek 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 zero or more attributes. Named parameters can be assigned values and attributes by using the typeset special command. The attributes supported by the Shell are described later with the typeset special command. Exported parameters pass values and attributes to the environment.

The shell supports a 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**) followed by a]. To assign values to an array, use **set –A** *name value* .... The value of all subscripts must be in the range of 0 through 1023. 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 element zero.

The *value* of a *name*d 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 below.
Positional parameters, parameters denoted by a number, may be 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*.

${*parameter*}
: The shell reads all the characters from **${** to the matching **}** as part of the same word even if it contains braces or metacharacters. 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 non-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 to in this way.

${*parameter*:?*word*}
: If *parameter* is set and is non-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 non-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 second 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 the *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 second form the largest matching pattern is deleted.

In the above, *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 expressions, then the shell only checks 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.
- **_** Initially, the value _ is an absolute pathname of the shell or script being executed as passed in the *environment*. Subsequently it is assigned the last argument of the previous command. This parameter is not set for commands which are asynchronous. file when checking for mail.
- **!** The process number of the last background command invoked.
- ERRNO The value of errno as set by the most recently failed system call. This value is system dependent and is intended for debugging purposes.
- LINENO The line number of the current line within the script or function being executed.
- OLDPWD The previous working directory set by the `cd` command.
- OPTARG The value of the last option argument processed by the `getopts` special command.
- OPTIND The index of the last option argument processed by the `getopts` special command.
- PPID The process number of the parent of the shell.
- PWD The present working directory set by the `cd` command.

RANDOM Each time this parameter is referenced, a random integer, uniformly distributed between 0 and 32767, 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 *emacs*, *gmacs*, or *vi* and the VISUAL variable is not set, then the corresponding option (see Special Command **set** below) 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* below.) This file is typically used for *alias* and *function* definitions.

FCEDIT
: The default editor name for the **fc** command.

FPATH
: The search path for function definitions. This path is searched when a function with the **–u** attribute is referenced and when a command is not found. If an executable file is found, then it is read and executed in the current environment.

IFS Internal field separators, normally **space**, **tab**, and **new-line** that is 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* below).

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* below.)

HISTSIZE
: If this parameter is set when the shell is invoked, then the number of previously entered commands 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.

LOGNAME
: The name of the user's login account, corresponding to the login name in the user database.

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 file names. 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 file name can be followed by a **?** and a message that will be printed. The message will undergo parameter 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* below). The user may not change **PATH** if executing under `rksh` (except in *.profile*).

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 Re-entry* below).

PS2
: Secondary prompt string, by default "**>** ".

PS3
: Selection prompt string used within a **select** loop, by default "**#?** ".

PS4
: The value of this parameter is expanded for parameter substitution and precedes each line of an execution trace. If omitted, the execution trace prompt is "**+** ".

SHELL
: The pathname of the *shell* is kept in the environment. At invocation, if the basename of this variable matches the pattern ***r*sh**, 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 *emacs*, *gmacs*, or *vi* then the corresponding option (see Special Command **set** below) 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 at all 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.

File Name 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 lexicographically sorted file names that match the pattern. If no file name is found that matches the pattern, then the word is left unchanged. When a *pattern* is used for file name generation, the character **.** at the start of a file name or immediately following a /, as well as the character / 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.

A *pattern-list* is a list of one or more patterns separated by each other with a |. Composite patterns can be formed with one or more of the following:

?(*pattern-list*)
: Optionally matches any one of the given patterns.

*(*pattern-list*)
: Matches zero or more occurrences of the given patterns.

+(*pattern-list*)
: Matches one or more occurrences of the given patterns.

@(*pattern-list*)
: Matches exactly one of the given patterns.

!(*pattern-list*)
: Matches anything, except one of the given patterns.

Quoting.

Each of the *metacharacters* listed above (See *Definitions* above) 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 \. The pair **\new-line** 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 **$@** is identical when not quoted or when used as a parameter assignment value or as a file name. However, when used as a command argument, **$*** 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 quote marks (``) \ quotes the characters \, `, and

If the grave quotes occur within double quotes then \ also quotes the character ".

The special meaning of reserved words or aliases can be removed by quoting any character of the reserved word. The recognition of function names or special command names listed below 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 two and thirty-six representing the arithmetic base and *n* is a number in that base. If *base* is omitted then base 10 is used.

An arithmetic expression uses the same syntax, precedence, and associativity of expression of the C language. All the integral operators, other than ++, −−, ?:, and , are supported. Named parameters can be referenced by name within an arithmetic expression without using the parameter substitution syntax. When a named parameter is referenced, its value is evaluated as an arithmetic expression.

An internal integer representation of a *named parameter* can be specified with the **–i** option of the **typeset** special command. Arithmetic evaluation is performed on the value of each assignment to a named parameter with the **–i** attribute. If you do not specify an arithmetic base, the first assignment to the parameter determines the arithmetic base. This base is 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 new-line is typed and further input is needed to complete a command, then the secondary prompt (that is, the value of **PS2**) is issued.

Conditional Expressions.

A *conditional expression* is used with the [[compound command to test attributes of files and to compare strings. Word splitting and file name generation are not performed on the words between [[and]]. Each expression can be constructed from one or more of the following unary or binary expressions:

–a *file* True, if *file* exists.
–b *file* True, if *file* exists and is a block special file.
–c *file* True, if *file* exists and is a character special file.
–d *file* True, if *file* exists and is a directory.
–f *file* True, if *file* exists and is an ordinary file.
–g *file* True, if *file* exists and is has its setgid bit set.
–k *file* True, if *file* exists and is has its sticky bit set.
–n *string* True, if length of *string* is non-zero.
–o *option* True, if option named *option* is on.
–p *file* True, if *file* exists and is a fifo special file or a pipe.
–r *file* True, if *file* exists and is readable by current process.
–s *file* True, if *file* exists and has size greater than zero.
–t *fildes* True, if file descriptor number *fildes* is open and associated with a terminal device.

–u *file* True, if *file* exists and is has its setuid bit set.
–w *file* True, if *file* exists and is writable by current process.
–x *file* True, if *file* exists and is executable by current process. If *file* exists and is a directory, then the current process has permission to search in the directory.
–z *string* True, if length of *string* is zero.
–L *file* True, if *file* exists and is a symbolic link.
–O *file* True, if *file* exists and is owned by the effective user id of this process.
–G *file* True, if *file* exists and its group matches the effective group id of this process.
–S *file* True, if *file* exists and is a socket.

file1 **–nt** *file2*
True, if *file1* exists and is newer than *file2*.

file1 **–ot** *file2*
True, if *file1* exists and is older than *file2*.

file1 **–ef** *file2*
True, if *file1* and *file2* exist and refer to the same file.

string = *pattern*
True, if *string* matches *pattern*.

string **!=** *pattern*
True, if *string* does not match *pattern*.

string1 < *string2*
True, if *string1* comes before *string2* based on ASCII value of their characters.

string1 > *string2*
True, if *string1* comes after *string2* based on ASCII value of their characters.

exp1 **–eq** *exp2*
True, if *exp1* is equal to *exp2*.

exp1 **–ne** *exp2*
True, if *exp1* is not equal to *exp2*.

exp1 **–lt** *exp2*
True, if *exp1* is less than *exp2*.

exp1 **–gt** *exp2*
True, if *exp1* is greater than *exp2*.

exp1 **–le** *exp2*
True, if *exp1* is less than or equal to *exp2*.

exp1 **–ge** *exp2*
True, if *exp1* is greater than or equal to *exp2*.

In each of the above expressions, if *file* is of the form **/dev/fd/***n*, where *n* is an integer, then the test applied to the open file whose descriptor number is *n*.

A compound expression can be constructed from these primitives by using any of the following, listed in decreasing order of precedence.

(*expression*)
True, if *expression* is true. Used to group expressions.

! *expression*
True if *expression* is false.

expression1 **&&** *expression2*
True, if *expression1* and *expression2* are both true.

expression1 **||** *expression2*

True, if either *expression1* or *expression2* is true.

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 below. File name 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. If the file exists, and the **noclobber** option is on, this causes an error; otherwise, it is truncated to zero length.

>|*word* Sames as >, except that it overrides the **noclobber** option.

>>*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* Open file *word* for reading and writing as standard input.

<<[–]*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 file name 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, **new-line** 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 file descriptor *digit* (see *dup*(2)). Similarly for the standard output using >**&** *digit*.

<&– The standard input is closed. Similarly for the standard output using >**&–**.

<&p The input from the co-process is moved to standard input.

>&p The output to the co-process is moved to standard output.

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>*fname* 2>&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 file descriptor 1 had been) 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:

```
TERM=450 cmd args                          and
(export TERM; TERM=450; cmd args)
```

are equivalent (as far as the above execution of *cmd* is concerned).

If the **–k** flag is set, *all* parameter assignment arguments are placed in the environment, even if they occur after the command name. The following first prints **a=b c** and then **c:**

```
echo a=b c
set –k
echo a=b c
```

This feature is intended for use with scripts written for early versions of the shell and its use in new scripts is strongly discouraged. It is likely to disappear someday.

Functions.

The **function** reserved word, described in the *Commands* section above, 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* below).

Functions execute in the same process as the caller and share all files and present working directory with the caller. Traps caught by the caller are reset to their default action inside the function. A trap condition that is not caught or ignored by the function causes the function to terminate and the condition to be passed on to the

caller. A trap on EXIT set inside a function is executed after the function completes in the environment of the caller. 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** or **+f** option of the **typeset** special command. The text of functions will also be listed with **–f**. 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 specified in the ENV file with the **–xf** option of **typeset**

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.

This paragraph and the next require features that are not in all versions of the UNIX operating system and may not apply. If you are running a job and wish to do something else you may hit the key **^Z** (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 **^Z** takes effect immediately and is like 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 like they do when they try to read input.

There are several ways to refer to jobs in the shell. A job can be referred to by the process id of any process of the job or by one of the following:
%number The job with the given number.
%string Any job whose command line begins with *string*.
%?string Any job whose command line contains *string*.
%% Current job.
%+ Equivalent to *%%*.
%– Previous job.

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.

When the monitor mode is on, each background job that completes triggers any trap set for **CHLD**.

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 do this 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 **&** and job **monitor** option is not active. Otherwise, signals have the values inherited by the shell from its parent (but see also the **trap** command below).

Execution.

Each time a command is executed, the above substitutions are carried out. If the command name matches one of the *Special Commands* listed below, 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:** (specifying **/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. A sub-shell is spawned to read it. All non-exported 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 executed in a sub-shell without removing non-exported 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 giving 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 re-executed 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 re-execute the most recent command which starts with the letter **c**, replacing the first occurrence of the string **bad** with the string **good**.

In-line Editing Options

Normally, each command line entered from a terminal device is simply typed followed by a new-line ('RETURN' or 'LINE FEED'). If either the **emacs**, **gmacs**, or **vi** option is active, the user can edit the command line. To be in either of these edit modes **set** the corresponding 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 carriage return without line feed and that a space (' ') must overwrite the current character on the screen. 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 about the cursor. The mark is a > (<, *) if the line extends on the right (left, both) side(s) of the window.

The search commands in each edit mode provide access to the history file. Only strings are matched, not patterns, although a leading ^ in the string restricts the match to begin at the first character in the line.

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 **^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**. This is entered by depressing 'f' while holding down the 'CTRL' (control) key. The 'SHIFT' key is *not* depressed. (The notation **^?** indicates the DEL (delete) key.)

The notation for escape sequences is **M-** followed by a character. For example, **M-f** (pronounced Meta f) is entered by depressing ESC (ascii **033**) followed by 'f'. (**M-F** would be the notation for ESC followed by 'SHIFT' (capital) 'F'.)

All edit commands operate from any place on the line (not just at the beginning). Neither the "RETURN" nor the "LINE FEED" key is entered after edit commands except when noted.

^F	Move cursor forward (right) one character.
M-f	Move cursor forward one word. (The emacs 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 forward to character *char* on current line.
M-^]*char*	Move cursor back to character *char* on current line.
^X^X	Interchange the cursor and mark.
erase	(User defined erase character as defined by the *stty*(1) command, usually **^H** or #.) Delete previous character.
^D	Delete current character.
M-d	Delete current word.
M-^H	(Meta-backspace) Delete previous word.
M-h	Delete previous word.
M-^?	(Meta-DEL) Delete previous word (if your interrupt character is **^?** (DEL, 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 lower case.
^K	Delete from the cursor to the end of the line. If preceded by a numerical parameter whose value is less than the current cursor position, then delete from given position up to the cursor. If preceded by a numerical parameter whose value is greater than the current cursor position, then delete from cursor up to given cursor position.
^W	Kill from the cursor to the mark.
M-p	Push the region from the cursor to the mark on the stack.
kill	(User defined kill character as defined by the stty command, usually **^G** or @.) Kill the entire current line. If two *kill* characters are entered in succession, all kill characters from then on cause a line feed (useful when using paper terminals).
^Y	Restore last item removed from line. (Yank item back to the line.)
^L	Line feed and print current line.
^@	(Null character) Set mark.
M-*space*	(Meta space) Set mark.
^J	(New line) Execute the current line.
^M	(Return) Execute the current line.
eof	End-of-file character, normally **^D**, is processed as an End-of-file only if the current line is null.
^P	Fetch previous command. Each time **^P** is entered the previous command back in time is accessed. Moves back one line when not on the first line of a multi-line command.
M-<	Fetch the least recent (oldest) history line.
M->	Fetch the most recent (youngest) history line.
^N	Fetch next command line. Each time **^N** is entered the next command line 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. *String* is terminated by a "RETURN" or "NEW LINE". If string is preceded by a ^, the

matched line must begin with *string*. 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 current line from the history file.

M-*digits* (Escape) Define numeric parameter, the digits are taken as a parameter to the next command. The commands that accept a parameter are **^F**, **^B**, *erase*, **^C**, **^D**, **^K**, **^R**, **^P**, **^N**, **^]**, **M-.**, **M-^]**, **M-_**, **M-b**, **M-c**, **M-d**, **M-f**, **M-h** **M-l** and **M-^H**.

M-*letter* Soft-key – Your alias list is searched 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 meta-functions. **M-]***letter* Soft-key – Your alias list is searched 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 can be used to program functions keys on many terminals.

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 rather than the last word.

M-_ Same as **M-.**.

M-* Attempt file name generation on the current word. An asterisk is appended if the word doesn't match any file or contain any special pattern characters.

M-ESC File name completion. Replaces the current word with the longest common prefix of all filenames matching the current word with an asterisk appended. If the match is unique, a / is appended if the file is a directory and a space is appended if the file is not a directory.

M-= List files matching current word pattern if an asterisk were appended.

^U Multiply 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.

M-# Insert a # at the beginning of the line and execute it. This causes a comment to be inserted in the history file.

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 ESC (**033**) and 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 and it contains any control characters or less than one second has elapsed since the prompt was printed. The ESC 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	(User defined erase character as defined by the stty command, usually **^H** or **#**.) Delete previous character.
^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 **^V**. The **^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 end of word.
[*count*]**E**	Cursor to end of the current blank delimited word.
[*count*]**h**	Cursor backward (left) one character.
[*count*]**b**	Cursor backward one word.
[*count*]**B**	Cursor to preceding blank separated word.
[*count*]**|**	Cursor to column *count*.
[*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**.
[*count*]**;**	Repeats *count* times, the last single character find command, **f**, **F**, **t**, or **T**.
[*count*]**,**	Reverses the last single character find command *count* times.
0	Cursor to start of line.
^	Cursor to first non-blank character in line.
$	Cursor to end of line.

Search Edit Commands

These commands access your command history.

[*count*]**k**	Fetch previous command. Each time **k** is entered the previous command back in time is accessed.
[*count*]**–**	Equivalent to **k**.
[*count*]**j**	Fetch next command. Each time **j** is entered the next command forward 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*. *String* is terminated by a "RETURN" or "NEW LINE". If string is preceded by a **^**, the matched line

must begin with *string*. If *string* is null the previous string will be used.

?*string* Same as / except that search will be in the forward direction.

n Search for next match of the last pattern to / or **?** commands.

N Search for next match of the last pattern to / or **?**, but in reverse direction. Search history for the *string* entered 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 current character through the character that *motion* would move the cursor to and enter input mode. If *motion* is **c**, the entire line will be deleted and input mode entered.

C Delete the current character through the end of line and enter input mode. Equivalent to **c$**.

S Equivalent to **cc**.

D Delete the current character through the end of line. Equivalent to **d$**.

[*count*]**d***motion*
d[*count*]*motion*
Delete 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 **0i**.

[*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 characters on the screen with characters you type overlay fashion.

[*count*]**r***c* Replace the *count* character(s) starting at the current cursor position with *c*, and advance the cursor.

[*count*]**x** Delete current character.

[*count*]**X** Delete preceding character.

[*count*]**.** Repeat the previous text modification command.

[*count*]**~** Invert the case of the *count* character(s) starting at the current cursor position 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 file name generation attempted. If no match is found, it rings the bell. Otherwise, the word is replaced by the matching pattern and input mode is entered.

**** Filename completion. Replaces the current word with the longest common prefix of all filenames matching the current word with an asterisk appended. If the match is unique, a / is appended if the file is a directory and a space is appended if the file is not a directory.

Other Edit Commands

Miscellaneous commands.

[*count*]y*motion*

y[*count*]*motion*
: Yank current character through character that *motion* would move the cursor to and puts them into the delete buffer. The text and cursor are unchanged.

Y Yanks from current position to end of 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 control mode.

^J (New line) Execute the current line, regardless of mode.

^M (Return) Execute the current line, regardless of mode.

Sends the line after inserting a # in front of the line. Useful for causing the current line to be inserted in the history without being executed.

= List the file names that match the current word if an asterisk were appended it.

@*letter* Your alias list is searched 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 and the exit status, when there is no syntax error, is zero. Commands that are preceded by one or two † are treated specially in the following ways:

1. Parameter assignment lists preceding the command remain in effect when the command completes.
2. I/O redirections are processed after parameter assignments.
3. Errors cause a script that contains them to abort.
4. Words, following a command preceded by †† that are in the format of a parameter assignment, are expanded with the same rules as a parameter assignment. This means that tilde substitution is performed after the = sign and word splitting and file name generation are not performed.

† **:** [*arg* ...]
: The command only expands parameters.

† **.** *file* [*arg* ...]
: Read the complete *file* then execute the commands. 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. The exit status is the exit status of the last command executed.

†† **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 remained tracked. Without the **–t** flag, for 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 for scripts invoked by name. The exit status is non-zero if a *name* is given, but no value, for which no alias has been defined.

bg [*job*...]
This command is only on systems that support job control. Puts each specified *job* into the background. The current job is put in the background if *job* is not specified. See *Jobs* for a description of the format of *job*.

† **break** [*n*]
Exit from the enclosing **for while until** or **select** loop, if any. If *n* is specified then break *n* levels.

† **continue** [*n*]
Resume 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 path name, 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 the current directory name, **PWD** and tries to change to this new directory.

The **cd** command may not be executed by `rksh`.

echo [*arg* ...]
See *echo*(1) for usage and description.

† **eval** [*arg* ...]
The arguments are read as input to the shell and the resulting command(s) 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 numbers greater than 2 that 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 except for a shell which has the *ignoreeof* option (See **set** below) turned on.

†† **export** [*name*[=*value*]] ...
The given *names* 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 starting 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 is the previous command for editing and –16 for listing. The flag **–r** reverses the order of the commands and the flag **–n** suppresses command numbers when listing. In the second form the *command* is re-executed after the substitution *old=new* is performed.

fg [*job*...]
This command is only on systems that support job control. Each *job* specified is brought to the foreground. Otherwise, the current job is brought into the foreground. See *Jobs* for a description of the format of *job*.

getopts *optstring name* [*arg* ...]
Checks *arg* for legal options. If *arg* is omitted, the positional parameters are used. An option argument begins with a + or a –. An option not beginning with + or – or the argument –– ends the options. *optstring* contains the letters that *getopts* recognizes. If a letter is followed by a **:**, that option is expected to have an argument. The options can be separated from the argument by blanks.

getopts places the next option letter it finds inside variable *name* each time it is invoked with a + prepended when *arg* begins with a +. The index of the next *arg* is stored in **OPTIND**. The option argument, if any, gets stored in **OPTARG**.

A leading **:** in *optstring* causes **getopts** to store the letter of an invalid option in **OPTARG**, and to set *name* to **?** for an unknown option and to **:** when a required option is missing. Otherwise, **getopts** prints an error message. The exit status is non-zero when there are no more options.

jobs [**–lnp**] [*job* ...]

Lists information about each given job; or all active jobs if *job* is omitted. The **–l** flag lists process ids in addition to the normal information. The **–n** flag only displays jobs that have stopped or exited since last notified. The **–p** flag causes only the process group to be listed. See *Jobs* for a description of the format of *job*.

kill [*–sig*] *job* ...
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"). 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 *job* can the process id of a process that is not a member of one of the active jobs. See *Jobs* for a description of the format of *job*. In the second form, **kill –l**, the signal numbers and names are listed.

let *arg* ...
Each *arg* is a separate *arithmetic expression* to be evaluated. See *Arithmetic Evaluation* above, for a description of arithmetic expression evaluation.

The exit status is 0 if the value of the last expression is non-zero, and 1 otherwise.

† **newgrp** [*arg* ...]
Equivalent to **exec /bin/newgrp** *arg* ....

print [**–Rnprsu**[*n*]] [*arg* ...]
The shell output mechanism. With no flags or with flag – or –– 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 standard output. The **–u** flag can be used to specify a one digit file descriptor unit number **n** on which the output will be placed. The default is 1. If the flag **–n** is used, no **new-line** 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 fields 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 field is assigned to the first *name*, the second field to the second *name*, etc., with leftover fields 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 exit status is 0 unless an end-of-file is encountered. An end-of-file with the **–p** option causes cleanup for this process so that another can be spawned. If the first argument contains a **?**, the remainder of this word is used as a *prompt* on standard error when the shell is interactive. The exit status is 0 unless an end-of-file is encountered.

†† **readonly** [*name*[=*value*]] ...
The given *names* are marked readonly 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 [**±aefhkmnopstuvx**] [**±o** *option*]... [**±A** *name*] [*arg* ...]
The flags for this command have meaning as follows:

–A Array assignment. Unset the variable *name* and assign values sequentially from the list *arg*. If **+A** is used, the variable *name* is not unset first.

–a All subsequent parameters that are defined are automatically exported.

–e If a command has a non-zero exit status, execute the **ERR** trap, if set, and exit. This mode is disabled while reading profiles.

–f Disables file name generation.

–h Each command 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 Read commands and check them for syntax errors, but do not execute them. Ignored for interactive shells.

–o The following argument can be one of the following option names:

allexport
Same as **–a**.

errexit Same as **–e**.

bgnice All background jobs are run at a lower priority. This is the default mode.

emacs Puts you in an *emacs* style in-line editor for command entry.

gmacs Puts you in a *gmacs* style in-line 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 file name generation have a trailing / appended.

monitor Same as **–m**.

noclobber
Prevents redirection > from truncating existing files. Require >| to truncate a file when turned on.

noexec Same as **–n**.

noglob Same as **–f**.
nolog Do not save function definitions in history file.
nounset Same as **–u**.
privileged
Same as **–p**.
verbose Same as **–v**.
trackall Same as **–h**.
vi Puts you in insert mode of a *vi* style in-line editor until you hit escape character **033**. This puts you in 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 option name is supplied then the current option settings are printed.

–p Disables processing of the **$HOME/.profile** file and uses the file **/etc/suid_profile** instead of the ENV file. This mode is on whenever the effective uid (gid) is not equal to the real uid (gid). Turning this off causes the effective uid and gid to be set to the real uid and gid.
–s Sort the positional parameters lexicographically.
–t Exit after reading and executing one command.
–u Treat unset parameters as an error when substituting.
–v Print shell input lines as they are read.
–x Print commands and their arguments as they are executed.
– Turns off **–x** and **–v** flags and stops examining arguments for flags.
– – Do 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 **$–**. Unless **–A** is specified, the remaining arguments are positional parameters and are assigned, in order, to **$1 $2** .... If no arguments are given then the names and values of all named parameters are printed on the standard output. If the only argument is +, the names of all named parameters are printed.

† **shift** [*n*]
The positional parameters from **$***n***+1** ... are renamed **$1** ... , default *n* is 1. The parameter *n* can be any arithmetic expression that evaluates to a non-negative number less than or equal to **$#**.

† **times**
Print the accumulated user and system times for the shell and for processes run from the shell.

† **trap** [*arg*] [*sig*] ...
arg is a command 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 non-zero exit status. *sig* is **DEBUG** then *arg* will be executed after each command. 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** [±**HLRZfilrtux**[*n*]] [*name*[=*value*]] ...
Sets attributes and values for shell parameters. 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 UNIX system to host-name file mapping on non-UNIX system machines.

–L Left justify and remove leading blanks from *value*. If *n* is non-zero it defines the width of the field, otherwise it 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 Right justify and fill with leading blanks. If *n* is non-zero it defines the width of the field, otherwise it is determined by the width of the value of first assignment. The field is left filled with blanks or truncated from the end if the parameter is reassigned. The **L** flag is turned off.

–Z Right justify and fill with leading zeros if the first non-blank character is a digit and the **–L** flag has not been set. If *n* is non-zero it defines the width of the field, otherwise it 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**, **–u** and **–x**. The flag **–t** turns on execution tracing for this function. The flag **–u** causes this function to be marked undefined. The **FPATH** variable will be searched to find the function definition when the function is referenced. The flag **–x** allows the function definition to remain in effect across shell procedures invoked by name.

–i Parameter is an integer. This makes arithmetic faster. If *n* is non-zero it defines the output arithmetic base, otherwise the first assignment determines the output base.

–l All upper-case characters converted to lower-case. The upper-case flag, **–u** is turned off.

–r The given *names* are marked readonly 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 lower-case characters are converted to upper-case characters. The lower-case 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 *parameters* which have these flags set is printed. (Using + rather than – keeps the values from being printed.) If no *names* and flags are given, the *names* and *attributes* of all *parameters* are printed.

ulimit [**–HSacdfmnpstvw**] [*limit*]

Set or display a resource limit. The available resources limits listed below. Many systems to not contain one or more of these limits. The limit for a specified resource is set when *limit* is specified. The value of *limit* can be a number in the unit specified below with each resource, or the value **unlimited**. The **H** and **S** flags specify whether the hard limit or the soft limit for the given resource is set. A hard limit cannot be increased once it is set. A soft limit can be increased up to the value of the hard limit. If neither the **H** or **S** options is specified, the limit applies to both. The current resource limit is printed when *limit* is omitted. In this case the soft limit is printed unless **H** is specified. When more that one resource is specified, then the limit name and unit is printed before the value.

- **–a** Lists all of the current resource limits.
- **–c** The number of 512-byte blocks on the size of core dumps.
- **–d** The number of K-bytes on the size of the data area.
- **–f** The number of 512-byte blocks on files written by child processes (files of any size may be read).
- **–m** The number of K-bytes on the size of physical memory.
- **–n** The number of file descriptors.
- **–p** The number of 512-byte blocks for pipe buffering.
- **–s** The number of K-bytes on the size of the stack area.
- **–t** The number of seconds to be used by each process.
- **–v** The number of K-bytes for virtual memory.
- **–w** The number of K-bytes for the swap area.

If no option is given, **–f** is assumed.

umask [*mask*]

The user file-creation mask is set to *mask* (see *umask*(2)). *mask* can either be an octal number or a symbolic value as described in *chmod*(1). If a symbolic value is given, the new umask value is the complement of the result of applying *mask* to the complement of the previous umask value. If *mask* 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, i. e., their values and attributes are erased. Readonly variables cannot be unset. If the flag, **–f**, is set, then the names refer to *function* names. Unsetting **ERRNO, LINENO, MAILCHECK, OPTARG, OPTIND, RANDOM, SECONDS, TMOUT**, and _ causes removes their special meaning even if they are subsequently assigned to.

† **wait** [*job*]

Wait for the specified *job* and report its termination status. If *job* is not given then all currently active child processes are waited for. The exit

status from this command is that of the process waited for. See *Jobs* for a description of the format of *job*.

whence [**–pv**] *name* ...
For each *name*, indicate how it would be interpreted if used as a command name.

The flag, **–v**, produces a more verbose report.

The flag, **–p**, does a path search for *name* even if name is an alias, a function, or a reserved word.

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 *getgid* settings will be ignored. Commands are then read as described below; 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 above, 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.

The remaining flags and arguments are described under the **set** command above. The `rksh` command is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. The actions of `rksh` are identical to those of *sh*, except that the following are disallowed:

changing directory (see *cd*(1)),
setting the value of **SHELL**, **ENV**, or **PATH**,
specifying path or command names containing /,
redirecting output (**>**, **>|** , **<>** , and **>>**).

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, `rksh` 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** has complete control over user actions, by performing guaranteed setup actions and leaving the user in an appropriate directory (probably *not* the login directory).

The system administrator often sets up a directory of commands (that is, **/usr/rbin**) that can be safely invoked by `rksh`. Some systems also provide a restricted editor *red*.

Exit Status

Errors detected by the shell, such as syntax errors, cause the shell to return a non-zero exit status. Otherwise, the shell returns the exit status of the last command executed (see also the **exit** command above). If the shell is being used non-interactively then execution of the shell file is abandoned. Run time 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

/etc/passwd
/etc/profile
/etc/suid_profile
$HOME/.profile
/tmp/sh*
/dev/null

See Also

cat(1), cd(1), chmod(1), cut(1), echo(1), emacs(1), env(1), gmacs(1), newgrp(1), stty(1), test(1), umask(1), vi(1), dup(2), exec(2), fork(2), ioctl(2), lseek(2), paste(1), pipe(2), signal(2), umask(2), ulimit(2), wait(2), rand(3), a.out(5), profile(5), environ(7).

Morris I. Bolsky and David G. Korn, *The KornShell Command and Programming Language*, Prentice Hall, 1989.

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 before 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. |.

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.

Traps are not processed while a job is waiting for a foreground process. Thus, a trap on **CHLD** won't be executed until the foreground job terminates.

Name

last – indicate last logins of users and teletypes

Syntax

last [–N] [*name...*] [*tty...*]

Description

The `last` command looks back in the `wtmp` file, which records all logins and logouts, for information about a user, a teletype or any group of users and teletypes. Arguments specify names of users or teletypes of interest. Names of teletypes can be given fully or abbreviated. For example 'last 0' is the same as 'last tty0'.

If multiple arguments are given, the information that applies to any of the arguments is printed. For example, 'last root console' lists all of "root's" sessions as well as all sessions on the console terminal.

The `last` command prints the sessions of the specified users and teletypes, most recent first, indicating the times at which the session began, the duration of the session, and the teletype on which the session took place. If the session is ongoing or was cut short by a reboot, `last` indicates that this is so.

The pseudo-user *reboot* logs in at reboots of the system. Therefore, the following example gives an indication of mean time between reboot:

```
last reboot
```

The `last` command with no arguments prints a record of all logins and logouts, in reverse order.

If `last` is interrupted, it indicates how far the search has progressed in `wtmp`. If interrupted with a quit signal (generated by a <CTRL/E>) `last` indicates how far the search has progressed so far, and the search continues.

Options

–N   Limits the number of output lines to the specified number.

Files

/usr/adm/wtmp
  login data base

/usr/adm/shutdownlog
  records that shutdowns occurred and why

See Also

wtmp(5), ac(8), lastcomm(1)

Name

lastcomm – show last commands executed in reverse order

Syntax

lastcomm [*command name...*] [*user name...*] [*terminal name...*]

Description

The `lastcomm` command gives information on previously executed commands. With no arguments, `lastcomm` prints information about all the commands recorded during the current accounting file's lifetime. If called with arguments, only accounting entries with a matching command name, user name, or terminal name are printed. The following example produces a listing of all the executions of commands named `a.out` by user *root* on the terminal *ttyd0*:

```
lastcomm a.out root ttyd0
```

For each process entry, the following are printed:

The name of the user who ran the process.

Flags, as accumulated by the accounting facilities in the system.

The command name under which the process was called.

The amount of cpu time used by the process (in seconds).

The time the process exited.

The flags are encoded as follows:

''S'' indicates the command was executed by the super-user

''F'' indicates the command ran after a fork, but without a following *exec*

''C'' indicates the command was run in PDP-11 compatibility mode (VAX only)

''D'' indicates the command terminated with the generation of a *core* file

''X'' indicates the command was terminated with the signal SIGTERM

See Also

last(1), sigvec(2), acct(5), core(5)

lb_admin(1ncs)

Name

lb_admin – Location Broker Administrative Tool

Syntax

/etc/ncs/lb_admin [**–version**] [**–nq**]

Description

The `lb_admin` tool monitors and administers the registrations of DECrpc-based servers in Global Local Broker (GLB) or Local Location Broker (LLB) databases. A server registers Universal Unique Identifiers (UUIDs) specifying an object, a type, and an interface, along with a socket address specifying its location. A client can locate servers by issuing lookup requests to GLBs and LLBs.

In accepting input or displaying output, `lb_admin` uses either character strings or descriptive textual names to identify objects, types, and interfaces. A character string directly represents the data in a UUID in the following format:

nnnnnnnnnnnn.nn.nn.nn.nn.nn.nn.nn.nn

where each *n* is a hexadecimal digit.

With `lb_admin`, you examine or modify only one database at a time, referred to as the current database. The `use_broker` command selects the type of Location Broker database, GLB or LLB. The `set_broker` command selects the host whose LLB database is to be accessed.

Information about individual command interfaces is available through the `help` command.

Options

–nq Do not query for verification of wildcard expansions in `unregister` operations.

–version Display the version of the Network Computing Kernel (NCK) that this `lb_admin` belongs to, but do not start the tool. (NCK is part of the Network Computing System (NCS) on which DECrpc is based.)

Commands

In the descriptions of `lookup`, `register`, and `unregister`, the *object*, *type*, and *interface* arguments can be either character strings representing UUIDs or textual names corresponding to UUIDs, as described earlier.

In the descriptions of `register` and `unregister`, the *location* argument is a string in the format *family:host[port]*, where *family* is an address family, *host* is a host name, and *port* is a port number. The only value for *family* is **ip**. You can use a leading number sign (#) to indicate that a host name is in the standard numeric form. For example, **ip:vienna[1756]**, and **ip:#192.5.5.5[1791]** are both acceptable *location* specifiers.

`a[dd]` Synonym for `register`.

`c[lean]` Find and delete obsolete entries in the current database.

When you issue the clean command, lb_admin attempts to contact each server registered in the database. If the server does not respond, lb_admin tries to look up its registration in the LLB database at the host where the server is located, tells you the result of this lookup, and asks whether you want to delete the entry. If a server responds, but its UUIDs do not match the entry in the database, lb_admin tells you this result and asks whether you want to delete the entry, even if you used the **-nq** option to lb_admin.

There are two situations in which it is likely that a database entry should be deleted:

- The server does not respond, lb_admin succeeds in contacting the LLB at the host where the server is located, and the server is not registered with that LLB. The server is probably no longer running.
- A server responds, but its UUIDs do not match the entry in the database. The server that responded is not the one that registered the entry.

Entries that meet either of these conditions are probably safe to delete and are considered eligible for automatic deletion (described in the next paragraph). In other situations, it is best not to delete the entry unless you can verify directly that the server is not running (for example, by listing the processes running on its host).

When the clean command asks whether you want to delete an entry, choose one of the following responses:

y[es] Delete the entry.

n[o] Leave the entry intact in the current database.

g[o] Invoke automatic deletion, in which all eligible entries (see the previous paragraph) are deleted and all ineligible entries are left intact, without your being queried, until all entries have been checked.

q[uit] Terminate the clean operation.

d[elete] Synonym for unregister.

h[elp] [*command*] or ? [*command*]

Display a description of the specified *command* or, if none is specified, list all of the lb_admin commands.

l[ookup] *object type interface*

Look up and display all entries with matching *object, type,* and *interface* fields in the current database. Use the letter **l** to list all of the entries in the database. You can use asterisks as wildcards for any of the arguments. If all the arguments are wildcards, or if no arguments are given, lookup displays the entire database.

q[uit] Exit the lb_admin session.

r[egister] *object type interface location annotation [flag]*

Add the specified entry to the current database. You can use an

asterisk to represent the nil UUID in the *object, type*, and *interface* fields.

The *annotation* is a string of up to 64 characters annotating the entry. Use double quotation marks (" ") to delimit a string that contains a space or contains no characters. To embed a double quotation mark in the string, precede it with a backslash (\).

The *flag* is either local (the default) or global, indicating whether the entry should be marked for local registration only or for registration in both the LLB and the GLB databases. The *flag* is a field that is stored with the entry; it does not affect where the entry is registered. The `set_broker` and `use_broker` commands select the particular LLB or GLB database for registration.

`set_broker` [*broker_switch*] *location*
: Set the host for the current LLB or GLB. If you specify global as the *broker_switch*, `set_broker` sets the current GLB; otherwise, it sets the current LLB. The *host* is a *location* specifier as described earlier, but the *[port]* portion is ignored and can be omitted.

  Issue the `use_broker` command, not the `set_broker` command, to determine whether subsequent operations will access the LLB or the GLB.

`set_t[imeout]` *[short / long]*
: Set the timeout period used by `lb_admin` for all of its operations. With an argument of *short* or *long*, `set_timeout` sets the timeout accordingly. With no argument, it displays the current timeout value.

`u[nregister]` *object type interface location*
: Delete the specified entry from the current database.

  You can use an asterisk as a wildcard in the *object, type,* and *interface* fields to match any value for the field. Unless you suppress queries by specifying the **-nq** option of `lb_admin`, `unregister` asks you whether to delete each matching entry. Choose one of the following responses:

  y[es] Delete the entry.

  n[o] Leave the entry in the database.

  g[o] Delete all remaining database entries that match, without your being queried.

  q[uit] Terminate the `unregister` operation, without deleting any more entries.

`us[e_broker]` *[broker_switch]*
: Select the type of database that subsequent operations will access, GLB or LLB. The *broker_switch* is either global or local. If you do not supply a *broker_switch*, `use_broker` tells whether the current database is global or local.

  Use `set_broker` to select the host whose GLB or LLB is to be accessed.

See Also

llbd(8ncs), nrglbd(8ncs)
Guide to the Location Broker

Name

ld, uld – RISC link editor and ucode link editor

Syntax

ld [*options*] [*file file file* ...
uld *options*] [*file file file* ...

Description

The `ld` command is the RISC link editor. It links RISC extended *coff* object files. The archive format understood by `ld` is the one created by the archiver `ar`(1).

The `ld` command 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 *files* are given. The `ld` command combines them, producing an object module that can be executed or used as input for a subsequent `ld` run. (In the latter case, the **–r** option must be given to preserve the relocation entries.) The output of `ld` is left in **a.out**. By default, this file is executable if no errors occurred during the load.

The argument object files are concatenated in the order specified. The entry point of the output is the beginning of the text segment (unless the **–e** option is specified).

The `uld` command combines several ucode object files and libraries into one ucode object file. It hides external symbols for better optimizations by subsequent compiler passes. The symbol tables of *coff* object files loaded with ucode object files are used to determine what external symbols not to hide along with files specified by the user that contain lists of symbol names.

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`(1)) is a hash table and is searched to resolved external references that can be satisfied by library members. The ordering of library members is unimportant.

When searching for libraries the default directories to search in are /lib/, /usr/lib/ and /usr/local/lib/ . If the target byte ordering of the object files being loaded is of the opposite byte ordering of the machine the link editor is running on then the default search directories for libraries are changed. The change is to replace the last name of the directories from "lib/" to "libeb/" or "libel/" to match the target byte ordering of the objects being loaded.

The following symbols are reserved and should not be defined: etext, edata, end, _ftext, _fdata, _fbss, _gp, _procedure_table, _procedure_table_size, and _procedure_string_table. These loader defined symbols, if referred to, are set their values as described in `end`(3).

Options

The following options are recognized by both `ld` and `uld`. Those options used by one and not the other are ignored. Any option can be preceded by a '**k**' (for example **–ko** *outfile*) and except for **–kl***x* have the same meaning with or without the preceding '**k**'. This is done so that these options can be passed to both link editors through compiler drivers.

–o *outfile* — Produce an output object file by the name *outfile*. The name of the default object file is **a.out**.

–l*x* — Search a library **lib***x***.a,** where *x* is a string. A library is searched when its name is encountered, so the placement of a **–l** is significant.

–kl*x* — Search a library **lib***x***.b,** where *x* is a string. These libraries are intended to be ucode object libraries. In all other ways, this option is like the **–l***x* option.

–L*dir* — Change the algorithm of searching for **lib***x***.a** or **lib***x***.b** to look in *dir* before looking in the default directories. This option is effective only if it precedes the **–l** options on the command line.

–L — Change the algorithm of searching for **lib***x***.a** or **lib***x***.b** to **never** look in the default directories. This is useful when the default directories for libraries should not be searched and only the directories specified by **–L***dir* are to be searched.

–K*dir* — Change the default directories to the single directory *dir*. This option is only intended to be used by the compiler driver. Users should use the **–L** and **–L***dir* options to get the effect they desire.

–B*string* — Append *string* to the library names created for the **–l***x* and **–kl***x* when searching for library names. For each directory to be searched the name is first created with the *string* and if it is not found it is created without the *string*.

–p *file* — Preserve (do not hide) the symbol names listed in *file* when loading ucode object files. The symbol names in the file are separated by blanks, tabs, or newlines.

–s — Strip the symbolic information from the output object file.

–x — Do not preserve local (non-.globl) symbols in the output symbol table; enter external and static symbols only. This option saves some space in the output file.

–r — Retain relocation entries in the output file. Relocation entries must be saved if the output file is to become an input file in a subsequent `ld` run. This option also prevents final definitions from being given to common symbols, and suppresses the undefined symbol diagnostics.

–d — Force definition of common storage and define loader defined symbols even if **–r** is present.

–u *symname* — Enter *symname* as an undefined in the symbol table. This is useful for loading entirely from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

–F or **–z** — Arrange for the process to be loaded on demand from the resulting executable file (413 format) rather than preloaded, a ZMAGIC file. This is the default.

–n — Arrange (by giving the output file a 0410 "magic number") that when the output file is executed, the text portion is read-only and

shared among all users executing the file, an NMAGIC file. This involves moving the data areas up to the first possible *pagesize* byte boundary following the end of the text.

–N
: Place the data section immediately after the text and do not make the text portion read only or sharable, an OMAGIC file. (Use "magic number" 0407.)

–T *num*
: Set the text segment origin. The argument *num* is a hexadecimal number. See the notes section for restrictions.

–D *num*
: Set the data segment origin. The argument *num* is a hexadecimal number. See the notes section for restrictions.

–B *num*
: Set the bss segment origin. The argument *num* is a hexadecimal number. This option can be used only if the final object is an OMAGIC file.

–e *epsym*
: Set the default entry point address for the output file to be that of the symbol *epsym*.

–m
: Produce a map or listing of the input/output sections on the standard output (UNIX system V-like map).

–M
: Produce a primitive load map, listing the names of the files that are loaded (UNIX 4.3bsd-like map).

–S
: Set silent mode and suppress non-fatal errors.

–v
: Set verbose mode. Print the name of each file as it is processed.

–y*sym*
: Indicate each file in which *sym* appears, *sym*'s type and whether the file defines or references *sym*. Many such options may be given to trace many symbols.

–V
: Print a message giving information about the version of `ld` being used.

–VS *num*
: Use **num** as the decimal version stamp to identify the a.out file that is produced. The version stamp is stored in the optional and symbolic headers.

–f *fill*
: Set the fill pattern for ''holes'' within an output section. The argument *fill* is a four-byte hexadecimal constant.

–G *num*
: The argument *num* is taken to be a decimal number that is the largest size in bytes of a *.comm* item or literal that is to be allocated in the small bss section for reference off the global pointer. The default is 8 bytes.

–bestGnum
: Calculate the best **–G** num to use when compiling and linking the files which produced the objects being linked. Using too large a number with the **–G** *num* option may cause the gp (global-pointer) data area to overflow; using too small a number may reduce your program's execution speed.

–count, –nocount, –countall
: These options control which objects are counted as recompilable for the best **–G** *num* calculation. By default, the **–bestGnum** option assumes you can recompile everything with a different **–G**

num option. If you cannot recompile certain object files or libraries (because, for example, you have no sources for them), use these options to tell the link editor to take this into account in calculating the best **–G** *num* value. **–nocount** says that object files appearing after it on the command line cannot be recompiled; **–count** says that object files appearing after it on the command line can be recompiled; you can alternate the use of **–nocount** and **–count. –countall** overrides any **–nocount** options appearing after it on the command line.

–b Do not merge the symbolic information entries for the same file into one entry for that file. This is only needed when the symbolic information from the same file appears differently in any of the objects to be linked. This can occur when object files are compiled, by means of conditional compilation, with an apparently different version of an include file.

–jmpopt and **–nojmpopt**
Fill or do not fill the delay slots of jump instructions with the target of the jump and adjust the jump offset to jump past that instruction. This always is disabled for debugging (when the **–g1, –g2** or **–g** flag is present). When this option is enabled it requires that all of the loaded program's text be in memory and could cause the loader to run out of memory. The default is **–nojmpopt.**

–g or **–g[0123]** These options are accepted and except for **–g1, –g2** or **–g** disabling the **–jmpopt** have no other effect.

–A *file* Specifies incremental loading. For example, linking is to be done in a manner so that the resulting object may be read into an already executing program. The next argument, *file,* is the name of a file whose symbol table is taken as a basis on which to define additional symbols. Only newly linked material is entered into the text and data portions of **a.out,** but the new symbol table reflects every symbol defined before and after the incremental load. This argument must appear before any other object file in the argument list. The **–T** option may be used as well, and is taken to mean that the newly linked segment commences at the corresponding address (which must be a correct multiple for the resulting object type). The default resulting object type is an OMAGIC file and the default starting address of the text is the old value of end rounded to SCNROUND as defined in the include file *<scnhdr.h>*. Using the defaults, when this file is read into an already executing program the intial value of the break must also be rounded. All other objects except the argument to the **–A** option must be compiled **–G** *0* and this sets **–G** *0* for linking.

The following option is not intended for general use.

–i *file* The .text section of *file* is moved into the .init section of the resulting object file.

The link editors `ld` and `uld` accept object files targeted for either byte ordering with their headers and symbolic tables in any byte ordering; however `ld` and `uld` are faster if the headers and symbolic tables have the byte ordering of the machine that they are running on. The default byte ordering of the headers and symbolic tables is

the target byte ordering of the output object file. For non-relocatable object files the default byte ordering of the headers and symbolic tables cannot be changed.

–EB Produce the output object file with big-endian byte ordered headers and symbolic information tables.

–EL Produce the output object file with little-endian byte ordered headers and symbolic information tables.

Restrictions

The segments must not overlap and all addresses must be less than 0x80000000. The stack starts below 0x80000000 and grows through lower addresses so space should be left for it. For ZMAGIC and NMAGIC files the default text segment address is 0x00400000 and the default data segment address is 0x10000000. For OMAGIC files the default text segment address is 0x10000000 with the data segment following the text segment. The default for all types of files is that the bss segment follows the data segment.

For OMAGIC files to be run under the operating system the -B flag should not be used because the bss segment must follow the data segment which is the default.

The segments must be on 4 megabyte boundaries. Objects linked at addresses other than the default will not run.

Files

/lib/lib*.a
/usr/lib/lib*.a
/usr/local/lib/lib*.a libraries
a.out output file

See Also

cc(1), pc(1), f77(1), as(1), ar(1), end(3)

Name

ld – link editor

Syntax

ld [*option...*] *file...*

Description

The `ld` command combines several object programs into one, resolves external references, and searches libraries. In the simplest case, several object *files* are given, and `ld` combines them, producing an object module which can either be executed or can become the input for a further `ld` run. (In the latter case, the **–r** option must be given to preserve the relocation bits.) The output of `ld` is left on a.out. This file is only made executable if no errors occurred during the load.

The argument routines are linked together in the order specified. The entry point of the output is the beginning of the first routine, unless the **–e** option is specified.

If the argument is a library, it is searched only once at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. If a routine from a library references another routine in the library, and the library has not been processed by `ranlib`(1), the referenced routine must appear after the referencing routine in the library. Thus, the order of programs within libraries is important. The first member of a library should be a file named __.SYMDEF, which is a dictionary for the library that is produced by `ranlib`(1). The dictionary is searched repeatedly to satisfy as many references as possible.

The symbols _etext, _edata and _end (etext, edata and end in C) are reserved and, if referred to, are set to the first location above the program, the first location above initialized data, and the first location above all data, in that order. It is an error to define these symbols.

Options

The `ld` command has several options. Except for the **–l** option, they should appear before the file names.

–A Specifies incremental loading. Linking is done so that the resulting object may be read into an already executing program. The next argument is the name of a file whose symbol table is used to define additional symbols. Only newly linked material is entered into the text and data portions of a.out, but the new symbol table reflects every symbol defined before and after the incremental load. This argument must appear before any other object file in the argument list.

The **–T** option may be used as well, and is taken to mean that the newly linked segment commences at the corresponding address (which must be a multiple of 1024). The default value is the old value of _end.

–D Takes the next argument as a hexadecimal number and pads the data segment with zero bytes to the indicated length.

–d Forces definition of common storage even if the **–r** flag is present.

–e Takes the next argument as the name of the entry point of the loaded

program; location 0 is the default.

–L*dir* Adds *dir* to the list of directories that are searched for libraries. Directories specified with **–L** are searched before the standard directories.

–l*x* Abbreviates the library name lib*x*.a, where *x* is a string. The `ld` command searches for libraries first in any directories specified with **–L** options, then in the standard directories /lib, /usr/lib, and /usr/local/lib. A library is searched when its name is encountered, so the placement of a **–l** is significant.

–H Takes the next argument as a decimal integer, adds it to the end of text, and causes the data section to start at a higher address.

–M Produces a primitive load map, listing the names of the files that are loaded.

–N Indicates a portion of text to not make read-only or sharable. (Use magic number 0407.)

–n Arranges (by giving the output file a 0410 magic number) that the text portion is read-only and shared among all users executing the file when the output file is executed. This involves moving the data areas up to the first possible 1024 byte boundary following the end of the text.

–o Takes the *name* argument after **–o** as the name of the `ld` output file, instead of a.out.

–r Generates relocation bits in the output file so that it can be the subject of another `ld` run. This flag also prevents final definitions from being given to common symbols and suppresses the undefined symbol diagnostics.

–S Strips the output by removing all symbols except locals and globals.

–s Removes the symbol table and relocation bits to save space (this impairs the usefulness of the debuggers). This information can also be removed by `strip`(1).

–T Takes the next argument as a hexadecimal number which sets the text segment origin. The default origin is 0.

–t(trace) Prints the name of each file as it is processed.

–u Takes the next argument as a symbol and enters it as undefined in the symbol table. This is useful for loading from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

–X Saves local symbols except for those whose names begin with a capital L. This option is used by `cc`(1) to discard internally-generated labels while retaining symbols local to routines.

–x Discards local (non-global) symbols in the output symbol table; only enters external symbols. This option saves some space in the output file.

–Y*environment*
Adjusts the magic number in the output file so that the program runs in the specified *environment* . The parameter can be **POSIX, SYSTEM_FIVE, or BSD** . The parameter sets the program's execution

environment to conform with one of the three standards. If it is present, this parameter overrides the PROG_ENV environment variable. If no *environment* is given, **SYSTEM_FIVE** is assumed. If neither this parameter nor the PROG_ENV variable is present, **–YBSD** is assumed.

–y*sym* Indicates each file in which *sym* appears, its type, and whether the file defines or references it. Many such options may be given to trace many symbols. It is usually necessary to begin *sym* with an underscore (_), because external C, FORTRAN and Pascal variables begin with underscores.

–z Arranges for the process to be loaded on demand from the resulting executable file (413 format) rather than preloaded. This is the default. It results in a 1024 byte header on the output file followed by a text and data segment whose size is a multiple of 1024 bytes (being padded out with nulls in the file if necessary). With this format the first few BSS segment symbols may, from the output of `size`(1), appear to reside in the data segment. This avoids wasting the space which results from the roundup of the data segment size.

Restrictions

There is no way to force data to be page aligned. The `ld` command pads the images which are to be demand loaded from the file system to the next page boundary.

When linking code contains GFLOAT instructions, the GFLOAT versions of *libc* and/or the math library must be used rather than the normal DFLOAT versions. Link to these by using *–lcg* and/or *–lmg*.

The compiler and the linker `ld`(1) cannot detect the use of mixed double floating point types, and your program may produce erroneous results.

Files

/lib/lib*.a	libraries.
/usr/lib/lib*.a	libraries
/usr/local/lib/lib*.a	libraries
a.out	output file

See Also

ar(1), as(1), cc(1), ranlib(1)

leave(1)

Name

leave – remind you when you have to leave

Syntax

leave [hhmm]

Description

The `leave` command 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). 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. To get rid of it you should either log off or use "kill –9" giving its process ID.

See Also

calendar(1)

Name

lex – generate lexical analyzer

Syntax

lex [–tvfn] [*file...*]

Description

The `lex` command generates programs to be used in simple lexical analysis of text. The input *files* (standard input default) contain regular expressions to be searched for, and actions written in C to be executed when expressions are found.

A C source program, 'lex.yy.c', is generated. It is compiled using the following command line:

```
cc lex.yy.c -ll
```

This program copies unrecognized portions of the input to the output, and executes the associated C action for each regular expression that is recognized.

Options

–f Runs a faster compilation (does not pack resulting tables). This is limited to small programs.

–n Prints no summary information (default option).

–t Writes to standard output instead of to file `lex.yy.c`.

–v Prints one-line summary of generated statistics.

Examples

In the following example, the command

```
lex lexcommands
```

draws `lex` instructions from the file *lexcommands*, and places the output in lex.yy.c. The command

```
%%
[A-Z]  putchar(yytext[0]+'a'-'A');
[ ]+$
[ ]+   putchar(' ');
```

is an example of a `lex` program that would be put into a `lex` command file. This program converts upper case to lower, removes blanks at the end of lines, and replaces multiple blanks by single blanks.

See Also

sed(1), yacc(1)
"LEX – Lexical Analyzer Generator," *ULTRIX Supplementary Documents,* Vol. II:Programmer

line(1)

Name

line – read one line

Syntax

line

Description

The `line` command copies one line (up to a new-line) from the standard input and writes it on the standard output. It returns an exit code of 1 on EOF (end-of-file) and always prints at least a new-line. It is often used within shell files to read from the user's terminal.

See Also

sh(1), read(2)

Name

lint – a C program checker

Syntax

lint [option] ... file ...

Description

The `lint` command attempts to detect features of the C program files that are likely to be errors, nonportable, or wasteful. It also checks type usage more strictly than the compilers. Among the things that are currently detected are unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, the usage of functions is checked to locate functions that return values in some places, but not in others, functions called with varying numbers or types of arguments, and functions whose values are not used or whose values are used but none returned.

Arguments whose names end with `.c` are interpreted as C source files. Arguments whose names end with `.ln` interpreted as the result of an earlier invocation of `lint` with either the **–c** or the **–o** option used. The `.ln` files are analogous to `.o` (object) files that are produced by the `cc` command when given a `.c` file as input. Files with other suffixes are warned about and ignored.

The `lint` command takes all the `.c, .ln,` and `llib-lx.ln` (specified by **–l***x*) files and processes them in their command line order. By default, the `lint` command appends the standard C `lint` library `(llib-lc.ln)` to the end of the list of files. However, if the **–p** option is used, the portable C `lint` library `(llib-port.ln)` is appended instead. When the **–c** option is not used, the second pass of `lint` checks this list of files for mutual compatibility. When the **–c** option is used, the `.ln` and the `llib-lx.ln` files are ignored.

Options

Any number of `lint` options may be used, in any order, intermixed with filename arguments. The following options are used to suppress certain kinds of warning:

–a Suppress warnings about assignments of long values to variables that are not long.

–b Suppress warnings about break statements that cannot be reached. (Programs produced by `lex` or `yacc` often result in such warnings).

–h Do not apply heuristic tests that attempt to intuit bugs, improve style, and reduce waste.

–u Suppress warnings about functions and external variables used and not defined, or defined and not used. (This option is suitable when running `lint` on a subset of files of a larger program).

–v Suppress complaints about unused arguments in functions.

–x Do not report variables referred to by external declarations but never used.

The following arguments alter the behavior of `lint`.

–l*x* Include additional `lint` library `llib-lx.ln.` For example, you can

include a `lint` version of the Math Library `llib-lm.ln` by inserting **–lm** on the command line. This argument does not suppress the default use of `llib-lc.ln`. These lint libraries must be in the assumed directory. This option can be used to reference local `lint` libraries and is useful in the development of multi-file projects.

–n Do not check compatibility against either the standard or the portable `lint` library.

–p Attempt to check portability to other dialects (IBM and GCOS) of C. Along with stricter checking, this option causes all non-external names to be truncated to eight characters and all external names to be truncated to six characters and one case.

–c Cause `lint` to produce a `.ln` file for every `.c` file on the command line. The `.ln` files are the product of the `lint` command's first pass only, and are not checked for inter-function compatibility.

–o lib
Cause `lint` to create a lint library with the name `llib-llib.ln`. The **–c** option nullifies any use of the **–o** option. The `lint` library produced is the input that is given to the second pass `lint`. The **–o** option simply causes this file to be saved in the named `lint` library. To produce a `llib-llib.ln` without extraneous messages, use of the **–x** option is suggested. The **–v** option is useful if the source files for the `lint` library are just external interfaces (for example, the way the file `llib-lc` is written). These option settings are also available through the use of `lint` comments which are described later.

The **–D**, **–U**, and **–I** options of `cpp` and the **–g** and **–O** options of `cc` are also recognized as separate arguments. The **–g** and **–O** options are ignored, but, by recognizing these options, the behavior of `lint` is closer to that of the `cc` command's second pass. Other options are warned about and ignored. The preprocessor 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`.

The `lint` command produces its first output on a per-source-file basis. Warnings regarding included files are collected and printed after all source files have been processed. Finally, if the **–c** option is not used, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a warning stems from a given source file or from one of its included files, the source file name is printed followed by a question mark.

The behavior of the **–c** and the **–o** options allows for incremental use of `lint` on a set of C source files. Generally, one invokes `lint` once for each source file with the **–c** option. Each of these invocations produces a `.ln` file which corresponds to the `.c` file, and prints all messages that are about just that source file. After all the source files have been separately run through `lint`, it is invoked once more (without the **–c** option), listing all the `.ln` files with the needed **–l***x* options. This prints all the inter-file inconsistencies. This scheme works well with `make`; it allows `make` to be used to `lint` only the source files that have been modified since the last time the set of source files were checked by `lint`.

Restrictions

The system call `exit`, the function `longjmp`, and other functions that do not return a value are not interpreted correctly by the `lint` command.

Certain conventional comments in the C source change the behavior of `lint`:

/*NOTREACHED*/ at appropriate points stops comments about unreachable code. (This comment is typically placed just after calls to functions like `exit)`.

/*VARARGS*n**/ suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0.

/*ARGSUSED*/ turns on the **–v** 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** options.

Files

/usr/lib/cmplrs/cc/lint the directory where the lint libraries specified by the **–l***x* option must exist

/usr/lib/cmplrs/cc/lint[12]
first and second passes

/usr/lib/cmplrs/cc/lint/llib-lc.ln
declarations for C Library functions (binary format; source is in `/usr/lib/cmplrs/cc/lint/llib-lc`)

/usr/lib/cmplrs/cc/lint/llib-lcV.ln
System V declarations for standard functions

/usr/lib/cmplrs/cc/lint/llib-lcP.ln
POSIX declarations for standard functions

/usr/lib/cmplrs/cc/lint/llib-port.ln
declarations for portable functions (binary format; source is in `/usr/lib/cmplrs/cc/lint/llib-port`)

/usr/lib/cmplrs/cc/lint/llib-lm.ln
declarations for Math Library functions (binary format; source is in `/usr/lib/cmplrs/cc/lint/llib-lm`)

/usr/tmp/*lint* temporaries

See Also

cc(1), cpp(1), make(1).

Name

lint – check C code

Syntax

lint [*Options*] *file...*

Description

The lint command detects features of the C program *files* which are likely to be bugs, non-portable, or wasteful. It also checks the type usage of the program more strictly than the compilers. Among the things which are currently found are unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, the usage of functions is checked to find functions which return values in some places and not in others, functions called with varying numbers of arguments, and functions whose values are not used.

By default, it is assumed that all the *files* are to be loaded together; they are checked for mutual compatibility. Function definitions for certain libraries are available to lint. These libraries are referred to by a conventional name, such as '–lm', in the style of ld(1). Arguments ending in .ln are also treated as library files.

To create lint libraries, use the **–C** option as follows:

```
lint -Cdlib files . . .
```

The C sources of library *dlib.* are *files*. The result is a file *llib-ldlib.ln* in the correct library format suitable for linting programs using *dlib*. Note that if you have set the System V environment variable the System V lint library is used. For further information, see intro(2).

Options

Any number of the options in the following list may be used. The **–D**, **–U**, and **–I** options of cc(1) are also recognized as separate arguments.

–a Report assignments of long values to int variables.

–b Report **break** statements that cannot be reached. (This is not the default because most lex and many yacc outputs produce dozens of such comments.)

–c Complain about casts that have questionable portability.

–h Apply a number of heuristic tests to attempt to find bugs, improve style, and reduce waste.

–n Do not check compatibility against the standard library.

–p Attempt to check portability to the IBM and GCOS dialects of C.

–u Do not complain about functions and variables used and not defined, or defined and not used. (This is suitable for running lint on a subset of files out of a larger program.)

–v Suppress complaints about unused arguments in functions.

–x Report variables referred to be extern declarations, but never used.

–Y*environment*
Compiles C programs for *environment*. If *environment* is SYSTEM_FIVE or omitted, defines SYSTEM_FIVE for the preprocessor, `cpp`, and if the loader is invoked, specifies that the System V version of the C runtime library is used. Also, if the math library is specified with the **–lm** option, the System V version is used. If *environment* is POSIX, defines POSIX for the preprocessor. If the environment variable PROG_ENV has the value SYSTEM_FIVE or POSIX, the effect is the same as specifying the corresponding **–Y***environment* option to `cc`. The **-Y** option overrides the PROG_ENV variable; **-YBSD** can be used to override all special actions.

–z Do not complain about structures that are never defined (for example, using a structure pointer without knowing its contents.)

Restrictions

The `exit`(2) system calls and other functions that do not return are not understood; this causes various anomalies in `lint` output.

Certain conventional comments in the C source change the behavior of `lint`:

/*NOTREACHED*/	At appropriate points stops comments about unreachable code.
/*VARARGS*n**/	Suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0.
/*NOSTRICT*/	Shuts off strict type checking in the next expression.
/*ARGSUSED*/	Turns on the **–v** option for the next function.
/*LINTLIBRARY*/	At the beginning of a file shuts off complaints about unused functions in this file.

Files

/usr/lib/lint1	Program
/usr/lib/lint2	Program
/usr/lib/lint/llib-lc.ln	Declarations for standard functions
/usr/lib/lint/llib-lc	Human readable version of above
/usr/lib/lint/llib-lcV.ln	System V declarations for standard functions
/usr/lib/lint/llib-lcP.ln	POSIX declarations for standard functions
/usr/lib/lint/llib-port.ln	Declarations for portable functions
/usr/lib/lint/llib-port	Human readable . . .
llib-l*.ln	Library created with **–C**

See Also

cc(1)

''Lint, a C Program Checker'', *ULTRIX Supplementary Documents,* Vol. II:Programmer

Name

lk – link editor

Syntax

lk [*option...*] *file...*

Description

The `lk` command combines several object programs into one, resolves external references, and searches libraries. In the simplest case, several object *files* are given, and `lk` combines them, producing an object module which can be executed. The output of `lk` is always a standard ULTRIX `a.out` object module. This file is made executable only if no errors occurred during the load.

The argument routines are linked together in the order specified. The entry point of the output is the beginning of the first routine, unless the **–e** option is specified.

If the argument is a library, it is searched only once at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. If a routine from a library references another routine in the library and the library has not been processed by `ranlib`(1), the referenced routine must appear after the referencing routine in the library. Thus the order of programs within libraries may be important. The first member of a library should be a file named '__.SYMDEF', which is a dictionary for the library as produced by `ranlib`(1). The dictionary is searched repeatedly to satisfy as many references as possible.

The symbols '_etext', '_edata' and '_end' ('etext', 'edata' and 'end' in C) are reserved, and if referred to, are set to the first location above the program, the first location above initialized data, and the first location above all data in that order. It is an error to define these symbols.

Like the `ld` linker, the `lk` linker can process ULTRIX object modules, `a.out` files, archived libraries (.a files), and ranlib-generated indexed libraries. Unlike the `ld` linker, however, the `lk` linker can also process object modules generated by the VAX FORTRAN compiler. All `lk` command options can also be specified on the `fort` command.

Options

The `lk` command has several options. Except for **–l**, they should appear before the file names.

–D *number* — Sets data segment length. The 'number' is a number specifying the desired length of the data segment. The linker pads the data segment to this length with zero bytes.

–e *symbol* — Take the argument as the name of the entry point of the loaded program. Location 0 is the default.

–H *number* — Takes number argument as a decimal integer, adds it to end of text, and starts data section at a higher address.

–K — Produces full load map, cross-referencing all defined symbols.

–L*dir* — Add *dir* to the list of directories in which libraries are

searched for. Directories specified with **–L** are searched before the standard directories.

–l*x*
Abbreviation for the library name '/lib/lib*x*.a', where *x* is a string. If that does not exist, `lk` tries '/usr/lib/lib*x*.a' A library is searched when its name is encountered, so the placement of a **–l** is significant.

–M
Produces full load map, consisting of a module and program section synopsis and symbol cross-reference. Only symbols that are referenced appear in the cross-reference. Use **–K** to cross-reference all symbols.

–N
Do not make text portion read only or sharable. (Use magic number 0407.)

–n
Arranges (by giving the output file a 0410 "magic number") that when the output file is executed, the text portion is read-only and shared among all users executing the file. This involves moving the data areas up to the first possible 1024 byte boundary following the end of the text.

–o *name*
Takes the *name* argument after **–o** as the name of the `lk` output file, instead of `a.out`.

–S
Strips the output by removing all symbols except locals and globals.

–s
Removes the symbol table and relocation bits to save space. This impairs the usefulness of the debuggers. This information can also be removed by `strip`(1).

–T *number*
Takes the argument as a hexadecimal number which sets the text segment origin. The default origin is 0.

–t
Displays the name of each file as it is processed.

–u *symbol*
Enters argument as undefined symbol in symbol table. This is useful for loading from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

–X
Saves local symbols except for those whose names begin with 'L'. This option is used by `cc`(1) to discard internally-generated labels while retaining symbols local to routines.

–x
Suppresses saving nonglobal symbols in output symbol table; enters only external symbols. This option saves some space in the output file.

–Y*environment*
Adjust the magic number in the output file so that the program runs in the specified *environment* . The parameter can be POSIX, SYSTEM_FIVE, or BSD. The parameter sets the program's execution environment to conform with one of the three standards. This parameter overrides the PROG_ENV environment variable, if it is present. If neither this parameter nor the PROG_ENV variable is present, –YBSD is assumed.

–y*sym* Indicates each file in which *sym* appears, its type and whether the file defines or references it. Many such options may be given to trace many symbols. (It is usually necessary to begin *sym* with an '_', as external C and PASCAL variables begin with underscores.)

–z Loads process on demand from the resulting executable file (413 format) rather than preloaded. This is the default. It results in a 1024 byte header on the output file followed by a text and data segment, whose size is a multiple of 1024 bytes (being padded out with nulls in the file if necessary). With this format the first few BSS segment symbols may actually appear, from the output of `size`(1), to live in the data segment. This avoids wasting the space which results from the data segment size roundup.

The `lk` linker does not support the following `ld` options: **–A, –d,** or **–r.**

Restrictions

The `lk` command pads the images which are to be demand loaded from the file system to the next page boundary.

When linking code containing GFLOAT instructions, the GFLOAT versions of *libc* and/or the math library must be used rather than the normal DFLOAT versions. Link to these by using *–lcg* and/or *–lmg*.

The compiler and the linker `lk`(1) cannot detect the use of mixed double floating point types, and your program may produce erroneous results.

Files

/lib/lib*.a	libraries
/usr/lib/lib*.a	libraries
/usr/local/lib/lib*.a	libraries
a.out	output file
a.map	map file

See Also

ar(1), as(1), cc(1), ld(1), ranlib(1)

Name

ln – link to a file

Syntax

ln [**–f**] [**–i**] [**–s**] *name1* [*name2*]
ln [**–f**] [**–i**] [**–s**] name ... directory

Description

A link is a directory entry referring to a file. A file, together with its size and all its protection information 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 may not refer to directories.

Given one or two arguments, `ln` creates a link to an existing file *name1*. If *name2* is given, the link has that name. The *name2* may also be a directory in which to place the link. Otherwise it is placed in the current directory. If only the directory is specified, the link is made to the last component of *name1*.

Given more than two arguments, `ln` makes links to all the named files in the named directory. The links made have the same name as the files being linked to.

Options

–f Forces existing *destination* pathnames to be removed before linking without prompting for confirmation.

–i Write a prompt to standard output requesting information for each link that would overwrite an existing file. If the response from standard input is affirmative, and if permissions allow, the link is done. The **–i** option has this effect even if the standard input is not a terminal.

–s Creates a symbolic link.

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 returns 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.

See Also

cp(1), mv(1), rm(1), link(2), readlink(2), stat(2), symlink(2)

Name

lock – reserve a terminal

Syntax

lock

Description

The `lock` command requests a password from the user, then asks you to confirm the password by typing it in again. The terminal refuses to relinquish the terminal until the password is repeated. If you forget the password, you must log in elsewhere and kill the lock process.

Restrictions

Should time out after 15 minutes.

Name

login – log in to a system

Syntax

login [*username*]

Description

The login command is used when a user initially signs on, or it may be used at any time to change from one user to another. The latter case is the one summarized above and described here. To sign on initially, see the *Guide to System Environment Setup*.

If login is invoked without an argument, it asks for a user name, and, if appropriate, a password. Echoing is turned off (if possible) during the typing of the password, so it does not appear on the written record of the session.

After a successful login accounting files are updated, the user is informed of the existence of mail, the message of the day is printed, and the time of last successful login is displayed. The display of all this information can be prevented by creating the file *.hushlogin* in the accounts login directory. This is useful for accounts such as *uucp*.

If ULTRIX security features are enabled additional things may happen. These include the display of the number of failed login attempts since the last successful login and forcing the setting of a new password. See the *ULTRIX Security Guide for Users and Programmers* for more information.

The login command initializes the user and group IDs, the working directory, and the users audit information, then executes a command interpreter, usually sh(1), according to specifications found in a password file. Argument 0 of the command interpreter is ''–sh'', or more generally the name of the command interpreter with a leading dash (''–'') prepended.

The login command also initializes the environment environ(7) with information specifying home directory, command interpreter, terminal type (if available) and user name.

When login is used in conjunction with getty(8) it is the responsibility of the getty program to initialize the terminal attributes. Specifically if a terminal is setup to use 8-bit characters the getty program should use a gettytab(5) entry which specifies 8-bit characters. If a terminal is setup in 8-bit mode but fails to specify an 8-bit gettytab entry, then characters output by both login and getty may appear as multinational characters.

If the file /etc/nologin exists, login prints its contents on the user's terminal and exits. This is used by shutdown(8) to stop users logging in when the system is about to go down.

The login command is recognized by sh(1) and csh(1) and executed directly (without forking).

If a root login is attempted and an invalid command interpreter is specified, the sh interpreter is used.

Options

–r Used by the remote login server, `rlogind`(8c), to force `login` to enter into an initial connection protocol.

–P *<programname>* Causes `login` to set it's standard input and output to be connected to the prompting program *<programname>*.

–C *string* Allows the system to specify a command to be run using the user's shell. This option causes a user shell **-c** *string* to be exec'ed.

–e Forces `login` to use an extended protocol when communicating with a prompter program (see –P).

Restrictions

To provide flow control, CTRL/S and CTRL/Q are ignored and are therefore invalid characters in a login name.

Diagnostics

Login incorrect
If the username and password are not a valid combination.

Too many users logged on already. Try again later.
The system has the maximum licensed number of users logged on already.

Requires secure terminal
An attempt was made to login as UID 0 on a line that is not marked as *secure* in `/etc/ttys`.

No shell
The login shell specified for the account cannot be executed. Consult the system administrator.

No directory! Logging in with home=/
The HOME directory for the account is inaccessible. This can happen if the directory resides on an NFS file system served by a host that is not currently available.

You have too many processes running
Completion of login would exceed the maximum number of running processes allowed for the user.

You have mail
You have a non-empty mail spool file.

If ULTRIX security features are enabled the following messages are also possible from login:

Your password has expired
The password for your account has not been changed recently enough. Consult your system administrator.

Your password has expired, please change it
Your password has expired recently. You will be forced to change it before you can proceed any further.

Your password will expire very soon
Your pasword will expire in less than 24 hours.

Your password will expire in %d days
The "%d" will be replaced with the number of days until your password expires. You should consider changing your password now.

This account is disabled
Consult your system administrator.

Kerberos initialization failure
Consult your system administrator.

Files

/etc/utmp	accounting
/usr/adm/wtmp	accounting
/usr/spool/mail/*	mail
/etc/motd	message-of-the-day
/etc/auth.[pag,dir]	authorization data base
/etc/passwd	password file
/etc/nologin	stops logins
/etc/svc.conf	sets I&A security level
.hushlogin	makes login quieter
/etc/securetty	lists ttys that root may log in on

See Also

mail(1), passwd(1), yppasswd(1yp), passwd(5yp), environ(7), getty(8), init(8), rlogind(8c), shutdown(8)
Guide to System Environment Setup
ULTRIX Security Guide for Users and Programmers
Security Guide for Administrators

Name

logname – get login name

Syntax

logname

Description

The `logname` command returns the a user's login name on the system.

Files

/etc/profile

See Also

login(1), environ(5)

Name

look – find lines in sorted data

Syntax

look [**–df**] *string* [*file*]

Description

The look command consults a sorted *file* and prints all lines that begin with *string*. It uses binary search. B

Options

The options **d** and **f** affect comparisons as in sort(1). If no *file* is specified, /usr/dict/words is assumed with collating sequence **–df.**

–d Uses dictionary order: only letters, digits, tabs and blanks can be compared.

–f Folds uppercase to lowercase (compares equally).

Files

/usr/dict/words

See Also

grep(1), sort(1)

Name

indxbib, lookbib – build inverted index for a bibliography, lookup bibliographic references

Syntax

indxbib *database...*
lookbib *database*

Description

The `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 ''%''.

The `indxbib` command is a shell script that calls `/usr/lib/refer/mkey` and `/usr/lib/refer/inv.` The first program, `mkey,` truncates words to 6 characters, and maps upper case to lower case. 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. The second program, *inv,* creates an entry file (.ia), a posting file (.ib), and a tag file (.ic), all in the working directory.

The `lookbib` command uses an inverted index made by `indxbib` to find sets of bibliographic references. It 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.

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

x.ia, *x*.ib, *x*.ic, where *x* is the first argument, or if these are not present, then *x*.ig, *x*

See Also

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

lorder(1)

Name

lorder – determine relation for an object library

Syntax

lorder *file*...

Description

The input is one or more object or library archive *files*. For further information, see `ar`(1). The standard output is a list of pairs of object file names, 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).

This one-liner intends to build a new library from existing '.o' files.

```
ar cr library `lorder *.o | tsort`
```

The use of `lorder`(1) is unnecessary when you use `ranlib`(1), which converts an ordered archive into a randomly accessed library.

Restrictions

The names of object files, in and out of libraries, must end with '.o'; nonsense results otherwise.

Files

*symref, *symdef

See Also

ar(1), join(1), ld(1), nm(1), ranlib(1), sed(1), sort(1), tsort(1)

Name

lp – send requests to an LP line printer

Syntax

lp [**–c**] [**–d** *dest*] [**–n** *number*] [–] [*files*]

Description

The `lp` command arranges for the named files and associated information (collectively called a *request*) to be printed by a line printer. If no file names are mentioned, the standard input is assumed. When a file name is designated by a minus sign (–) it 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 are printed.

This command exists for X/OPEN compatibility.

Options

The following options to `lp` may appear in any order and may be intermixed with file names:

–c Makes copies of the *files* to be printed immediately when `lp` is invoked. Normally, *files* are not copied, but are linked whenever possible. If the **–c** 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 without the **–c** option, any changes made to the named *files* after the request is made but before it is printed are reflected in the printed output.

–d *dest* Chooses *dest* as the printer that is to do the printing. If *dest* is a printer, then the request is printed on that specific printer. By default, *dest* is taken from the environment variable PRINTER if it is set. Otherwise, a default destination, lp, is used.

–n *number* Prints *number* copies (default of 1) of the output.

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), lpr(1), lprm(1), pr(1), symlink(2), printcap(2), lpc(8), lpd(8)

Name

lpq – spool queue examination program

Syntax

lpq [*options*] [*job #...*] [*user...*]

Description

The lpq command examines the spooling area used by lpd for printing files on the printer, and reports the status of jobs. The lpq command invoked without any arguments reports on any jobs currently in the default queue.

When jobs are being printed, lpq reports the queue as being ''active''. For each job submitted, lpq reports the user's name, current rank in the queue, the names of files comprising the job, the job identifier and the total size in bytes. The job identifier is a number which may be supplied to lprm to remove a specific job.

Job ordering is determined by the FIFO (First In, First Out) algorithm used to scan the spooling directory. When the queue is empty, lpq reports that there are ''no entries''.

File names may be unavailable for some jobs, for example, if the lpr command is used without specifying a file name, or if lpr is used in a pipe line. If file names are unavailable, the file name reported by lpq is ''standard input''.

Arguments

job #... Causes lpq to report on only the job number(s) specified.

user... Causes lpq to report on only the jobs for the specified user(s).

Options

+*n* Scan and display the spool queue until the queue is empty. The queue is scanned every *n* seconds, if no argument is specified the queue is scanned every 30 seconds.

–l Display the status of each job on more than one line if necessary. If this option is not used, the status of each job is displayed on one line.

–P*printer*
Report the status of the spool queue for the *printer* specified. If this option is not used, the spool queue displayed is the one defined by the PRINTER environment variable. If a queue is not defined by the PRINTER environment variable, the spool queue displayed is for the printer named ''lp'' in the printcap file.

Restrictions

The displayed status of the spool queue may not always be the current status. This is because jobs may be completed after the queue has been examined, but before the status has been displayed.

Error Messages

Two of the most common error messages from `lpq` are:

Warning: no daemon present
A daemon is not available for the specified printer. Refer to the `lpc(8)` command to find out how to restart the printer daemon.

unknown printer
The printer specified as an argument to the **–P** option, doesn't have an entry in the `/etc/printcap` file.

Files

`/etc/termcap`	For manipulating 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(8), lpd(8)

Name

lpr – off line print

Syntax

lpr [*options*] [*file...*]

Description

The `lpr` command puts files in the spooling area used by `lpd`. The files are printed by `lpd` when the printer is available. If no file names are specified, the standard input is used.

If options are specified which would cause a conflicting action, the last option specified is the one used. For example, in the command

```
lpr -h -Jjob
```

the **–J** option overrides the **–h** option.

Options

–C*class*
Print the argument *class*, as the job classification on the banner page. If this option is not used, the name of the node from where the `lpr` command was issued is printed.

–h Do not print the banner page.

–i*n* Indent the printed output by *n* spaces. An argument must be supplied with this option. You should note that this is not compatible to previous versions of `lpr`.

–J*job*
Print the argument *job*, as the job name on the banner page. If this option is not used, the job name is the name of the first file specified in the `lpr` command. If no file name is specified, the job name ''stdin'' is used.

–m Send a mail note to you when the job has been completed.

–p Format the files using the `pr` command.

–P*printer*
Send the output to the spool queue for the *printer* specified. If this option is not used, the output is sent to the spool queue defined by the PRINTER environment variable. If a queue is not defined by the PRINTER environment variable, the output is sent to the default printer.

–s Use the `symlink` system call to link data files, rather than trying to copy them. This can be used if the file size exceeds the spool directory limit. Refer to the **mx** capability in `printcap(5)`. Note that the files should not be modified or removed until they have been printed.

–T*title*
Print the argument *title* at the head of each page. If a title is not specified, the name of the file is used. If no file name is specified, then the title part of the header is left blank. The **–T** option is only meaningful with the **–p** option.

–w*n*

Print the job using a page width of *n* characters. If this option is not used, the page width is taken from the `printcap` file. If there is no entry in the `printcap` file, the page width used is 132 characters.

–z*n* Print the job using a page length of *n* lines. If this option is not used, the page length is taken from the `printcap` file. If there is no entry in the `printcap` file, the page length used is 66 lines.

–1*font*

–2*font*

–3*font*

–4*font*
Use the font file specified by *font* in font position **1**, **2**, **3** or **4**. These options can only be used with troff, ditroff and TeX files (the **–n**, **–t**, and **–d** options respectively).

–#*n* Print *n* copies of the specified file(s).

The following options are for use with PostScript (TM) printers with specialized support, refer to `lpd(8)`. Each option requires one argument. The arguments can be abbreviated as long as the abbreviations are unique for each option.

–D*datatype*
Define the data type to the print daemon, `lpd`. If the **–D** option is not used, the data type is taken from the `printcap` file. If no entry for the data type is found in the `printcap` file, the print job is sent to the printer without translation. The following are valid arguments for the **–D** option.

ansi	ANSI data
ascii	ASCII data
postscript	PostScript (TM) data
regis	REGIS data
tek4014	Tektronix 4014
xyz	You can specify other data types, but you must write an appropriate translator, refer to `xlator_call(8)`.

–F*pagesize*
Select the size of the pages to be printed. The page size is the text intended to be printed on a separate sheet. If the **–F** option is not used the page size used is the same as the sheet size, refer to option **–S**. If the sheet size is not specified, the value is taken from the `printcap` file. If there is no entry in the `printcap` file, the page size is LETTER (8.5 x 11 inches). The **–F** option is ignored if the data type is PostScript (TM). The following are valid arguments for the **–F** option.

letter or *a*	8.5 x 11 inches, 216 x 279 mm
ledger or *b*	11 x 17 inches, 279 x 432 mm
legal	8.5 x 14 inches, 216 x 356 mm
executive	7.5 x 10.5 inches, 191 x 254 mm
a5	5.8 x 8.3 inches, 148 x 210 mm
a4	8.3 x 11.7 inches, 210 x 297 mm
a3	11.7 x 16.5 inches, 297 x 420 mm
b5	7.2 x 10.1 inches, 176 x 250 mm
b4	10.1 x 14.3 inches, 250 x 353 mm

–I*tray*

Select the input paper tray that will supply paper for the print job. The tray name is given by the argument as follows:

top	The upper 250-sheet input tray.
middle	The middle 250-sheet input tray.
bottom or *lcit*	The large capacity input tray.

If the **–I** option is not used, the **–S** option to selects the input tray. If the **–I** option and the **–S** option are both specified, the input tray must contain the required paper size. If the **–I** option is not specified, the value for the output paper tray is taken from the `printcap` file. If no entry is present there, the default paper tray for the printer is used.

–K*sides*

Print the pages of the job on sheets in the way specified by *sides*. The valid arguments are:

1 or *one_sided_simplex*	Print on one side of the sheet only.
2 or *two_sided_duplex*	Print on both sides of the sheet, the second side is reached by flipping the sheet about its left edge, as in the binding of a book.
tumble or *two_sided_tumble*	Print on both sides of the sheet, but print the opposite way up on each side, so the second side can be read by flipping the sheet along its top axis.
one_sided_duplex	Print on one side of the paper only, but retain the page layout intended for two sided duplex printing. The layout refers to such things as where the margins are, and where the page numbers are.
one_sided_tumble	Print on one side of the paper only, but retain the page layout intended for tumble printing.
two_sided_simplex	Print on two sides of the paper of the paper, but retain the page layout intended for one sided simplex.

–L*filename*

Use the commands in the layup definition file, specified by *filename*, to alter the appearance of the printed output. If *filename* does not begin with /, the current directory is searched, followed by `/usr/lib/lpdfilters`. Refer to the documentation for your printer for the commands available.

–M*message*

Use the messages generated by the print job in the way specified by *message*. If the **–M** option is not used, messages are not recorded, unless indicated by an entry in the `printcap` file.

keep	Record the messages in the message file and mail the file to you.
ignore	Do not record messages.

–N*n*

Print *n* pages on a single sheet. The number must be in the range 0 to 100. If you specify `lpr -N0` the `/etc/printcap` entry is overridden and the default layup file is not used. If you specify `lpr -N1` the pages are printed "1-up" but with a border. If the **–N** option is not used, one page is printed on one sheet.

–o*tray*

Select the output tray where the printed job will be deposited. The tray name is given by the argument as follows:

top	Top tray, with face-down stacking.
side	Side tray, with face-down stacking.
face-up	Side tray, with face-up stacking.
upper	Upper tray if there are two trays on top of the printer. If there are not two trays, the *top* tray is used.
lower	Lower tray if there are two trays on top of the printer. If there are not two trays, the *top* tray is used.
lcos	Large capacity output stacker

If the **–o** option is not specified, the value for the input paper tray is taken from the `printcap` file, and if no entry is present there, from the printer.

–O*orientation*

Print the page in the way specified by *orientation*. The orientation is given by the argument as follows:

portrait	The printed output is parallel to the short side of the page.
landscape	The printed output is parallel to the long side of the page.

If the **–O** option is not specified, the value orientation is taken from the `printcap` file, and if no entry is present there, from the printer.

–S*pagesize*

Select the physical size of the sheets to be printed. If the **–S** option is not used the sheet size used is the same as the page size, refer to option **–F**. If the page size is not specified, the value is taken from the `printcap` file. If there is no entry in the `printcap` file the sheet size is LETTER (8.5 x 11 inches). The valid arguments for the **–S** option are the same as for the **–F** option.

–X*n*

Print each page *n* times. The number must be in the range 1 to 100. The output is uncollated, for a collated output use the **–#** option. If the **–X** option is not used, each page is printed once.

–Z*lowlim,uplim*

Print the pages of the job between *lowlim* and *uplim*. If *lowlim* is not specified, the first page printed is the first page of the job. If *uplim* is not specified, the last page printed is the last page of the job. The maximum value which can be specified for *uplim* is 10000. Banner pages are not included in the count. Note that these limits apply to the entire print job, not to individual files within a multi-file job.

The following options are used to notify the spooling daemon for the printer that the files are not standard text files. Any of these options will override the **–D** option regardless of the order in which they appear. The `lpd` print daemon uses the appropriate filters to ensure the files are printed correctly.

-g Assume the files contain standard plot data produced by the plot routines.

-l Print the files using a filter which prints the control characters and suppresses the page breaks.

-t Assume the files contain data produced by troff.

-x Assume the files do not require filtering before printing.

The following options also notify the spooling daemon for the printer that the files are not standard text files. The lpd filters for the following options are not supplied as part of the standard ULTRIX operating system.

-c Assume the files contain data produced by cifplot.

-d Assume the files contain data produced by TeX (DVI output from Stanford).

-f Interpret the first character of each line as a standard FORTRAN carriage control character.

-n Assume the files contain data produced by device independent troff (ditroff).

-v Assume the files contain a raster image for devices like Versatec.

Restrictions

Fonts for troff and TeX reside on the host with the printer. It is not possible to use local font libraries.

Diagnostics

Files with more than x bytes are truncated to x bytes. The default value for x is 1025024 bytes, but this can be changed by using the **mx** capability in the /etc/printcap file. The lpr command will not print files which appear to be in a.out or ar format. If a user other than root prints a file and spooling is disabled, lpr will print a disabled message and will not put jobs in the queue. If a connection to lpd on the local machine cannot be made, lpr will print that the daemon cannot be started.

Files

/etc/passwd	Personal identification
/etc/printcap	Printer capabilities data base
/usr/lib/lpd	Line printer daemon
/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(5), lpc(8), lpd(8)

Name

lprm – remove jobs from line printer queue

Syntax

lprm [**–P***printer*] [–] [*job #...*] [*user...*]

Description

The `lprm` command removes a job, or jobs, from a printer's spool queue. Since the spooling directory is protected from users, using `lprm` is normally the only method by which a user may remove a job.

The `lprm` command without any arguments deletes the currently active job if it is owned by the user who invoked `lprm`.

If the – flag is specified, `lprm` removes all jobs which a user owns. If the super-user 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.

Specifying a user's name, or list of user names, causes `lprm` to attempt to remove any jobs queued belonging to that user (or users). This form of invoking `lprm` is useful only to the super-user.

A user may dequeue an individual job by specifying its job number. This number may be obtained from the `lpq`(1) program. For example,

```
% lpq -l

1st: ken      [job #013ucbarpa]
       (standard input)    100 bytes
% lprm 13
```

The `lprm` command announces the names of any files it removes and is silent if there are no jobs in the queue which match the request list.

The `lprm` command kills off an active daemon, if necessary, before removing any spooling files. If a daemon is killed, a new one is automatically restarted upon completion of file removals.

Options

–	Removes all jobs owned by you only.
–P *printer*	Removes jobs from specified printer. It may be used to specify the queue associated with a specific printer (otherwise the default printer, or the value of the PRINTER variable in the environment is used).

Restrictions

Since there are race conditions possible in the update of the lock file, the currently active job may be incorrectly identified.

Diagnostics

''Permission denied" if the user tries to remove files other than his own.

Files

/etc/printcap	printer characteristics file
/usr/spool/*	spooling directories
/usr/spool/*/lock	lock file used to obtain the pid of the current daemon and the job number of the currently active job

See Also

lpq(1), lpr(1), lpd(8)

Name

lpstat – printer status information

Syntax

lpstat [*options*]

Description

The `lpstat` utility prints the status of the system printers.

Without any options, `lpstat` prints the status of print requests made to the default printer.

This command exists for X/OPEN compatibility.

Options

Some of the options can be followed by a list of arguments. The arguments must be specified as follows:

```
lpstat -uuser1,user2,user3
```

List items can be separated by spaces, but the list must be enclosed in quotes. If you do not include any arguments, all the information relevant to the option is printed.

The valid options for `lpstat` are:

–a [*printer1, printer2, ...*]

Print whether or not printers are accepting print requests.

–d Print the name of the default system printer.

–o [*printer1, printer2, ...*]

Print the status of print requests.

–p [*printer1, printer2, ...*]

Print the status of printers.

–r Print the status of the line printer daemon, `lpd`.

–s Print a status summary, including the status of the line printer daemon `lpd`, and the default system printer.

–t Print all status information.

–u [*user1, user2, ...*]

Print the status of users' print requests.

See Also

lp(1), lpq(1), lpr(1), lpc(8)

Name

ls – list and generate statistics for files

Syntax

ls [*options*] *name ...*

Description

For each directory argument, `ls` lists the contents of the directory. For each file argument, `ls` repeats the file name and gives any other information you request with the options available. By default, the list is sorted alphabetically. When no argument is given, the current directory is listed. When several arguments are given, files are listed first, followed by directories and the files within each directory. Options are listed below.

Options

–1 Displays one entry per line. This is the default when output is not to a terminal.

–a Displays all entries including those beginning with a period (.).

–C Forces multicolumn output for pipe or filter. This is the default when the output is to a terminal.

–c Uses time of last modification of file status information (file creation, mode, etc) for sorting (with the **–t** option) or printing (with the **–l** option) rather than the time of file modification or access. See also the **–t** and **–u** options.

–d Displays names of directories only, not contents. Use this option with **–l** to get the status of a directory.

–F Marks directories with trailing slash (/), sockets with a trailing equal sign (=), symbolic links with a trailing at sign (@), and executable files with a trailing asterisk (*).

–f Displays names in the order they exist in directory. For further information, see `dir`(5). Entries beginning with a period (.) are also listed. This option overrides the **–l, –t, –s,** and **–r** options.

–g Displays assigned group ID (used with **-l** only). Default is assigned owner ID.

–i Displays the i-number for each file in the first column of the report.

–L Lists the information, if the file is a symbolic link, for the file or directory the link references rather than that for the link itself.

–l Lists the mode, number of links, owner, size in bytes, and time of last modification for each file. If the file is a special file, the size field contains the major and minor device numbers instead of the size. If the file is a symbolic link, the pathname of the linked-to file is printed, preceded by ''–>''.

The mode field consists of 11 characters. The first character indicates the type of entry:

d if the entry is a directory
b if the entry is a block-type special file

c if the entry is a character-type special file
l if the entry is a symbolic link
s if the entry is a socket
– if the entry is a plain file

The next 9 characters are interpreted as three sets of three characters each. The first set of three characters refers to file-access permissions for the user; the next set, for the user-group; and the last set, for all others. 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 the set-group-id bit set; likewise, the user-execute permission character is given as **s** if the file has the set-user-id bit set.

The last character of the mode (normally 'x' or '–') is **t** if the 1000 bit of the mode is on. See `chmod`(2) for the meaning of this mode.

–q Forces the printing of nongraphic characters in file names as the question mark character (?). This is the default when output is to a terminal.

–R Recursively lists all subdirectories.

–r Sorts entries in reverse alphabetic or time order.

–s Displays the size in kilobytes of each file. This is the first item listed in each entry.

–t Sorts by time modified (most recently modified first) instead of by name. See also the **–c** and **–u** options.

–u Uses the time of last access instead of last modification for sorting (with the **–t** option) or printing (with the **–l** option).

Restrictions

The output device is assumed to be 80 columns wide.

New line and tab are considered printing characters in file names.

The option setting based on whether the output is a teletype is undesirable as "ls -s" is much different than "ls -s |lpr". On the other hand, not doing this setting would make old shell scripts which used **ls** almost certain to fail.

Files

`/etc/passwd` Used to obtain user id's for **ls –l**

`/etc/group` Used to obtain group id's for **ls –g**

Name

ltf – labeled tape facility

Syntax

ltf *option* [*keys*] *file...*

Description

The ltf command reads and writes single-volume Versions 3 and 4 ANSI-compatible tape volumes. For a description of the label conventions, see ltf(5). The *file* argument specifies each file or directory name that is to be processed. If a directory name is specified, the complete directory tree is processed.

Options

The actions of ltf are controlled by one of the following option characters that must appear as the first command-line argument: **–c, –H, –t,** and **–x.**

–c Creates a new volume assigning an interchange file name to the files on the volume. That is, ltf initializes the volume and writes each named file onto the output file. Then ltf assigns an "interchange" file name to the files being created on the volume. This "interchange" file name is a name that can be recognized by a non-ULTRIX system. (Permissible ULTRIX file names are not allowed in all forms of ANSI volumes). This file name is 17 characters in length and includes only capital letters and the "a" characters, see ltf(5). It is formed by converting all lower case letters to upper case, converting non-"a" characters to upper case Z, and truncating the resultant string to 17 characters. If ANSI Version 4 volumes are being used, the original ULTRIX file name is preserved in HDR3 through HDR9 and EOF3 through EOF9. For further information, see ltf(5).

–H Displays help messages for all options and keys.

–t Lists each named file on the specified volume. If no file argument is given, information about all files on the volume is provided. If **–t** is used without v or V (verbose keys), the interchange file names are also included in the list.

–x Extracts each named file from the volume to the user's current directory. If no file argument is given, the entire content of the volume is extracted. If the **p** key is not specified when extracting files from a volume written by an ULTRIX system, the files are restored to the current user and group IDs and to the mode set by the umask(2) system call.

Keys

The following optional *keys* can be specified to enable or disable ltf actions as specified:

a Outputs an ANSI-compatible Version 3 format volume. This key can be used with the **–c** option only. The default version is 4. For further information, see ltf(5).

h Write to a tape volume the file that a symbolic link points to instead of creating the symbolic link on a volume. The file written to the tape now has the same name as the symbolic link. This key can be used with the **–c** option only.

When extracting, if a symbolic link exists in the current directory that has the same name as a file on the tape volume, the link is followed and the file that the symbolic link currently points to is overwritten with the extracted file. To avoid overwriting files, use the **w** key.

o Omits directory blocks from the output volume. When creating a volume, the directory files are omitted, and when listing or extracting, the V key is disabled.

O Omits usage of optional headers HDR3 through HDR9 and EOF3 through EOF9. For further information, see `ltf`(5). If a file is created on an ULTRIX system without the use of the O key, these file headers contain the complete ULTRIX disk file name. Some non-ULTRIX systems are not able to process volumes containing these header labels. Thus, it is helpful to use this qualifier to avoid unnecessary error messages when planning to use non-ULTRIX systems.

p Restores files to original mode, user ID and group ID that is written on the tape volume. This key can be used with the **–x** option on ULTRIX files and by the superuser only.

v Displays long form information about volume and files. Normally, `ltf` operates with little terminal output.

When used in conjunction with the **–t** option, **v** gives more information about the volume entries than when used in conjunction with the **–c** and **–x** options. The following line is typical output from **–tv** functions.

```
ltf:  Volume ID is:  ULTRIX   Volume is:  ANSI Version #4
ltf:  Owner  ID is:  OwnerID
ltf:  Implementation ID is:   SystemID
ltf:  Volume  created   on:   System

t(1,1) rw-r--r-- 103/3 owner Feb 2 12:34 2530 bytes <cc >D file1
t(2,1) rw-r--r-- 103/3 owner Jun 29 09:34 999 bytes <com>D file2
t(3,1) rwxrwxrwx 293/10 name Jan 24 10:20 1234 bytes <bin>F name
t(4,1) --xrwx--- 199/04 theowner Jan 24 10:21 12345 bytes <asc>D
                long file name
```

The first field contains the file sequence number and the file section number of the file. If an ULTRIX system created the labeled volume, the second and third fields contain the mode, and owner/group ID of the file. Otherwise, these two fields are filled with dashes. The fourth field contains the file owner name. The fifth field contains latest modification time. The year is included if the modification time is older than Jan 1 of the current year. The sixth field contains the number of bytes used on the volume for the file. If the volume is non-ULTRIX, this field contains the number of blocks with the block size in parenthesis. The seventh field contains the ANSI file type (angle brackets) and the file record format (one character suffix). The file record formats are: F (fixed length), D (variable length), or S (spanned/segmented records). The eighth (last) field contains the name of the file. If the file name does not fit within the 12 spaces left in the line, the name appears on the next line preceded by a carriage return. A long file name will be continued over one or more lines thus it is recommended to keep auto wrap on in the terminal setup. Also if a file on a volume is either a symbolic or hard link, information about the linked file is displayed on the next line, preceded by a carriage return.

V Displays verbose information about directories.

w Warns the user if file name is in danger of being truncated when using **–c** or if it could be overwritten using **–x.** Normally, `ltf` operates silently and does not let the user know what is happening. When **–cw** is specified, `ltf` displays two warning messages if the interchange name and the ULTRIX file name are not the same. When **–xw** is specified, `ltf` displays a warning message if a file is about to be overwritten. Another message is displayed asking for approval to overwrite the file. If the user types no or presses return, then the option exists to type in a new file name or press return to quit. If a new file name is typed, this name is also checked. Thus, `ltf` does not continue until a unique file name is typed. When **–x** is specified, `ltf` does not warn the user if a directory name already exists.

0..31
Selects a unit number for a named tape device. These unit numbers can be entered when using the default tape name, /dev/rmt0h.

The following optional keys require an additional argument to be specified on the command line. If two or more of these keys are used, their respective arguments are to appear in the exact order that the keys are specified.

B *size*
Set the blocking factor to *size*. This specifies the maximum number of bytes that can be written in a block on a volume. If no value is specified, *size* defaults to 2048 bytes. The maximum size is 20480 bytes and the minimum size is 18 bytes. The **B** key need only be specified with **–c.**

The *size* may be specified as n bytes, (where n is assumed to be decimal) or as nb, (a multiple of 512 bytes using n followed by 'b', where 'b' signifies the multiple of 512) or as nk, (a multiple of 1024 bytes using n followed by 'k', where 'k' signifies the multiple of 1024).

f *device*
Sets the device file name to *device*. The default is /dev/rmt0h. The use of the **f** key overrides the **0..31** keys.

I *file*
Allows file name to be supplied either interactively or from a specified file. Normally, `ltf` expects the argument file names to be part of the command line. The **I** key allows the user to enter argument file names either interactively or from a specified file. If *file* is a dash (–), `ltf` reads standard input and prompts for all required information. All of the file names are requested first, followed by a single return before the arguments are processed. If *file* is a valid file name, *file* is opened and read to obtain argument file names.

L *label*
Specifies a six-character volume identifier *label*. The default *label* for ULTRIX systems is 'ULTRIX'.

P *position*
Specifies file sequence and section number at which volume will be positioned, using #,#. The first #, represents the file sequence number, while second #, the file section number. The file sequence number begins at 1 and is incremented for each file in the current file set. Since this implementation of `ltf` only produces one file set, the file sequence number for volumes written with this implementation is the number of the file as it is written on the volume. The file section number begins at 1 and is incremented for each file section on any one

volume. This number is necessary when files are written in multi-volume format where the need may exist to split a file across volumes; however since this implementation of `ltf` writes only single volumes, the file section number is always 1 for volumes written with this implementation. If no file arguments are specified, all files from the position number to the end of the tape are listed or extracted. Otherwise, particular files that exist between the position number and the end of the tape can be listed or extracted. A warning message appears if a file is requested that exists before the position number specified. The **P** key cannot be used with the **–c** option.

Examples

```
ltf -cfB /dev/rmt0h 100 file1 file2 file3
```

This example creates a new volume for file1, file2, and file3 using device /dev/rmt0h (**f** key) and a blocking factor of 100 (**B** key).

Restrictions

The `ltf` command does not support floppy diskettes or multi-volume tapes.

Diagnostics

Diagnostics are written to the standard error file. They come in four forms: fatal errors, warnings, information, and prompts. The `ltf` command terminates when it detects that a fatal error has occurred.

The diagnostics are intended to be self-explanatory. Their general format is:

```
ltf: FATAL > a fatal error message
ltf: Warning > a warning or advisory message
ltf: Info > an information message
ltf: a prompt asking for input
```

See Also

ltf(5)

Index

Special Characters

? command (TELNET), 1–690
? command (tftp), 1–698

Numbers

2780e emulator spooler, 1–2
See also 3780e emulator spooler
3780e emulator spooler, 1–3
See also 2780e emulator spooler

A

A C program checker
lint(1), 1–353
account command (ftp), 1–251
adb debugger, 1–5, 1–11
See also gcore command
addresses, 1–10
command list, 1–7, 1–10
core file, 1–5
diagnostics, 1–11
dyadic operators, 1–6
expressions, 1–5
monadic operators, 1–6
od command, 1–5
options, 1–5
restricted, 1–11
variables, 1–10
addbib program, 1–12
keyletters, 1–12
options, 1–12
admin command (sccs), 1–14 to 1–18, 1–599
See also delta command (sccs)
See also val command (sccs)
admin command (sccs) (cont.)
See also vc command (sccs)
options, 1–14
ali command, 1–19
alias command (csh), 1–128
alias command (mail), 1–396
See also unalias command (mail)
alias command (pdx), 1–507
aliases file
rebuilding, 1–463
alloc command (csh), 1–128
alternates command (mail), 1–396
anno command, 1–21
annotating messages, 1–21
append command (ftp), 1–251
apply program, 1–22
restricted, 1–22
apropos command, 1–23
ar program, 1–27
See also nm command
See also ranlib command
options, 1–27
restricted, 1–28
archive file
copying, 1–105, 1–107
maintaining, 1–27
ordering, 1–370
printing object files, 1–475
reconstructing, 1–557, 1–558
arithmetic language
See bc language
arithmetic package
See dc program

as assembler, 1–33
as command (RISC), 1–29
ascii command (ftp), 1–251
ascii command (tftp), 1–698
assign command (pdx), 1–506
at command, 1–34
restricted, 1–35
auth database
examination, 1–628
shexp command, 1–628
awk programming language, 1–36, 1–38
See also nawk utility
See also sed stream editor
built-in functions, 1–37
restricted, 1–38
statement list, 1–36

B

basename command, 1–39
bc language, 1–40 to 1–42
See also dc program
dc program and, 1–41
restricted, 1–42
bdiff command, 1–43
See also diff command
bell command (ftp), 1–251
bg command (csh), 1–128
bibliography
creating, 1–12
editing, 1–12
finding references, 1–369
formatting, 1–586
indexing, 1–369
searching, 1–567
sorting, 1–646
biff command, 1–44
binary command (ftp), 1–251
binary command (tftp), 1–698
binary file
finding printable strings, 1–656
installing, 1–292
sending in mail, 1–729
binmail program, 1–45
See also mail program
command reference list, 1–45
options, 1–46
restricted, 1–46
blank
defined, 1–617
Bourne shell
sh command interpreter, 1–713
break command (csh), 1–128
break command (sh), 1–614
break command (System V), 1–623
breaksw command (csh), 1–128
broadcast message
sending, 1–759
bsf command (mt), 1–444
bsr command (mt), 1–444
burst command, 1–47
bye command (ftp), 1–251

C

C compiler
See cc compiler
C flow graph
See cflow command
C program
building cross-reference table, 1–157
creating error message file, 1–434
displaying call graph profile data and, 1–271
displaying on standard output, 1–56
formatting, 1–289 to 1–291
implementing shared constant strings, 1–777
verifying, 1–356
C shell
See csh command interpreter, 1–118
cache command (mt), 1–444
cal command, 1–49
restricted, 1–49
calendar
printing, 1–49
calendar command, 1–50
See also leave command
restricted, 1–50

call command (dbx), 1–173
call command (pdx), 1–506
capsar utility, 1–51
case command (csh), 1–129
case command (ftp), 1–251
case command (sh), 1–610
case command (System V), 1–618
cat command, 1–54
  See also more command
catch command (dbx), 1–172
catpw command
  reference page, 1–55
cb program, 1–56
cc compiler, 1–64 to 1–67
  See also ctags command
  See also ctrace debugger
  See also cxref command
  See also gprof command
  See also ld command
  See also lk command
  See also mkstr command
  See also prof command
  See also xstr command
  diagnostics, 1–153
  options, 1–64 to 1–66, 1–745
  restricted, 1–66
ccat command, 1–94
cd command (csh), 1–68, 1–129
cd command (ftp), 1–251
cd command (sh), 1–68, 1–614
cd command (System V), 1–68, 1–623
cdc command (sccs), 1–69, 1–70
  restricted, 1–70
cdoc command, 1–71
cdup command (ftp), 1–251
cflow command, 1–73
  options, 1–74
  restricted, 1–74
changequote macro, 1–390
character
  translating, 1–708
chdir command (csh), 1–129
chdir command (mail), 1–396
check command (sccs), 1–599
checknr command, 1–75
  options, 1–75
chfn command, 1–77
  See also finger command
  restricted, 1–77
chgrp command, 1–78
  See also install command
chmod command, 1–79, 1–81e
  See also install command
  restricted, 1–80
chsh program, 1–82
clean command (sccs), 1–599
clear command, 1–83
clhrdsf command (mt), 1–444
close command (ftp), 1–252
close command (TELNET), 1–689
clserex command (mt), 1–444
clsub command (mt), 1–444
cmp command, 1–84
col command, 1–85
  restricted, 1–85
colcrt command, 1–86
  See also ul command
colrm command, 1–87
column
  filtering multiple, 1–85
  removing from file, 1–87
comb command (sccs), 1–88
  options, 1–88
  restricted, 1–88
comm command, 1–90
command
  applying to arguments, 1–22
  executing later, 1–34
  getting online information, 1–762
  locating online information, 1–409, 1–765
  showing executed, 1–337
  timing, 1–701
comp command, 1–91
compact command, 1–94
comparing files with cmp, 1–84
comparing files with comm, 1–90

compiler
creating, 1–779
compress command, 1–96, 1–98
compressing sparse data files, 1–596
connect command (tftp), 1–698
cont command (dbx), 1–172
cont command (pdx), 1–506
continue command (csh), 1–129
continue command (sh), 1–614
continue command (System V), 1–623
copy command (mail), 1–396
See also save command (mail)
copying sparse data files, 1–596
cord command, 1–100
cp command, 1–104
See also dd command
See also mv command
cpio command, 1–105, 1–106, 1–107
ar command, 1–105
function keys, 1–106
options, 1–105
restricted, 1–107
cpp command, 1–108, 1–111
cpustat command (SMP), 1–114
cr command (ftp), 1–252
crash dump
anyalyzing, 1–544
create command (sccs), 1–599
creating messages, 1–91
crypt command
encryption, 1–116
csh command interpreter, 1–118
See also echo command
argument list processing, 1–138
built-in commands, 1–128
command definition, 1–119
command input/output, 1–126
command substitution, 1–125
expressions, 1–127
file name substitution, 1–125
flow of control, 1–128
lexical structure, 1–118
non-built-in commands, 1–138
quoted strings and, 1–122
csh command interpreter (cont.)
repeating commands, 1–120
reporting job status, 1–120
restricted, 1–143
running jobs, 1–119
signal handling, 1–139
substituting an alias, 1–122
variable substitution, 1–123, 1–125
variables, 1–135
csplit command, 1–144
ctags command, 1–146
options, 1–146
restricted, 1–147
ctc command, 1–149
ctcr command, 1–149
ctod command, 1–148
ctrace command, 1–150e
ctrace debugger, 1–149 to 1–154
diagnostics, 1–152
options, 1–149
restricted, 1–152
statement-by-statement control, 1–151
cu command, 1–702
cut command, 1–155
See also paste command
options, 1–155
cxref command, 1–157

D

date
printing, 1–158
setting, 1–158
showing, 1–756, 1–768
date command, 1–158, 1–159e
diagnostics, 1–160c
field descriptors, 1–158
multiuser mode and, 1–160c
dbx command (RISC only), 1–161
dbx debugger, 1–170
See also gcore command
accessing source files, 1–174
adb debugger, 1–170
arguments, 1–170

dbx debugger (cont.)
command aliases, 1–174
executing commands, 1–171
machine-level commands, 1–176
miscellaneous commands, 1–176
options, 1–170
printing variables, 1–173
restricted, 1–177
dc program, 1–178
See also bc language
diagnostics, 1–180
dd command, 1–181 to 1–183
diagnostics, 1–183
example, 1–182
options, 1–181 to 1–182
restricted, 1–183
debug command (ftp), 1–252
debugger
dbx command, 1–161
source-level, 1–161
decompressing sparse data files, 1–596
define macro, 1–390
deledit command (sccs), 1–600
delete command (dbx), 1–172
delete command (ftp), 1–252
delete command (mail), 1–396
See also undelete command (mail)
delete command (pdx), 1–506
delget command (sccs), 1–600
delta
defined, 1–598
delta command (sccs), 1–184 to 1–186, 1–599
See also rmdel command (sccs)
cdc command (sccs), 1–69
keyletters, 1–184 to 1–185
restricted, 1–185
deroff interpreter, 1–187
restricted, 1–187
df command
See also dumpfs command
dgate command, 1–190
dgated daemon, 1–190
diagnostics
explained, 1–1
diagnostics (cont.)
handling, 1–758
diction program, 1–191
restricted, 1–191
diff command, 1–192
diagnostics, 1–194
restricted, 1–193
diff3 command, 1–195
restricted, 1–196
diffmk command, 1–197
restricted, 1–197
diffs command (sccs), 1–600
dir command (csh), 1–129
dir command (ftp), 1–252
dircmp command, 1–198
directory
comparing, 1–192
creating, 1–432
listing, 1–382
removing, 1–580
dirname command, 1–199
disconnect command (ftp), 1–252
disk
displaying free space, 1–188
displaying usage, 1–556
displaying used space, 1–188
reporting I/O statistics, 1–295
reporting statistics, 1–755
summarizing usage, 1–207
disk quota
displaying, 1–556
display command (TELNET), 1–690
display folders
pathname, 1–430
displaying folders, 1–243
dist command, 1–201
divert macro, 1–390
divnum macro, 1–391
dnl macro, 1–391
domain
defined, 1–204
getting name, 1–204
setting name, 1–204

domainname command, 1–204
dp command (mail), 1–396
dtoc command, 1–205
du command, 1–207
restricted, 1–207
dump command (pdx), 1–507
dumpdef macro, 1–392

E

echo arguments, 1–209
echo command (csh), 1–129
echo command (general), 1–208
echo command (System V), 1–623
ed line editor, 1–210
command list, 1–213
constructing addresses, 1–212
constructing regular expressions, 1–210
diagnostics, 1–218
interrupt signal, 1–218
restricted, 1–218
edit command (mail), 1–396
edit command (pdx), 1–507
edit command (sccs), 1–599
editors
ed, 1–210
edit, 1–223
ex, 1–223
red, 1–210
sed, 1–604
vi (screen), 1–749
egrep command, 1–276
else command (csh), 1–130
encryption
crypt command, 1–116
ex editor, 1–223
secret mail, 1–776
vi screen editor, 1–749
view command, 1–752
end command (csh), 1–130, 1–135
endif command (csh), 1–131
endnote
formatting, 1–567
environment
printing variable values, 1–530
eof command (mt), 1–444
eotdis command (mt), 1–444
eoten command (mt), 1–444
error command, 1–220 to 1–222
options, 1–221
restricted, 1–222
error message
producing, 1–208
viewing in source code, 1–220 to 1–222
errprint macro, 1–392
eval command (csh), 1–129
eval command (sh), 1–614
eval command (System V), 1–623
eval macro, 1–391
ex editor, 1–223
exec command (csh), 1–129
exec command (sh), 1–614
exec command (System V), 1–623
exit code
exit status, 1–1
exit command (csh), 1–129
exit command (mail), 1–396
exit command (sh), 1–614
exit command (System V), 1–624
exit status
defined, 1–1
expand command, 1–225
See also fold command
Expanding packed messages, 1–47
explain program, 1–191
export command (sh), 1–614
export command (System V), 1–624
expr command, 1–226
examples, 1–226
expression
taking arguments as, 1–226
extract utility, 1–228
eyacc compiler, 1–232

F

false command, 1–713

fg command (csh), 1–129

fgrep command, 1–276

file

- *See also specific files*
- appending, 1–688
- backing up, 1–680 to 1–683
- backing up multiple, 1–418
- breaking into pieces, 1–652
- changing tabs to blanks in, 1–225
- combining, 1–300
- comparing, 1–43, 1–84, 1–90, 1–192, 1–195, 1–197, 1–300, 1–641, 1–724
- compressing, 1–94
- converting, 1–181
- converting to sccs format, 1–598e
- copying, 1–104
- copying portions, 1–677
- copying remote, 1–559
- cutting fields from, 1–155
- determining extension, 1–233
- displaying, 1–54, 1–94
- displaying first lines, 1–279
- dumping in various format, 1–490
- finding, 1–234
- finding executable, 1–767
- finding pattern, 1–276, 1–368
- getting block count, 1–673
- getting character count, 1–760
- getting line count, 1–760
- getting word count, 1–760
- listing information, 1–382
- merging, 1–641
- merging horizontally, 1–501
- moving, 1–446
- overwriting, 1–688
- printing, 1–374
- printing at line printer, 1–529
- processing matching text, 1–36
- removing, 1–580
- renaming, 1–446
- reversing lines, 1–577

file (cont.)

- sending to remote host, 1–732
- sorting, 1–641
- specifying line width, 1–239
- transferring, 1–251
- transferring remote, 1–698
- updating, 1–402
- updating date, 1–707
- xcpp file, 1–157

file command (general), 1–233

file command (mail), 1–396

- *See also* folder command (mail)

file command (pdx), 1–507

file name

- stripping affixes, 1–39

file transfer program

- ftp program, 1–251

find command, 1–234

- *See also* test command

finger command, 1–236

- and the who command, 1–236
- options, 1–236
- restricted, 1–236

fix command (sccs), 1–600

fmt text formatter, 1–238

- *See also* pr command

fold command, 1–239

- *See also* expand command

folder command, 1–240

folder command (mail), 1–397

folders

- removing, 1–584

folders command, 1–243

folders command (mail), 1–397

footnote

- formatting, 1–567

for command (sh), 1–610

for command (System V), 1–617

foreach command (csh), 1–130

fork

- reporting, 1–755

form command (ftp), 1–252

Fortran program

- breaking into separate files, 1–250

forw command, 1–245
Forwarding messages, 1–245
from command (mail), 1–249, 1–397
fsf command (mt), 1–445
fsplit program, 1–250
fsr command (mt), 1–445
ftp program, 1–251
See also rcp command
command list, 1–251
file-naming conventions, 1–257
options, 1–258
parameters supported, 1–257
restricted, 1–258
f77 compiler
See also ctags command
See also gprof command
See also prof command
f77 program
displaying call graph profile data and, 1–271

G

gcore command, 1–260
gencat utility, 1–261
get command (ftp), 1–252
get command (sccs), 1–263 to 1–268, 1–599
See also delta command (sccs)
See also rmdel command (sccs)
See also unget command (sccs)
See also what command (sccs)
auxiliary file list, 1–267
identification keywords, 1–266
options, 1–263
restricted, 1–267
get command (tftp), 1–698
getopt command, 1–269
glob command (csh), 1–130
glob command (ftp), 1–252
goto command (csh), 1–130
gprof command, 1–271
options, 1–271
restricted, 1–272
graph
drawing, 1–274
graph command, 1–274
See also plot command
See also spline command
options, 1–274
restricted, 1–275
graphics filter, 1–523
grep command, 1–276
See also cut command
See also look command
See also sed stream editor
diagnostics, 1–277
options, 1–277
restricted, 1–277
gripe command (pdx), 1–507
group
displaying memberships, 1–278
group ID
changing, 1–78
groups command, 1–278

H

hard link
defined, 1–362
hash command (ftp), 1–252
hash command (System V), 1–624
hashstat command (csh), 1–130
head command
See also tail command, 1–279
headers command (mail), 1–397
help command (mail), 1–397
help command (pdx), 1–507
history command (csh), 1–130
hold command (mail), 1–397
host
printing, 1–281
host ID
See also host name
printing in hexadecimal, 1–280
setting in hexadecimal, 1–280
host name
See also host ID
setting, 1–281

hostid command, 1–280
hostname command, 1–281

I

ic utility, 1–282
id command, 1–286
if command (csh), 1–130
if command (sh), 1–610
if command (System V), 1–618
ifdef macro, 1–390
ifelse macro, 1–391
ignore command (dbx), 1–172
ignore command (mail), 1–397
inc command, 1–287
include macro, 1–391
Incorporating mail, 1–287
incr macro, 1–391
indent command
    comments recognized, 1–290
    diagnostics, 1–291
    multiline expressions and, 1–290
    options, 1–289, 1–289 to 1–291
    restricted, 1–291
    setting default formatting, 1–290
index command, 1–549
index macro, 1–391
indxbib command, 1–369
info command (sccs), 1–599
install command, 1–292
Internet File Transfer Protocol interface
    ftp program, 1–251
interprocess communication package
    reporting status, 1–297
intro(1) keyword, 1–1
invcutter command, 1–293
I/O statistics
    See also disk
    See also terminal
    reporting, 1–295
iostat command
    See also netstat command
    See also vmstat command, 1–295
ipcrm command, 1–296
ipcs command
    column headings listed, 1–297
    options, 1–297, 1–297

J

jobs command (csh), 1–131
join command
    comm command, 1–300, 1–300
    restricted, 1–301
    sort command, 1–300

K

kill command (csh), 1–131
kill command (general), 1–303
    restricted, 1–303
kits
    setld format distribution kits, 1–304
    updating master inventory, 1–464

L

last command
    See also lastcomm command, 1–336
lastcomm command, 1–337
    See also last command
lcd command (ftp), 1–252
ld command, 1–347
    See also lk command
    See also ranlib command
    See also strip command
    options, 1–347
    restricted, 1–349
leave command, 1–350
len macro, 1–391
lex program generator, 1–351
    options, 1–351
lexical analysis program
    example, 1–351
    generating, 1–351
library file
    archive file, 1–27

limit command (csh), 1–131
line command, 1–352
line feed
reversing, 1–85
link
creating, 1–362
defined, 1–362
link editor (general)
See ld command
link editor (VAX FORTRAN)
See lk command
lint command, 1–356
exit system call and, 1–357
options, 1–356
list command (pdx), 1–507
Listing formatted messages, 1–425
lk command, 1–359
See also ranlib command
See also strip command
options, 1–359
restricted, 1–361
ln command, 1–362
options, 1–362
Location Broker
lb_admin, 1–338
lock
command, 1–363
logging in
See also password, 1–364
to remote system, 1–578, 1–702
login
printing last, 1–336
login command (csh), 1–132
login command (general)
dgate command, 1–190
diagnostics, 1–365, 1–364
login command (sh), 1–614
login shell field
changing, 1–82
login time
showing, 1–768
logname command, 1–367
logout command (csh), 1–132
look command, 1–368
lookbib command, 1–369
lorder command, 1–370
See also ranlib command
See also tsort command
lp command, 1–371
lpq command, 1–372
lpr command, 1–374
See also lpq command
See also lprm command
See also print command (general)
lprm command, 1–379
diagnostics, 1–380
restricted, 1–379
lpstat command, 1–381
ls command (ftp), 1–252
ls command (general), 1–382
options, 1–382
restricted, 1–383
ltf command, 1–384
diagnostics, 1–387
keys, 1–384
options, 1–384, 1–387

M

m4 macro processor
macro list, 1–390, 1–389
macdef command (ftp), 1–253
machine command, 1–393
magnetic tape
labeling, 1–384
manipulating, 1–444
mail
creating a distribution list, 1–395
deleting, 1–394
ending a session, 1–395
formatting, 1–238
listing header lines in mailbox file, 1–249
printing, 1–394, 1–531
processing for sendmail daemon, 1–582
reading, 1–394
replying to, 1–395
reporting incoming, 1–44

mail (cont.)
sending, 1–45, 1–394, 1–401
sending binary file, 1–729
specifying messages, 1–394
undeleting, 1–394
mail aliases
listing, 1–19
mail command (mail), 1–397
mail program, 1–401
See also biff command
See also fmt text formatter
See also from command (mail)
See also prmail command
See also talk program
See also uuencode command
See also write command
command list, 1–396
flags, 1–395, 1–394
tilde escapes, 1–399
make command, 1–402
make command (System V)
options, 1–402
make keyword, 1–402
maketemp macro, 1–391
man command, 1–409
See also apropos command
See also man macro package
See also ul command
See also whatis command (general)
options, 1–410
mark command, 1–415
mdelete command (ftp), 1–253
mdir command (ftp), 1–253
mdtar command
See also tar command
diagnostics, 1–420
function modifiers, 1–418
key list, 1–418, 1–418
restricted, 1–420
memory
reporting statistics, 1–754
mesg command, 1–421
See also talk program
message
copying to another user, 1–772
interactive, 1–678
prohibiting, 1–421
replying to, 1–572
show next message, 1–466
show previous message, 1–528
message queue
removing, 1–296
reporting status, 1–297
Message sequences, 1–415
messages
check for, 1–441
filing in other folders, 1–570
select by content, 1–515
mget command (ftp), 1–253
MH overview, 1–422
mh summary, 1–422
mhl command, 1–425
mhmail command, 1–429
mhpath command, 1–430
mkdir command, 1–432
See also rmdir command
mkdir command (ftp), 1–253
mkstr command, 1–434
See also xstr command
mls command (ftp), 1–253
mode
changing, 1–79
mode command (ftp), 1–253
mode command (TELNET), 1–689
mode command (tftp), 1–698
more command, 1–438 to 1–440
options, 1–438
moving sparse data files, 1–596
mput command (ftp), 1–253
msgchk command, 1–441
msh command, 1–442
mt program, 1–444, 1–445e
command list, 1–444
mv command, 1–446

N

name
defined, 1–617
nawk utility
arrays, 1–448
built-in functions, 1–452, 1–453
described, 1–447
restrictions, 1–457
statement list, 1–455
user-defined functions, 1–455
netstat command, 1–461
See also iostat command
See also vmstat command
options, 1–462
network
displaying status, 1–461
interface display, 1–461
routing table display, 1–461
Network Interface Definition Language Compiler
nidl, 1–468
newaliases command, 1–463
newinv command, 1–464
next command, 1–466
next command (dbx), 1–173
next command (mail), 1–397
next command (pdx), 1–506
nice command (csh), 1–132
nice command (sh), 1–467
nl command, 1–471
nm command
diagnostics, 1–475
options, 1–475, 1–475
nmap command (ftp), 1–253
nocache command (mt), 1–445
nohup command (csh), 1–132
nohup command (sh), 1–467
notify command (csh), 1–132
nroff text processor
See also checknr command
See also colcrt command
See also roffbib text processor
See also soelim command
See also tbl preprocessor
nroff text processor (cont.)
options, 1–477
previewing output, 1–86
refer preprocessor, 1–567, 1–477
nslookup command, 1–479
nsquery command, 1–485
ntp command, 1–487
sample output, 1–488
ntrans command (ftp), 1–254

O

object file
combining, 1–347, 1–359
finding printable strings, 1–656
ordering, 1–370, 1–718
printing size, 1–634
od command
See also strings command
options, 1–490, 1–490
offline command (mt), 1–445
onintr command (csh), 1–132
online information
accessing, 1–23
open command (ftp), 1–254
open command (TELNET), 1–689
otalk program, 1–678

P

pack command, 1–494
packf command, 1–496
page
reporting statistics, 1–755
page command, 1–438
page size
printing, 1–497
pagesize command, 1–497
parameter
defined, 1–611, 1–617
Pascal compiler
error recovery, 1–232, 1–503
Pascal execution profiler
See pxp command

Pascal interpreter
See px command
Pascal interpreter and executer
See pix command
Pascal interpreter code translator
See pi code translator
See pix command
Pascal program
creating line-numbered listing, 1–555
debugging, 1–505
displaying call graph profile data and, 1–271
interpreting, 1–513, 1–519, 1–552
listing cross-references, 1–555
merging compiled modules, 1–525
profiling, 1–553
passive verb
finding, 1–670
passwd command, 1–498
See also yppasswd command, 1–498
passwd file (general)
user name and, 1–77
password
changing, 1–498
changing in yellow pages, 1–783
creating, 1–498
printing with catpw, 1–55
paste command
diagnostics, 1–502
examples, 1–501
options list, 1–501, 1–501
pattern
matching, 1–612, 1–620
pc compiler
See also ctags command
See also gprof command
See also ld command
See also make command (general)
See also pdx debugger
See also pi code translator
See also pix command
See also pmerge command
See also prof command
See also px command
See also pxp command
pc compiler (cont.)
See also pxref program
options, 1–503, 1–503
restricted, 1–504
pdx debugger, 1–505
See also pi code translator
instructor-level commands, 1–507
option, 1–508
restricted, 1–508
pg command, 1–509
pi code translator, 1–513
See also pix command
See also pmerge command
See also px command
diagnostics, 1–514
flags, 1–513
restricted, 1–513
pi command (pdx), 1–507
pick command, 1–515
pipeline
defined, 1–610, 1–617
lists, 1–610
pipelines
lists, 1–617
pix command, 1–519
See also px command
plot command, 1–523
See also prof command
See also spline command
See also term command
pmerge command, 1–525
popd command (csh), 1–132
pr command
See also print command (general)
preserve command (mail), 1–397
See also hold command (mail)
prev command, 1–528
print command, 1–526
print command (general), 1–529
print command (mail), 1–394, 1–397
See also ignore command (mail)
See also print command (mail)
print command (pdx), 1–506

print queue
removing jobs, 1–379
printenv command, 1–530
printer
See also printer queue
changing tabs to blanks for, 1–225
folding text lines for, 1–239
status information, 1–381
printer queue
displaying, 1–372
priority
setting low, 1–467
prmail command, 1–531
process
getting core image, 1–260
printing status, 1–544, 1–756, 1–768
reporting statistics, 1–754
suspending, 1–635
terminating, 1–303
process ID
getting, 1–544
prof command, 1–532, 1–536
See also gprof command
options, 1–536
restricted, 1–536
profile data
analyzing, 1–532
profile file
displaying data, 1–536
program
executing later, 1–34
locating binary, 1–765
locating manual, 1–765
locating source, 1–765
updating, 1–402
prompt command (ftp), 1–254
prompter editor front-end, 1–538
proxy command (ftp), 1–254
prs command, 1–543e
prs command (sccs), 1–541
options, 1–541
ps command
See also w command
field list, 1–545
ps command (cont.)
options, 1–544, 1–544
restricted, 1–545
ptx command
options, 1–549, 1–549
restricted, 1–550
pushd command (csh), 1–132
put command (ftp), 1–255
put command (tftp), 1–699
pwd command (general)
See also dirs command (csh), 1–551
pwd command (nfs), 1–255
pwd command (System V), 1–624
px command, 1–552
See also pi code translator
See also pix command
pxp command, 1–553
options, 1–553
restricted, 1–554
pxref program, 1–555

Q

quit command (mail), 1–397
quit command (nfs), 1–255
quit command (pdx), 1–507
quit command (TELNET), 1–689
quit command (tftp), 1–699
quota command, 1–556
quote command (ftp), 1–255

R

ranlib command, 1–557, 1–558
See also lorder command
rcp command, 1–559
rcvstore command, 1–561
read command (sh), 1–614
read command (System V), 1–624
readability
analyzing, 1–670
readonly command (sh), 1–614
readonly command (System V), 1–624

recv command (ftp), 1–255
red line editor, 1–210
redistributing messages, 1–201
refer preprocessor, 1–567
See also indxbib command
See also lookbib command
addbib program, 1–567
lookbib command, 1–567
options, 1–567
restricted, 1–568
roffbib text processor, 1–567
sortbib command, 1–567
Reference Pages Manual
accessing on line, 1–409
printing, 1–409
refile command, 1–570
rehash command (csh), 1–133
relational data base operator, 1–300
relocation bits
removing, 1–658
reminder service
creating a calendar, 1–50
reminding you to leave, 1–350
remote system
logging in, 1–190
remotehelp command (ftp), 1–255
rename command (ftp), 1–255
repeat command (csh), 1–133
repl command, 1–572
reply command (mail), 1–398
rerun command (dbx), 1–171
reset command, 1–576
See also tset command
reset command (ftp), 1–255
respond command (mail), 1–398
See also reply command (mail)
return code
exit status, 1–1
return command (dbx), 1–173
return command (System V), 1–624
rev command, 1–577
rewind command (mt), 1–445
rewolffl command (mt), 1–445
rexmt command (tftp), 1–699
rlogin command, 1–578
See also dgate command
See also rcp command
See also tip command
rlogin command (general)
See also rlogin command
rm command, 1–580
confirming file removal, 1–580e
examining files, 1–581e
options, 1–580
removing file, 1–580e
rmail command, 1–582
rmdel command (sccs), 1–583
rmdir command (ftp), 1–255
rmdir command (general), 1–580
rmf command, 1–584
rmm command, 1–585
roffbib text processor, 1–586
rsh program, 1–588
See also rcp command
See also rlogin command
options, 1–588
restricted, 1–588
rsh5 program, 1–617
restricted, 1–626
run command (dbx), 1–171
run command (pdx), 1–505
run queue
showing average, 1–768
runique command (ftp), 1–255
ruptime command
description, 1–590
options, 1–590
restrictions, 1–590

S

sact command (sccs), 1–593
save command (mail), 1–398
scan command, 1–594
scat command, 1–596
SCCS file
changing delta commentary, 1–69

SCCS file (cont.)
changing parameters, 1–14 to 1–18
comparing, 1–601
creating, 1–14 to 1–18
data keywords, 1–541
getting, 1–263 to 1–268
identifying, 1–761
printing, 1–541, 1–593
reconstructing, 1–88
recording changes, 1–597
removing delta, 1–583
ungetting, 1–723
validating, 1–737
version control, 1–739
SCCS identification string
See SID
sccs preprocessor, 1–597
See also get command (sccs)
See also SCCS file
See also sccshelp command
changing file, 1–184 to 1–186
command list, 1–599
keywords, 1–600
sccsdiff command, 1–601
sccsdiff command (sccs), 1–600
sccshelp command, 1–600, 1–602
script command, 1–603
sed command, 1–604
semaphore set
removing, 1–296
reporting status, 1–297
send command, 1–607
send command (ftp), 1–255
send command (TELNET), 1–690
sendport command (ftp), 1–256
set command (csh), 1–133
set command (mail), 1–398
See also unset command (mail)
options, 1–400
set command (sh), 1–614
set command (System V), 1–624
set command (TELNET), 1–691
setenv command (csh), 1–133
setld
format distribution kits, 1–304
newinv command, 1–464
Setting current folder, 1–240
sh command (pdx), 1–507
sh command interpreter, 1–610, 1–616
See also echo command
See also false command
See also wait command (general)
command substitution, 1–611
directing input, 1–612
directing output, 1–612
environment, 1–613
executing commands, 1–614
parameter substitution, 1–611
prompts, 1–612
quoting characters, 1–612
signals, 1–614
special commands, 1–614
shared memory
reporting status, 1–297
shared memory ID
removing, 1–296
shell command (mail), 1–398
shell command interpreter
diagnostics, 1–616
restricted, 1–616
shexp command, 1–628
shift command (csh), 1–133
shift command (sh), 1–615
shift command (System V), 1–625
shift macro, 1–391
show command, 1–630
sh5 command interpreter, 1–617 to 1–627
command substitution, 1–618
comments, 1–618
directing input, 1–621
directing output, 1–621
environment, 1–622
executing commands, 1–622
exit status, 1–626
invoking, 1–625
parameter substitution, 1–618
prompts, 1–621

sh5 command interpreter (cont.)
restricted, 1–626
signals, 1–622
special characters and, 1–621
SID
defined, 1–598
simple command
defined, 1–610, 1–617
sinclude macro, 1–391
size command (general), 1–634
size command (mail), 1–398
sleep command, 1–635
slocal command, 1–636
SMP
reporting CPU statistics, 1–114
soelim command, 1–640
Software kits
producing, 1–684
producing inventory records for, 1–293
sort command, 1–641, 1–642e
See also look command
See also uniq command
diagnostics, 1–642
options, 1–641
restricted, 1–642
sortbib command, 1–646
sortm command, 1–647
sort5 command, 1–643
source code control system preprocessor
See sccs preprocessor
source command (csh), 1–133
source command (mail), 1–398
source command (pdx), 1–507
sparse data files, 1–596
spell command, 1–649
options, 1–649
restricted, 1–650
spellin command, 1–649
spellout command, 1–649
spline command, 1–651
split command, 1–652
status command (dbx), 1–172
status command (ftp), 1–256
status command (mt), 1–445
status command (pdx), 1–506
status command (TELNET), 1–689
status command (tftp), 1–699
step command (dbx), 1–172
step command (pdx), 1–506
stop command (csh), 1–134
stop command (dbx), 1–172
stop command (pdx), 1–506
stopi command (pdx), 1–507
stream text editor, 1–604
strextract utility, 1–654
string
defined, 1–656
strings command, 1–656
strip command, 1–658
strmerge utility, 1–659
struct command (ftp), 1–256
stty command, 1–662
See also tset command
See also tty command
style program, 1–670
See also diction program
su command, 1–671
substr macro, 1–391
sum command, 1–673
See also wc command
sunique command (ftp), 1–256
superblock
updating, 1–675
suspend command (csh), 1–134
switch command (csh), 1–134
symbol table
printing, 1–475
removing, 1–658
updating, 1–674
symbol type
reference list, 1–475
symbolic link, 1–362
symorder command, 1–674
sync command, 1–675
syscmd macro, 1–391
system
See also host ID

system (cont.)
See also host name
changing user information, 1–77
listing user information, 1–236
reporting statistics, 1–754
showing login time, 1–756
showing run queue average, 1–725, 1–756
showing uptime, 1–725, 1–756, 1–768
showing user activity, 1–756, 1–768
showing users, 1–726, 1–756, 1–768
system call tracer, 1–709

T

tab character
changing to spaces, 1–225
table
formatting, 1–686
tabs command
See also term command, 1–676
tags file
See ctags command
tail command, 1–677
talk program, 1–678
See also mesg command
See also write command
tar command
See also ar program
See also mdtar command
diagnostics, 1–683
keys, 1–680
options, 1–680 to 1–682, 1–680 to 1–683, 1–682e
restricted, 1–683
tarsets command, 1–684
tbl preprocessor
eqn and, 1–687, 1–686
tee command, 1–688
tell command (sccs), 1–599
TELNET protocol
See telnet user interface
telnet user interface, 1–689
command list, 1–689
terminal
capturing session in a file, 1–603
terminal (cont.)
clearing screen, 1–83
getting pathname, 1–719
locking, 1–363
reporting I/O statistics, 1–295
resetting, 1–576
setting, 1–714 to 1–717
setting tabs, 1–676
showing name, 1–756, 1–768
underlining and, 1–721
viewing one screenful at a time, 1–438 to 1–440
Terminals
setting input/output characteristics, 1–662
terminfo compiler
tic, 1–700
test command, 1–694
See also find command
command programming language, 1–696
test command (System V), 1–625
text processor
for monospace output, 1–477
tftp program, 1–698
authentication and, 1–699
tic
terminfo compiler, 1–700
time
setting, 1–158
showing, 1–756, 1–768
time command, 1–701
printing, 1–158
time command (csh), 1–134
timeout command (tftp), 1–699
times command (sh), 1–615
times command (System V), 1–625
tip command, 1–702
See also rlogin command
tilde escapes, 1–702
variables, 1–704, 1–706
toggle command (TELNET), 1–691
top command (mail), 1–398
touch command, 1–707
tr command, 1–708
trace command (dbx), 1–171

trace command (ftp), 1–256
trace command (general), 1–709
trace command (pdx), 1–505
trace command (tftp), 1–699
tracei command (pdx), 1–507
trans utility, 1–711
translit macro, 1–391
trap command (sh), 1–615
trap command (System V), 1–625
Trivial File Transfer Protocol
tftp program, 1–698
user interface, 1–698
true command, 1–713
tset command, 1–714 to 1–717, 1–716
See also term command
options, 1–715
restricted, 1–717
tsort command, 1–718
tty command, 1–719
type command (ftp), 1–256
type command (mail), 1–398
See also print command (mail)
type command (System V), 1–625
typescript file
creating, 1–603

U

uac command, 1–720
ul command, 1–721
ulimit command (System V), 1–625
umask command (csh), 1–134
umask command (sh), 1–615
umask command (System V), 1–625
unalias command (csh), 1–134
unalias command (mail), 1–398
uncompact command, 1–94
undefine macro, 1–390
undelete command (mail), 1–398
undivert macro, 1–390
unedit command (sccs), 1–599
unexpand command, 1–225
unget command (sccs), 1–723
unhash command (csh), 1–134
uniq command, 1–724
See also cmp command
See also comm command
See also diff command
See also diff3 command
See also diffmk command
See also join command
See also sccsdiff command
Universal Unique Identifiers
uuid_gen, 1–730
unlimit command (csh), 1–134
unset command (csh), 1–135
unset command (mail), 1–398
unset command (System V), 1–625
unsetenv command (csh), 1–135
uptime command, 1–725
See also w command
user command (ftp), 1–256
user ID
changing temporarily, 1–671
showing, 1–768
showing effective, 1–769
users command, 1–726
See also finger command
See also who command
uucp utility, 1–727
See also rmail command
See also uusend command
See also uustat program
displaying command status, 1–733
displaying connection status, 1–733
options, 1–727
remote system pathnames and, 1–728w
restricted, 1–728
uudecode command, 1–729
uuencode command, 1–729
uusend command, 1–732
uustat program
options, 1–733
uux command, 1–735

V

val command (sccs), 1–737
  interpreting 8-bit exit code, 1–737
  keyletters, 1–737
  processing multiple files, 1–738
  restricted, 1–738
VAX C
  vcc compiler, 1–742
VAX-11 assembler
  See as assembler
vc command (sccs), 1–739
  exit codes, 1–741
  options, 1–739
vcc compiler, 1–742
  default macros, 1–745
  default symbols, 1–745
  files, 1–746
  options, 1–742
  restricted, 1–746
vdoc command, 1–747
verbose command (ftp), 1–256
verbose command (tftp), 1–699
version control statement, 1–739
vfork
  reporting, 1–755
vi (screen) editor, 1–749
vi screen editor
  See also fmt text formatter
view command, 1–752
  encryption, 1–752
virtual memory
  reporting statistics, 1–754
visual command (mail), 1–398
vmstat command, 1–754
  See also iostat command
  See also netstat command
  format fields, 1–754

W

w command, 1–756
  options, 1–756
  output fields, 1–756
w command (cont.)
  restricted, 1–756
wait command (csh), 1–135
wait command (general), 1–758
wait command (sh), 1–615
wait command (System V), 1–625
wall command, 1–759
  See also mesg command
  See also write command (general)
wc command, 1–760
  See also sum command
weof command (mt), 1–444
what command (sccs), 1–599, 1–761
whatis command (general), 1–762
whatis command (pdx), 1–506
whatnow command, 1–763
where command (pdx), 1–507
whereis command, 1–765
  example, 1–765
which command (csh), 1–767
which command (pdx), 1–506
while command (csh), 1–135
while command (sh), 1–611
while command (SystemV), 1–618
who command, 1–768
  See also finger command
  See also users command
  See also whoami command
whoami command, 1–769
working directory
  changing, 1–68
  printing pathname, 1–551
write command
  See also mesg command
write command (general), 1–772
  See also talk program
  See also wall command
write command (mail), 1–398
  See also save command (mail)

X

xargs command, 1–773
xd command (pdx), 1–508
xi command (pdx), 1–508
xit command (mail), 1–399
See also exit command (mail)
xsend command
secret mail, 1–776
xstr command, 1–777
See also mkstr command
restricted, 1–778

Y

yacc compiler, 1–779
See also eyacc compiler
See also lex program generator
yellow pages service
changing password in, 1–783
yes command, 1–780
YP map
printing key values, 1–782
printing values, 1–781
YP server
determining, 1–785
ypcat command, 1–781
ypmatch command, 1–782
yppasswd command, 1–783
ypwhich command, 1–785

Z

z command (mail), 1–399
z command (TELNET), 1–689

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-baud modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital Subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal*	————	SSB Order Processing - WMO/E15 *or* Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

* For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

ULTRIX
Reference Pages Section 1: Commands A - L
AA-PC0WA-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? ______________________________

What do you like best about this manual? ______________________________

What do you like least about this manual? ______________________________

Please list errors you have found in this manual:

Page ______ Description ______________________________

Additional comments or suggestions to improve this manual:

____

What version of the software described by this manual are you using? ____________

Name/Title ______________________ Dept. __________

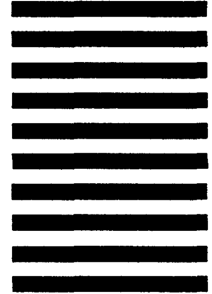
Company ______________________ Date __________

Mailing Address ______________________________

______________ Email ______________ Phone __________

------ Do Not Tear – Fold Here and Tape ------

digital™



BUSINESS REPLY MAIL
FIRST–CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062–9987

------ Do Not Tear – Fold Here ------

Cut
Along
Dotted
Line

Reader's Comments

ULTRIX
Reference Pages Section 1: Commands A - L
AA-PC0WA-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? ______________________

What do you like best about this manual? ______________________

What do you like least about this manual? ______________________

Please list errors you have found in this manual:

Page ______ Description ______________________

Additional comments or suggestions to improve this manual:

What version of the software described by this manual are you using? __________

Name/Title ______________________ Dept. __________

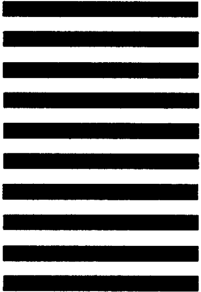
Company ______________________ Date __________

Mailing Address ______________________

______________________ Email ______________________ Phone __________

Do Not Tear – Fold Here and Tape

digital™



BUSINESS REPLY MAIL

FIRST–CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062–9987

Do Not Tear – Fold Here

Cut
Along
Dotted
Line