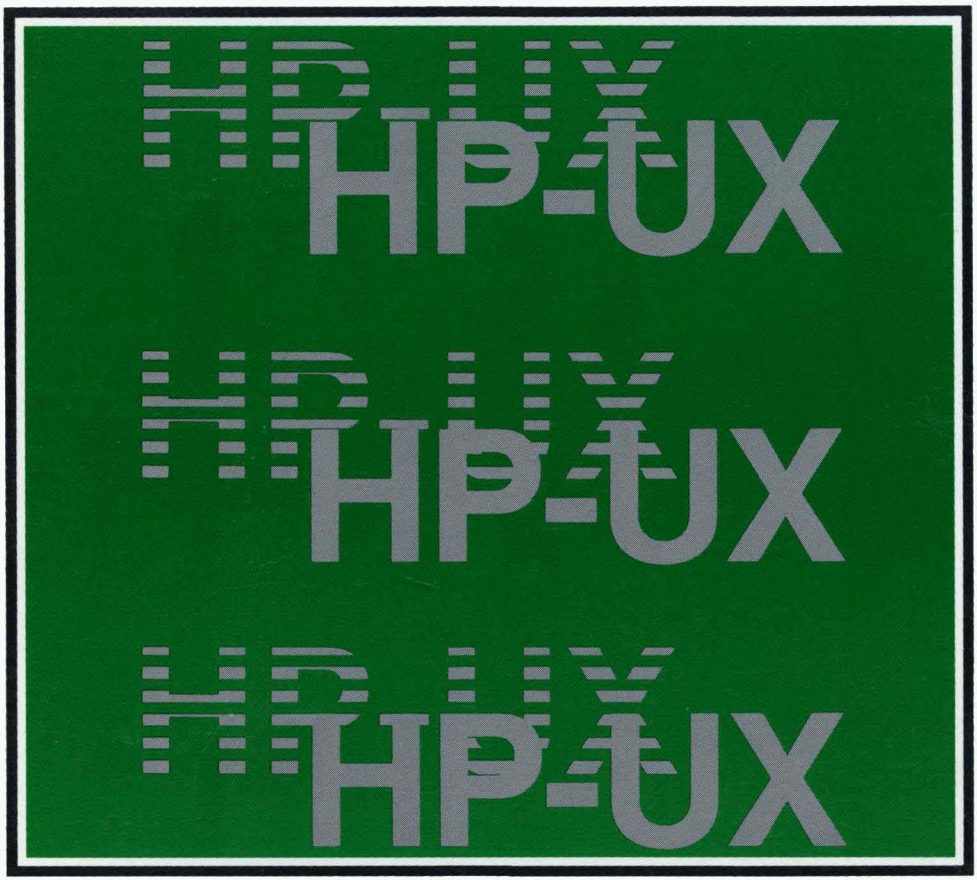
HP 9000 Series 200/500 Computers

HEWLETT PACKARD

HP-UX Reference
Vol. 1: Section 1



HP-UX Reference
Vol. 1: Section 1
for the HP 9000 Series 200/500 Computers

Manual Part No. 09000-90007

Hewlett-Packard Company
3404 East Harmony Road, Fort Collins, Colorado 80525

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

July 1984...Edition 1. This manual replaces HP-UX Reference Manual, 09000-90006

March 1985...Edition 1 with update

TABLE OF CONTENTS

1. Commands

2. System Calls

3. Subroutines

4. Special Files

5. File Formats

6. Games

No games are currently supported.

7. Miscellaneous Facilities

8. System Maintenance Utilities

Introduction

This manual describes the features of HP-UX in an alphabetical reference format. It is written for the user who is already familiar with UNIX or UNIX-like operating systems (UNIX is a trademark of Bell Telephone Laboratories, Inc.). The manual is intended for referencing specific details concerning the HP-UX operating system.

For a general overview of HP-UX, see the supplied tutorial text entitled *Introducing the UNIX System.* For details of the implementation and maintenance of the system, see the *HP-UX System Administrator Manual.*

This manual is divided into nine sections, some containing sub-classes that are interspersed throughout the section:

1. Commands and Application Programs:
   - 1. General-Purpose Commands.
   - 1C. Communications Commands.
   - 1G. Graphics Commands.
2. System Calls.
3. Subroutines:
   - 3C. C Library Routines.
   - 3M. Mathematical Library Routines.
   - 3S. Standard I/O Library Routines.
   - 3X. Miscellaneous Routines.
4. Special Files.
5. File Formats.
6. Games (none are currently implemented).
7. Miscellaneous Facilities.
8. System Maintenance Procedures.
9. Glossary.

Section 1 (*Commands and Application Programs*) describes programs intended to be invoked directly by the user or by command language procedures, as opposed to system calls (section 2) or subroutines (section 3), which are intended to be called by the user's programs. Commands generally reside in the directory **/bin** (for **bin**ary programs). Some programs also reside in **/usr/bin**, to save space in **/bin**, and to reduce search time for commonly-used commands. These directories are normally searched automatically by the command interpreter called the shell (*sh*(1)). Sub-class 1C contains communication programs such as *cu*, *fget*, etc. These entries may differ from system to system. A few commands are also located in **/lib** and **/usr/lib**.

Section 2 (*System Calls*) describes the entries into the HP-UX kernel, including the C language interface.

Section 3 (*Subroutines*) describes the available subroutines. Their binary versions reside in various system libraries in the directories **/lib** and **/usr/lib**. See *intro*(3) for descriptions of these libraries and the files in which they are stored.

Section 4 (*Special Files*) discusses the characteristics of each special file (device driver) that actually refers to an input/output device. The names in this section generally refer to Hewlett-Packard's device names for the hardware, rather than to the names of the special files themselves.

Section 5 (*File Formats*) documents the structure of particular kinds of files. For example, the format of the output of the link editor is given in *a.out*(5). Excluded are files used by only one command (for example, the assembler's intermediate files). In general, the C language **struct** declarations corresponding to these formats can be found in the directories **/usr/include** and **/usr/include/sys**.

Section 6 (*Games*) describes the games and educational programs that, as a rule, reside in the directory **/usr/games**. This section may or may not exist, depending on whether or not games are supported in each implementation of HP-UX.

Section 7 (*Miscellaneous Facilities*) contains a variety of things. Included are descriptions of character sets, macro packages, etc.

Section 8 (*System Maintenance Procedures*) discusses those commands which are useful for crash recovery and booting the system, plus commands used to perform system integrity checks and other maintenance procedures. Information in this section is mostly of interest to the super-user.

Section 9 (*Glossary*) defines terms used in this manual.

Each section (except 9) consists of a number of independent entries of a page or so each. The name of the entry appears in the upper corners of its pages. Entries within each section are alphabetized, with the exception of the introductory entry that begins each section. The page number of each entry starts at 1. Some entries may describe several routines, commands, etc. In such cases, the entry appears only once, alphabetized under its "major" name.

All entries are based on a common format, not all of whose parts always appear:

NAME

gives the name(s) of the entry and briefly states its purpose.

SYNOPSIS

summarizes the use of the entry (program) being described. A few conventions are used:

Boldface strings are literals and are to be typed just as they appear.

Italic strings represent substitutable argument names and program names found elsewhere in the manual.

Square brackets [ ] around an argument name indicate that the argument is optional. When an argument name is given as "name" or "file", it always refers to a *file* name.

Ellipses (...) are used to show that the previous argument may be repeated.

A final convention is used by the commands themselves. An argument beginning with a dash (–), a plus sign (+), or an equal sign (=) is often taken to be some sort of flag argument, even if it appears in a position where a file name could appear. Therefore, it is unwise to have files whose names begin with –, +, or =.

HP-UX COMPATIBILITY
shows the entry's HP-UX level and its origin, according to the HP-UX Compatibility Model (see *HP-UX Compatibility Model* following this introduction). This section also shows whether an optional HP software package is required.

DESCRIPTION
discusses the function and behavior of each entry.

HARDWARE DEPENDENCIES
points out variations from HP-UX standard due to the specific hardware involved.

EXAMPLE(S)
gives example(s) of usage, where appropriate.

FILES
gives the file names that are built into the program.

RETURN VALUE
discusses the meaning of values which are returned by the program.

SEE ALSO
gives pointers to related information.

DIAGNOSTICS
discusses the diagnostic indications that may be produced. Messages that are intended to be self-explanatory are not listed.

WARNINGS
points out potential pitfalls.

BUGS
gives known bugs and sometimes deficiencies. Occasionally, the suggested fix is also described.

A table of contents and a permuted index precede Section 1. On each *index* line, the title of the entry to which that line refers is followed by the appropriate section number in parentheses. This is important because there is considerable duplication of names among the sections, arising principally from commands that exist only to exercise a particular system call.

How to Get Started

This discussion provides the basic information you need to get started on HP-UX: how to log in and log out, how to communicate through your machine, and how to run a program. (See the supplied tutorial text for a more complete introduction to the system.)

Logging in. To log in you must have a valid user name, which may be obtained from the system administrator of your system. Keep pressing the "break" or "del" key until the **login:** message appears.

When a connection has been established, the system types **login:** and you then type your user name followed by the "return" key (or "enter" key, on some terminals). If you have a password (and you should!), the system asks for it, but does not print it on the terminal.

It is important that you type your login name in lower case if possible; if you type upper-case letters, HP-UX assumes that your terminal cannot generate lower-case letters, and that all subsequent upper-case input is to be treated as lower case. When you have logged in successfully, the shell types a **$**. (The shell is described below under *How to run a program.*)

For more information, consult *login*(1) and *getty*(8), which discuss the login sequence in more detail, and *stty*(1), which tells you how to describe the characteristics of your terminal to the system (*profile*(5) explains how to accomplish this last task automatically every time you log in).

Logging out. You can log out by typing an end-of-file indication (ASCII **EOT** character, usually typed as "control-d") to the shell. The shell will terminate and the **login:** message will appear again.

How to communicate through your terminal. When you type to HP-UX, the system usually gathers your characters and saves them. These characters will not be given to a program until you type a "return" (or a "new-line").

HP-UX terminal input/output is full-duplex. It has full read-ahead, which means that you can type at any time, even while a program is printing on your display or terminal. Of course, if you type during output, the output will have the input characters interspersed in it. However, whatever you type will be saved and interpreted in the correct sequence. There is a limit to the amount of read-ahead, but it is generous and not likely to be exceeded unless the system is in trouble. When the read-ahead limit is exceeded, the system throws away *all* the saved characters.

On an input line from a terminal, the character @ "kills" all the characters typed before it. The character # erases the last character typed. Successive uses of # will erase characters back to, but not beyond, the beginning of the line; @ and # can be typed as themselves by preceding them with \ (thus, to erase a \, you need two #s). These default erase and kill characters can be changed, and usually are (see *stty*(1)).

The ASCII **DC3** (control-s) character can be used to temporarily stop output. It is useful with CRT terminals to prevent output from disappearing before it can be read. Output is resumed when any character is typed. If *DC1* (control-q) or *DC3* are used to restart the program, they are not saved and passed to later programs. Any other characters are saved and passed as input to later programs.

The ASCII **DEL** character (sometimes labelled "rubout" or "rub") is not passed to programs, but instead generates an *interrupt signal*, just like the "break", "interrupt", or "attention" signal. This signal generally causes whatever program you are running to terminate. It is typically used to stop a long printout that you don't want. However, programs can arrange either to ignore this signal altogether, or to be notified when it happens (instead of being terminated). The editor *ed*(1), for example, catches interrupts and stops what *it* is doing, instead of terminating, so that an interrupt can be used to halt an editor printout without losing the file being edited.

The *quit* signal is generated by typing the ASCII octal 34 (control-\) character. It causes a running program to terminate.

Besides adapting to the speed of the terminal, HP-UX tries to be intelligent as to whether you have a terminal with the "new-line" key, or whether it must be simulated with a "carriage-return" and "line-feed" pair. In the latter case, all *input* "carriage-return" characters are changed to "line-feed" characters (the standard line delimiter), and a "carriage-return" and "line-feed" pair is echoed to the terminal. If you get into the wrong mode, see *stty*(1).

Tab characters are used freely in HP-UX source programs. If your terminal does not have the tab function, you can arrange to have tab characters changed into spaces during output, and echoed as spaces during input (not currently supported on the Series 500). The *stty*(1) command will set or reset this mode. The system assumes that tabs are set every eight character positions. The *tabs*(1) command will set tab stops on your terminal, if that is possible.

How to run a program. When you have successfully logged into HP-UX, a program called the shell is listening to your terminal. The shell reads the lines you type, splits them into command names and arguments, and executes the command. A command is simply an executable program. Normally, the shell looks first in your current directory (see *The current directory* below) for a program with the given name, and if none is there, then in system directories. There is nothing special about system-provided commands except that they are kept in directories where the shell can find them. You can also keep commands in your own directories and arrange for the shell to find them there.

The command name is the first word on an input line to the shell; the command and its arguments are separated from one another by space and/or tab characters.

When a program terminates, the shell will ordinarily regain control and type a **$** at you to indicate that it is ready for another command. The shell has many other capabilities, which are described in detail in *sh*(1).

The current directory. HP-UX has a file system arranged in a hierarchy of directories. When the system administrator gave you a user name, he or she also created a directory for you (ordinarily with the same name as your user name, and known as your *login* or *home* directory). When you log in, that directory becomes your *current* or *working* directory, and any file name you type is assumed to be in that directory by default. Because you are the owner of this directory, you have full permissions to read, write, alter, or destroy its contents. The permissions you have in other directories and files will have been granted or denied to you by their respective owners, or by the system administrator. To change the current working directory use *cd*(1).

Path names. To refer to files not in the current directory, you must use a path name. Full path names begin with /, which is the name of the *root* directory of the whole file system. After the slash comes the name of each directory containing the next sub-directory (followed by a /), until finally the file name is reached (e.g., **/usr/ae/filex** refers to file **filex** in directory **ae**, while **ae** is itself a subdirectory of **usr**; **usr** springs directly from the root directory). See the glossary for a formal definition of *path name*.

If your current directory contains subdirectories, the path names of files therein begin with the name of the corresponding subdirectory (*without* a prefixed /). Without important exception, a path name may be used anywhere a file name is required.

Important commands that modify the contents of files are *cp*(1), *mv*(1), and *rm*(1), which respectively copy, move (i.e., rename), and remove files. To find out the status of files or directories, use *ls*(1). Use *mkdir*(1) for making directories and *rmdir*(1) for destroying them.

For a more complete discussion of the file system, see the references cited at the beginning of the *Introduction* above. It may also be useful to glance through Section 2 of this manual, which discusses system calls, even if you don't intend to deal with the system at that level.

Writing a program. To enter the text of a source program into an HP-UX file, use *ed*(1), *ex*(1), or *vi*(1). The three principal languages available under HP-UX are C (see *cc*(1)), Fortran (see *fc*(1) or *f77*(1)), and Pascal (see *pc*(1)). After the program text has been entered with the editor and written into a file (whose name has the appropriate suffix), you can give the name of that file to the appropriate language processor as an argument. Normally, the output of the language processor will be left in a file in the current directory named **a.out** (if that output is precious, use *mv*(1) to give it a less vulnerable name). If the program is written in assembly language, you will probably need to link library subroutines with it (see *ld*(1)). Fortran, C, and Pascal call the linker automatically.

When you have gone through this entire process without encountering any diagnostics, the resulting program can be run by giving its name to the shell in response to the **$** prompt.

Your programs can receive arguments from the command line just as system programs do by using the *argc*, *argv*, and *envp* parameters. See the supplied C tutorial for details.

Text processing. Almost all text is entered through the editors *ed*(1), *ex*(1), or *vi*(1). The commands most often used to write text on a terminal are *cat*(1) and *pr*(1). The *cat*(1) command simply dumps ASCII text on the terminal, with no processing at all. The *pr*(1) command paginates the text, supplies headings, and has a facility for multi-column output.

Surprises. Certain commands provide *inter-user* communication. Even if you do not plan to use them, it would be well to learn something about them, because someone else may direct them toward you. To communicate with another user currently logged in, *write*(1) is used; *mail*(1) will leave a message whose presence will be announced to another user when he or she next logs in. The corresponding entries in this manual also suggest how to respond to these two commands if you are their target.

When you log in, a message-of-the-day may greet you before the first **$** prompt.

HP-UX Compatibility Model

HP-UX is Bell System III plus "HP value added". HP value added includes both Hewlett-Packard capabilities, such as graphics, and features from other UNIX systems, such as those from University of California at Berkeley.

Levels

The various HP-UX systems are listed below in order of increasing completeness; each contains all the elements of the previous one.

HP-UX/RUN ONLY
This describes a run-only kernel with no commands or applications attached.

HP-UX/NUCLEUS
This is the run-only kernel plus a minimum set of commands. It also provides a minimum command interpreter to permit access to the commands.

HP-UX/DEVELOPMENT
This is the first "normal" UNIX, but it does not include the full UNIX command set.

HP-UX/STANDARD
This is a nearly complete UNIX. It includes most of the capabilities from Bell, but not everything that HP will make available.

HP-UX/EXTENDED
This is the largest standard package. It contains almost everything HP-UX has to offer (a few Bell capabilities are not included).

OPTIONAL
For the purposes of the model, there are also capabilities that are never required, even at the **HP-UX/EXTENDED** level. The term **OPTIONAL** designates capabilities in this category.

NON-STANDARD
This designation is given to those keywords which have either not yet been approved as part of standard HP-UX, or never will be.

NAME

intro – introduction to commands

DESCRIPTION

This section describes, in alphabetical order, publicly-accessible commands. 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.

NAME

adb – debugger

SYNOPSIS

adb [**–w**] [objfil [corfil]]

HP-UX COMPATIBILITY

Level: HP-UX/DEVELOPMENT

Origin: System III

Remarks: *Adb* is implemented on the Series 200 only.

DESCRIPTION

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

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

Requests to *adb* are read from the standard input and responses are to the standard output. If the **–w** flag is present then *objfil* is created if necessary and opened for reading and writing so that it can be modified using *adb*. *Adb* 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 will be executed. The default *count* is 1. *Address* and *count* are expressions.

The interpretation of an address depends on the context in which it is used. If a subprocess is being debugged then addresses are interpreted in the usual way in the address space of the subprocess. 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 An octal number if *integer* begins with a 0; a hexadecimal number if preceded by 0x; a decimal number if preceded by 0d; otherwise the base of *integer* is whatever the default number base for input is. This base is initialized to hexadecimal.

integer.fraction
A 32 bit floating point number.

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

< name The value of *name*, which is either a variable name or a register name. *Adb* maintains a number of variables (see *VARIABLES*) named by single letters or digits. If *name* is a register name then the value of the register is obtained from the system header in *corfil*. The register names are **a0 ... a6 d0 ... d7 sp pc ps**.

symbol A *symbol* is a sequence of upper or lower case letters, underscores or digits, not starting with a digit. The value of the *symbol* is taken from the symbol table in *objfil*. An initial _ will be inserted at the beginning of *symbol* if needed.

_symbol

In C, the "true name" of an external symbol begins with . It may be necessary to utter this name to distinguish it from a symbol generated in assembly language.

(*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.

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

e1 + e2 Integer addition.

e1–e2 Integer subtraction.

*e1 *e2* Integer multiplication.

e1 %e2 Integer division.

e1 &e2 Bitwise conjunction.

e1 | e2 Bitwise disjunction.

e1 #e2 *E1* rounded up to the next multiple of *e2*.

COMMANDS

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

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

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

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

A *format* consists of one or more characters that specify a style of printing. Each format character may be preceded by a decimal integer that is a repeat count for the format character. While stepping through a format *dot* is incremented by the amount given for each format letter. If no format is given then the last format is used. The format letters available are as follows:

o 2 Print 2 bytes in octal. All octal numbers output by *adb* are preceded by 0.
O 4 Print 4 bytes in octal.
q 2 Print 2 bytes in signed octal.
Q 4 Print 4 bytes in signed octal.
d 2 Print 2 bytes in decimal.
D 4 Print 4 bytes in decimal.
x 2 Print 2 bytes in hexadecimal.
X 4 Print 4 bytes in hexadecimal.
u 2 Print 2 bytes as an unsigned decimal number.
U 4 Print 4 bytes as an unsigned decimal number.
f 4 Print the 32 bit value as a floating point number.
F 8 Print double floating point.
b 1 Print the addressed byte in hexadecimal.
B 1 Print the addressed byte in octal.

c 1 Print the addressed character. (The sign bit is ignored.)

C 1 Print the addressed character using the following escape convention. Character values 000 to 040 are printed as @ followed by the corresponding character in the range 0100 to 0140. The character @ is printed as @@. (The sign bit is ignored.)

s *n* Print the addressed characters until a zero character is reached. *N* is the length of the string, including the zero terminator.

S *n* Print a string using the @ escape convention. *n* is the length of the string including its zero terminator.

Y 4 Print 4 bytes in date format (see *ctime*(3C)).

i *n* Print as MC68000 instructions. *n* is the number of bytes occupied by the instruction.

I *n* Same as **i**, except that immediate constants are printed in decimal.

a 0 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

p 4 Print the addressed value in symbolic form using the same rules for symbol lookup as **a**.

t 0 When preceded by an integer tabs to the next appropriate tab stop. For example, **8t** moves to the next 8-space tab stop.

r 0 Print a space.

n 0 Print a 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.

new-line

Repeat the previous command with a *count* of 1. Also can be used to repeat a **:s** or **:c** command.

[**?**/]**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 is called to read the rest of the line following **!**.

$*modifier*

Miscellaneous commands. The available *modifiers* are:

<*f* Read commands from the file *f* and return.

>*f* Send output to the file *f*, which is created if it does not exist.
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 **a6**). If *count* is given then only the first *count* frames are printed.
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 The default for all integers input is octal.
d The default for all integers input is decimal.
x The default for all integers input is hexadecimal.
q Exit from *adb*.
v Print all non zero variables in octal.
n Set the number of significant digits for floating point dump to *address*.
m Print the address map.

new-line
print the process id and register values.

:*modifier*
Manage a subprocess. Available modifiers are:

b*c* Set breakpoint at *address*. The breakpoint is executed *count*–1 times before causing a stop. Each time the breakpoint is encountered the command *c* is executed. If this command sets *dot* to zero then the breakpoint causes a stop.
d Delete breakpoint at *address*.
d* Delete all breakpoints
r Run *objfil* as a subprocess. If *address* is given explicitly then the program is entered at this point; otherwise the program is entered at its standard entry point. *count* specifies how many breakpoints are to be ignored before stopping. Arguments to the subprocess may be supplied on the same line as the command. An argument starting with < or > causes the standard input or output to be established for the command. All signals are turned on on entry to the subprocess.
e Setup a subprocess as in **r**; no instructions are executed.
c*s* The subprocess is continued with signal *s* (see *signal*(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.
S*s* As for **c** except that a temporary breakpoint is set at the next instruction. Useful for stepping across subroutines.
x *a* [*b*]...
Execute subroutine *a* with parameters *[b]*...
k The current subprocess, if any, is terminated.

VARIABLES

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

0 The last value printed.
1 The last offset part of an instruction source.
2 The previous value of variable 1.

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 (0x107, 0x108)
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 address<e1
file address = *address* + *f1*–*b1*

Otherwise,

b2 address<e2
file address = *address* + *f2*–*b2*

Otherwise, the requested *address* is not legal. 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; in this way the whole file can be examined with no address translation.

In order for *adb* to be used on large files, all appropriate values are kept as signed 32 bit integers.

FILES

/dev/mem
/dev/swap
a.out
core

SEE ALSO

a.out(5), core(5).

DIAGNOSTICS

"Adb" when there is no current command or format, and comments about inaccessible files, syntax errors, abnormal termination of commands, etc. Exit status is 0, unless last command failed or returned non-zero status.

NAME

admin – create and administer SCCS files

SYNOPSIS

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

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Admin is used to create new SCCS files and change parameters of existing ones. Arguments to *admin*, which may appear in any order, consist of keyletter arguments, which begin with –, and named files (note that SCCS file names must begin with the characters **s.**). If a named file doesn't 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 silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The keyletter arguments are as follows. Each is explained as though only one named file is to be processed since the effects of the arguments apply independently to each named file.

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

–**i**[*name*] The *name* of a file from which the text for a new SCCS file is to be taken. The text constitutes the first delta of the file (see –**r** keyletter for delta numbering scheme). If the **i** keyletter is used, but the file name is omitted, the text is obtained by reading the standard input until an end-of-file is encountered. If this keyletter is omitted, then the SCCS file is created with an empty initial delta. Only one SCCS file may be created by an *admin* command on which the **i** keyletter is supplied. Using a single *admin* to create two or more SCCS files requires that they be created empty (no –**i** keyletter). Note that the –**i** keyletter implies the –**n** keyletter.

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

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

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

i Causes the "No id keywords (cm7)" message issued by *get*(1) or *delta*(1) to be treated as a fatal error. In the absence of this flag, the message is only a warning. The message is issued if no SCCS identification keywords (see *get*(1)) are found in the text retrieved or stored in the SCCS file.

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. Omitting any list is equivalent to **a**.

n Causes *delta*(1) to create a "null" delta in each of those releases (if any) being skipped when a delta is made in a *new* release (e.g., in making delta 5.1 after delta 2.7, releases 3 and 4 are skipped). These null deltas serve as "anchor points" so that branch deltas may later be created from them. The absence of this flag causes skipped releases to be non-existent 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 text retrieved by *get*(1). If the **m** flag is not specified, the value assigned is the name of the SCCS file with the leading **s.** removed.

t*type* *Type* of module in the SCCS file substituted for all occurrences of the %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 (see *delta*(1)). (If this flag is set when creating an SCCS file, the **m** keyletter must also be used even if its value is null).

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

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

–a*login* A *login* name, or numerical HP-UX group ID, to be added to the list of users which may make deltas (changes) to the SCCS file. A group ID is equivalent to specifying all *login* names common to that group ID. Several **a** keyletters may be used on a single *admin* command line. As many *login*s, or numerical group IDs, as desired may be on the list simultaneously. If the list of users is empty,

then anyone may 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 may be used on a single *admin* command line.

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

date and time created *YY/MM/DD HH:MM:SS* by *login*

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

–m[*mrlist*] 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 will occur if the **v** flag is not set or *MR* validation fails.

–h Causes *admin* to check the structure of the SCCS file (see *sccsfile*(5)), 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.

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

–z The SCCS file 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.

FILES

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

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

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

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

SEE ALSO

delta(1), ed(1), get(1), help(1), prs(1), what(1), sccsfile(5).
SCCS User's Guide in *HP-UX Concepts and Tutorials.*

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

ar – archive and library maintainer

SYNOPSIS

ar key [posname] afile name ...

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

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

The archive file format is consistent across all HP-UX implementations. It is only useful to port source archives. Individual files are inserted without conversion into the archive file. Note that *ar* files from other UNIX systems are not readable on HP-UX, even if those files contain ASCII text.

Key must be present, and is one character from the set **drqtpmx**, optionally concatenated with one or more of **vuaibcl**. *Afile* is the archive file. The *names* are constituent files in the archive file. The meanings of the *key* characters for operations on an archive are:

d Delete the named constituents from the archive file.

r Replace the named files, or add a new file to the archive. If the optional character **u** is used with **r**, then only those files with modified dates later than the archive files are replaced. If an optional positioning character from the set **abi** is used, then the *posname* argument must be present and specifies that new copies of the named files are to be placed after (**a**) or before (**b** or **i**) *posname*. In the absence of a positioning character, new files are placed at the end. *Ar* will create *afile* if it does not already exist. If there are no file *name*s, *ar* may create an empty archive file whose only contents is the archive magic number (see *magic*(5)).

q Quickly append the named files to the end of the archive file. Optional positioning characters are invalid. The command does not check whether the added members are already in the archive. This is useful only to avoid quadratic behavior when creating a large archive piece-by-piece. *Ar* will create *afile* if it does not already exist.

t Print a table of contents of the archive file. If no names are given, all files in the archive are described. If names are given, information about only those files appears.

p Print the named files in the archive.

m Move 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**, specifies where the files are to be moved. Note that, when used with a positioning character, the files are moved *in the same order* that they currently appear in the archive, *not* in the order specified on the command line. See EXAMPLES.

x Extract the named files. If no names are given, all files in the archive are extracted. In neither case does **x** alter (i.e. delete entries from) the archive file.

The meanings of the remaining optional modifying characters are:

v Verbose. When used with **t**, it gives a long listing of all information about the files from the archive headers. When used with the **d**, **m**, **p**, **q**, and **x** options, the verbose option causes *ar* to print the key letter and file name associated with each file for that operation. For the **r** operation, *ar* will show an "a" if it added a new file, or an "r" if it replaced an existing one.

c Create. Normally *ar* will create *afile* when it needs to (for the **r** and **q** operations). The create option suppresses the normal message that is produced when *afile* is created.

l Local. Normally *ar* places its temporary files in the directory /**tmp**. This option causes them to be placed in the current working directory. Only the **d**, **m**, and **r** options use temporary files.

Only the following combinations are meaningful:

d: **v**, **l**
r: **u**, **v**, **c**, **l**, and **a** | **b** | **i**
q: **v**, **c**
t: **v**
p: **v**
m: **v**, **l**, and **a** | **b** | **i**
x: **v**

For other combinations of modifiers with operations not shown in the above table, the modifier has no effect.

EXAMPLES

The command

```
ar r newlib.a f3 f2 f1 f4
```

will create a new file (if one does not already exist) in archive format with its constituents entered in the order shown in the above command line.

If you want to replace files f2 and f3 such that the new copies follow file f1, the commands

```
ar ma f2 newlib.a f3
ar ra f1 newlib.a f2 f3
```

will produce the desired effect. The archive will now be ordered

```
newlib.a: f1 f2´ f3´ f4
```

where the single quote marks indicate updated files. The first command says "move f3 after f2 in newlib.a", thus creating the order

```
f2 f3 f1 f4
```

The second command above says "replace files f2 and f3 in newlib.a, and put the new copies after f1". Note that the new files must be replaced in the same order as they already occur in the archive. This is why the first command above is used to create a new order that will be preserved in the replace operation. Thus, the two commands above cannot be replaced by

```
ar ra f1 newlib.a f2 f3
```

because the previous order of f2 and f3 in the archive will be preserved (no matter how the files are specified on the command line), producing the following archive:

```
newlib.a: f1 f3´ f2´ f4
```

FILES

/tmp/v* temporary files

SEE ALSO

ld(1), lorder(1), ranlib(1), ar(5).

WARNING

If you are the super-user, *ar* will alter any archive file, even if it is write-protected.

BUGS

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

Ar reports *cannot create file.a*, where file.a is an *ar*-format archive file, even if file.a already exists. This message is triggered when file.a is write-protected or inaccessible.

NAME

as – assembler for MC68000

SYNOPSIS

as [**–A**] [**–a** afile] [**–o** objfile] [file]

HP-UX COMPATIBILITY

Level: HP-UX/DEVELOPPMENT

Origin: System III

Remarks: *As* is implemented on the Series 200 only.

DESCRIPTION

As assembles the named *file*, or the standard input if no file name is specified. The optional arguments **–A** or **–a** may be used to obtain an assembly listing with offsets and instruction codes listed in hex. If **–A** is used the listing goes to standard output. If **–a** is used the listing goes to *afile*.

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, *.s* is stripped from the end of the file name (if there) and *.o* is appended to it. This becomes the name of the output file. *As* does not invoke *ld*.

FILES

/usr/tmp/*	temporary files
file.o	object file

SEE ALSO

adb(1), ld(1), nm(1), a.out(5).
MC68000 Assembler on HP-UX, in *HP-UX Concepts and Tutorials.*

DIAGNOSTICS

If the name chosen for the output file is of the form ***.[cs]**, the assembler issues an appropriate complaint and quits. When syntactic or semantic errors occur, a single-line diagnostic is typed out together with the line number and the file name in which it occurred.

NAME

asa – interpret ASA carriage control characters

SYNOPSIS

asa [files]

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System V

Remarks: *Asa* is in a preliminary state, and could change in the future.

DESCRIPTION

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

blank	single new-line before printing;
0	double new-line before printing;
1	new page before printing;
+	overprint previous line.

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

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

```
a.out | asa | lpr
```

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

```
asa file
```

SEE ALSO

fc(1), f77(1).

NAME

at - execute commands at a later time

SYNOPSIS

at time [day] [file]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: Version 7

DESCRIPTION

At stores a copy of the named *file* (standard input default) to be used as input to *sh*(1) at a specified later time. *File* must be a shell script. A *cd*(1) 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 copied file.

Time is one to four digits, with an optionally appended 'A', 'P', 'N', or 'M', standing for AM, PM, noon, or midnight, respectively. One and two digit numbers are taken to be hours; three and four digit numbers are taken to be hours and minutes. If none of the above-mentioned letters follow the digits, a 24 hour clock time is assumed.

Day is either a month name followed by a day number, or a day of the week. If the word 'week' follows *day*, invocation is moved seven days further off. Three-letter abbreviations for month and day names may be used. Examples of legal commands are

```
at 8am jan 24 scriptfile1
at 1530 fri week scriptfile2
```

At programs are executed by periodic execution of the command */usr/lib/atrun* from *cron*(8). The time interval accuracy of *at* depends upon how often *atrun* is executed.

Standard output or error output is lost unless redirected.

FILES

/usr/spool/at/*yy.ddd.hhhh.uu*
: activity to be performed at hour *hhhh* of day *ddd* of year *yy*. *uu* is a unique number.

/usr/spool/at/lasttimedone
: contains *hhhh* for last hour that *atrun* was executed.

/usr/spool/at/past
: directory of activities now in progress.

/usr/lib/atrun
: program that executes activites that are due.

pwd(1)

SEE ALSO

calendar(1), cron(8).

DIAGNOSTICS

Complains about various syntax errors and times out of range.

BUGS

Due to the time interval accuracy of the execution of */usr/lib/atrun*, there may be bugs in scheduling things almost exactly 24 hours into the future.

NAME

aterm – general purpose asynchronous terminal emulation

SYNOPSIS

aterm configfile

HP-UX COMPATIBILITY

Level: Data Communications – HP-UX/STANDARD

Origin: HP

Remarks: *Aterm* is implemented on the Series 500 only.

DESCRIPTION

Aterm is a general purpose asynchronous terminal emulator designed for maximum connection flexibility and simple file transfers without remote host support. Transparent pass-through mode provides all user terminal capabilities in multi-user systems.

Configfile is used by *aterm* to match the particular terminal configuration needed for the remote system you are logging onto. This file consists of configuration commands, one to a line. Each line consists of the command name and its arguments, if any. Only configuration parameters which differ from the standard default need be specified. Most configuration commands can also be given from the keyboard while the emulator is running. You can exit *aterm* by typing "~.".

The following list shows the recognized configuration command names:

da Serial device file name (no default);

hn Name of remote computer system (no default);

db Number of data bits per character: 5, 6, 7, or 8 (default = 7);

sb Number of stop bits per character: 1, 1.5, or 2 (default = 1);

pa Character parity: none (n), odd (o), even (e), zero (0), or one (1) (default = o);

dr Rate for data sent and received: 50, 75, 110, 134.5, 150, 300, 600, 1200, 1800, 2400, 3600, 4800, 9600, or 19200 baud (default = 2400 baud);

mc Modem control: enabled (+) for full-duplex modem, or disabled (–) for full-duplex hard-wired connection (default = –);

ss Switched service: auto-answer (a) or manual originate (o) (default = o);

ga Gap: number of character transmission times to delay between successive output characters; values range from 0 to 254 (default = 0);

ec Echo: enabled (+) if the host computer echos characters sent by the emulator, disabled (–) otherwise (default = –);

te Terminal ENQ/ACK: enabled (+) or disabled (–) (default = +);

he Host ENQ/ACK: enabled (+) or disabled (–) (default = –);

tx Terminal XON/XOFF: enabled (+) or disabled (–) (default = –);

hx Host XON/XOFF: enabled (+) or disabled (–) (default = –);

im Input mode: block (b), character (c), or line (l) (default = b);

om Output mode: character (c) or line (l) (default = c);

ph Prompt handshake: if enabled (+), the emulator waits for the prompt sequence before sending each line of data during an input diversion; if disabled (–), no wait is performed (default = –);

pt Prompt timeout: number of seconds to allow for receipt of a prompt sequence during an input diversion; values range from 1 to 600, with 0 disabling the timeout altogether (default = 0);

st Single text terminators: list of characters, any of which terminates a line sent by the host computer when the emulator is in input line mode; up to eight characters may be specified (default = no characters);

dt Double text terminator: a pair of characters which together terminate a line sent by the host computer when the emulator is in input line mode (default = carriage-return/line-feed);

ps Prompt sequence: one or two characters which terminate a line sent by the host computer when the emulator is in input line mode, and which satisfy the prompt handshake if enabled (default = DC1);

bl Beginning of line: character to be prefixed to each line sent to the host computer (default = none);

el End of line: one or two characters to be postfixed to each line sent to the host computer (default = carriage-return);

es Local command character: character which designates a local command to be interpreted by the emulator if it comes at the beginning of a line read from the standard input (default = ~).

Note that emulation does not include block or format modes.

SEE ALSO

cu(1C) if simple connections are adequate or if you are calling another UNIX system;
uucp(1C) for file transfers with other UNIX systems.

HP-UX Asynchronous Communications Guide.

BUGS

Core capabilities, as well as standard extensions for graphics or color, are not yet formally defined.

NAME

atrans – translate assembly language

SYNOPSIS

atrans [–**j**] [–**n**] [filename]

HP-UX COMPATIBILITY

Level: HP-UX/DEVELOPMENT

Origin: HP

Remarks: *Atrans* is implemented on the Series 200 only.

DESCRIPTION

Atrans translates an assembly language *source file* from **Series 200** Pascal workstation assembly language syntax to **Series 200** HP-UX assembly language syntax. If no *filename* is given, input is assumed to come from **stdin.**

All uppercase characters are converted to lowercase characters, except those in comments or in quoted strings.

Hexadecimal constants are converted from **Series 200** Pascal workstation syntax, *$<hex number>* to the **Series 200** HP-UX syntax, *0x<hex number>*.

Operands whose effective address is *program counter* with displacement will have the string *pc* inserted in them (e.g. 8(d6) will become 8(pc,d6)).

Lines containing the following list of **Series 200** Pascal workstation pseudo-ops have no parallel in **Series 200** HP-UX syntax and are translated as comment lines: *decimal, end, llen, list, lprint, nolist, noobj, nosyms, page, spc, sprint, ttl.*

Lines containing the *mname* or *src* pseudo-ops are translated as comment lines, and a warning is printed stating that modules are not supported by the *Series 200* HP-UX assembler.

The pseudo-ops, *def, refa,* and *refr,* are translated as *globl.*

The file name operand of an *include* pseudo-op will have quote marks put around it.

Certain pseudo-ops require manual intervention to translate. Each Line containing these pseudo_ops will cause a message to be printed stating that an error will be generated by the **Series 200** HP-UX assembler. These pseudo-ops are: *com, lmode, org, rorg, rmode, smode, start.*

The –**j** option converts opcodes with the **b***cc*[**.s**|**.l**] branch syntax to the **j***cc* syntax. It also converts *bsr[.s|.l]* to *jbsr*.

The –**n** option converts groups of blanks to tabs.

SEE ALSO

as(1).

NAME

awk – text pattern scanning and processing language

SYNOPSIS

awk [–**F**c] [prog] [files]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

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

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, see 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 *>expr* is present), separated by the current output field separator, and terminated by the output record separator. The *printf* statement formats its expression list according to the format (see *printf*(3S)).

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

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

A relational expression is one of the following:

```
expression matchop regular-expression
expression relop expression
```

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

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

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

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

or by using the **–F***c* 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**).

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 }
```

SEE ALSO

grep(1), lex(1), sed(1).
Awk: A Programming Language for Manipulating Data in *HP-UX Concepts and Tutorials.*

BUGS

Input white space is not preserved on output if fields are involved.
There are no explicit conversions between numbers and strings. To force an expression to be treated as a number add 0 to it; to force it to be treated as a string concatenate the null string (" ") to it.

NAME

banner – make posters in large letters

SYNOPSIS

banner strings

HP-UX COMPATIBILITY

Level: System III Compatibility - HP-UX/STANDARD

Origin: System III

DESCRIPTION

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

Each argument is on a separate line.

NAME

basename, dirname – extract portions of path names

SYNOPSIS

basename string [suffix]
dirname string

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

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

Basename returns the *string* unmodified if *string* does not contain the indicated *suffix*, or if *string* contains the *suffix*, but the *suffix* does not occur at the end of the *string*. If *string* and *suffix* are identical, *basename* returns the null string.

Dirname delivers all but the last level of the path name in *string*. If *string* is null or does not contain a directory component, *dirname* returns ".", indicating the current working directory.

HARDWARE DEPENDENCIES

Series 200:

Basename returns the null string if the indicated *suffix* does not occur in *string*.

Basename returns an unpredictable string (sometimes the null string, sometimes one or more characters from the original *string*) if the indicated *suffix* contains one or more characters from the actual suffix in *string*, but the *suffix* does not match the actual suffix exactly. For example, the returned string is unpredictable in cases like

```
basename file.trf .r
```

If the indicated *suffix* occurs in *string*, but not at the end of *string*, *basename* returns only that part of *string* which occurs before the indicated *suffix*. For example,

```
basename file.f.g .f
```

returns "file".

EXAMPLES

The following shell script, invoked with the argument **/usr/src/cmd/cat.c**, compiles the named file and moves the output to a file named **cat** in the current directory:

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

The following example will set the shell variable **NAME** to **/usr/src/cmd**:

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

RETURN VALUE

Both commands return 0 for success, 1 for failure. *Dirname* always succeeds, and thus always returns 0.

SEE ALSO

expr(1), sh(1).

NAME

bc – arbitrary-precision arithmetic language

SYNOPSIS

bc [**–c**] [**–l**] [file ...]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Bc is an interactive processor for a language that resembles C but provides unlimited precision arithmetic. It takes input from any files given, then reads the standard input. The options are as follows:

–c compile only. *Bc* is actually a preprocessor for *dc*(1), which *bc* invokes automatically. Specifying **–c** prohibits invocation of *dc*, and sends the *dc* input to the standard output.

–l causes an arbitrary precision math library to be linked in with the code.

The syntax for *bc* programs is as follows;

L means a letter in the range a–z;
E means expression;
S means statement;
R means relational expression.

Comments are enclosed in /* and */.

Names may be any of the following:

simple variables: L
array elements: L [E]
the words "ibase", "obase", and "scale"
stacks: ~L

Other operands are:

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)
Strings of ASCII characters enclosed in quotes /RB (").

Arithmetic Operators (yeild an E as a result) consist of:

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

Relaltional Operators (yeild an R when used as E *op* e).

= =+ =– =* =/ =% =^

Statements consist of:

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

```
break
quit
```

Function definitions are:

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

The following functions are contained in the –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. No operators are defined for strings, but the string is printed if it appears in a context where an expression result would be printed. 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, again as defined by *dc*(1).

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.

EXAMPLE

```
scale = 20
define e(x) {
        auto a, b, c, i, s
        a = 1
        b = 1
        s = 1
        for(i=1; 1==1; i++) {
                a = a*x
                b = b*i
                c = a/b
                if(c == 0) return(s)
                s = s+c
        }
}
```

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

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

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

FILES

/usr/lib/lib.b	mathematical library

/usr/bin/dc desk calculator proper

SEE ALSO

bc(10), dc(1).

BUGS

There is currently no && (AND) or || (OR) comparisons.

The *for* statement must have all three expressions.

Quit is interpreted when read, not when executed.

Bc's parser is not robust in the face of input errors. Some simple expression like 2 + 2 will tend to get it back into phase.

NAME

bdiff – big diff

SYNOPSIS

bdiff file1 file2 [n] [–**s**]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

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

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

FILES

/tmp/bd?????

SEE ALSO

diff(1).

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

bs – a compiler/interpreter for modest-sized programs

SYNOPSIS

bs [file [args]]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Bs is a remote descendant of Basic and Snobol4 with a little C language thrown in. *Bs* is designed for programming tasks where program development time is as important as the resulting speed of execution. Formalities of data declaration and file/process manipulation are minimized. Line-at-a-time debugging, the *trace* and *dump* statements, and useful run-time error messages all simplify program testing. Furthermore, incomplete programs can be debugged; *inner* functions can be tested before *outer* functions have been written and vice versa.

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

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

```
statement
label statement
```

A label is a *name* (see below) followed by a colon. A label and a variable can have the same name.

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

Statement Syntax:

expression
: The expression is executed for its side effects (value, assignment, or function call). The details of expressions follow the description of statement types below.

break *Break* exits from the inner-most *for*/*while* loop.

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

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

continue
: *Continue* transfers to the loop-continuation of the current *for*/*while* loop.

dump [name]
: The name and current value of every non-local variable is printed. Optionally, only the named variable is reported. After an error or interrupt, the number of the last statement is displayed. The user function trace is displayed after an error or *stop* that occurred in a function.

edit A call is made to the editor selected by the EDITOR environment variable if it is present, or *ed*(1) if EDITOR is undefined or null. If the *file* option is present on the command line, that file is passed to the editor as the file to exit. (Otherwise no filename is used.) Upon exiting the editor, a *compile* statement (and associated *clear*) is executed giving that file name as it's argument.

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

execute
Change to immediate execution mode (an interrupt has a similar effect). This statement does not cause stored statements to execute (see ***run*** below).

for name = expression expression statement
for name = expression expression
...
next

for expression, expression, expression statement
for expression, expression, expression
...
next The *for* statement repetitively executes a statement (first form) or a group of statements (second form) under control of a named variable. The variable takes on the value of the first expression, then is incremented by one on each loop, not to exceed the value of the second expression. The third and fourth forms require three expressions separated by commas. The first of these is the initialization, the second is the test (true to continue), and the third is the loop-continuation action (normally an increment).

fun f([a, ...]) [v, ...]
...
nuf *Fun* defines the function name, arguments, and local variables for a user-written function. Up to ten arguments and local variables are allowed. Such names cannot be arrays, nor can they be I/O associated. Function definitions may not be nested. Calling an undefined function is permissible, see function calls below.

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

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

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

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

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

obase *N*
Obase sets the output base to *N* (see *ibase* above).

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

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

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

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

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

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

!shell command
An immediate escape to the shell.

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

Expression Syntax:

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

name ([expression [, expression] ...])
Functions can be called by a name followed by the arguments in parentheses separated by commas. Except for built-in functions (listed below), the name must be defined with a *fun* statement. Arguments to functions are passed by value. If the function is undefined, the call history to the call of the function is printed, and a requent for a return value (as an expression) is made. The result of that expression is taken to be the result of the undefined function. This permits debugging programs where not all the functions are yet defined. The value is read from the current input list.

name [expression [, expression] ...]
This syntax is used to reference either arrays or tables (see built-in *table* functions below). For arrays, each expression is truncated to an integer and used as a specifier for the name. The resulting array reference is syntactically identical to a name; **a[1,2]** is the same as **a[1][2]**. The truncated expressions are restricted to values between 0 and 32 767.

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

by a possibly signed exponent.

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

(expression)

Parentheses are used to alter the normal order of evaluation.

(expression, expression [, expression . . .]) [expression]

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

(False, True)[a = = b]

has the value **True** if the comparison is true.

? expression

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

– expression

The result is the negation of the expression.

+ + name

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

— name

Decrements the value of the variable. The result is the new value.

! expression

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

expression *operator* expression

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

Binary Operators (in increasing precedence):

=

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

_

_ (underscore) is the concatenation operator.

& |

& (logical and) has result zero if either of its arguments are zero. It has result one if both of its arguments are non-zero; | (logical or) has result zero if both of its arguments are zero. It has result one if either of its arguments is non-zero. Both operators treat a null string as a zero.

< <= > >= = = !=

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

+ −
Add and subtract.

* / % Multiply, divide, and remainder.

^ Exponentiation.

Built-in Functions:

Dealing with arguments

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

narg( ) returns the number of arguments passed. At level zero, the command argument count is returned.

Mathematical

abs(x) is the absolute value of x.

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

ceil(x) returns the smallest integer not less than x.

cos(x) is the cosine of x (radians).

exp(x) is the exponential function of x.

floor(x) returns the largest integer not greater than x.

log(x) is the natural logarithm of x.

rand( ) is a uniformly distributed random number between zero and one.

sin(x) is the sine of x (radians).

sqrt(x) is the square root of x.

String operations

size(s) the size (length in bytes) of s is returned.

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

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

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

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

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

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

File handling

open(name, file, function)
close(name)

The *name* argument must be a *bs* variable name (passed as a string). For the *open*, the *file* argument may be **1)** a 0 (zero), 1, or 2 representing standard input, output, or error output, respectively, **2)** a string representing a file name, or **3)** a string beginning with an **!** representing a command to be executed (via *sh* –*c*). The *function* argument must be either **r** (read), **w** (write), **W** (write without new-line), or **a** (append). After a *close*, the *name* reverts to being an ordinary variable. If *name* was a pipe, a *wait*(2) is executed before the close completes. The *bs* exit command does not do such a wait. The initial associations are:

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

Examples are given in the following section.

access(s, m)

executes *access*(2).

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

Tables

table(name, size)

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

item(name, i)
key() The *item* function accesses table elements sequentially (in normal use, there is no orderly progression of key values). Where the *item* function accesses values, the *key* function accesses the "subscript" of the previous *item* call. It fails (or in the absence of an *interrogate* operator, returns null) if there was no valid subscript for the previous *item* call. The *name* argument should not be quoted. Since exact table sizes are not defined, the interrogation operator should be used to detect end-of-table; for example:

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

If the interrogation operator is not used, the result of *item* is null if there are not further elements in the table. Null is, however, a legal "subscript".

iskey(name, word)

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

Odds and ends

eval(s) The string argument is evaluated as a *bs* expression. The function is handy for converting numeric strings to numeric internal form. *Eval* can also be used as a crude form of indirection, as in:

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

which increments the variable *xyz*. In addition, *eval* preceded by the interrogation operator permits the user to control *bs* error conditions. For example:

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

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

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

"plot(request, args)"

The *plot* function produces output on devices recognized by *tplot*(1G). **tplot is not currently supported on HP–UX.** The *requests* are as follows:

Call	*Function*
plot(0, term)	causes further *plot* output to be piped into *tplot*(1G) with an argument of –T*term*. **tplot is not currently supported on HP–UX.**
plot(4)	"erases" the plotter.
plot(2, string)	labels the current point with *string*.
plot(3,x1,y1,x2,y2)	draws the line between (*x1*,*y1*) and (*x2*,*y2*).
plot(4, x, y, r)	draws a circle with center (*x*,*y*) and radius *r*.
plot(5,x1,y1,x2,y2,x3,y3)	draws an arc (counterclockwise) with center (*x1*,*y1*) and endpoints (*x2*,*y2*) and (*x3*,*y3*).
plot(6)	is not implemented.
plot(7, x, y)	makes the current point (*x*,*y*).
plot(8, x, y)	draws a line from the current point to (*x*,*y*).
plot(9, x, y)	draws a point at (*x*,*y*).
plot(10, string)	sets the line mode to *string*.
plot(11, x1, y1, x2, y2)	makes (*x1*,*y1*) the lower left corner of the plotting area and (*x2*,*y2*) the upper right corner of the plotting area.
plot(12, x1, y1, x2, y2)	causes subsequent x (y) coordinates to be multiplied by *x1* (*y1*) and then added to *x2* (*y2*) before they are plotted. The initial scaling is **plot(12, 1.0, 1.0, 0.0, 0.0)**.

Some requests do not apply to all plotters. All requests except zero and twelve are implemented by piping characters to *tplot*(1G). **tplot is not currently supported on HP–UX.** See *plot*(4) for more details. Each statement executed from the keyboard re-invokes *tplot*, making the results unpredictable if a complete picture is not done in a single operation. Plotting should thus be done either in a function or a complete program, so all the output can be directed to *tplot* in a single stream. **tplot is not currently supported on HP–UX.**

last( ) in immediate mode, *last* returns the most recently computed value.

Programming Tips:

Using *bs* as a calculator:

```
$ bs
#  Distance (inches) light travels in a nanosecond.
186000 * 5280 * 12 / 1e9
11.78496
...
#  Compound interest (6% for 5 years on $1,000).
int = .06 / 4
bal = 1000
for i = 1 5*4 bal = bal + bal*int
bal - 1000
346.855007
...
exit
```

The outline of a typical *bs* program:

```
# initialize things:
var1 = 1
open("read", "infile", "r")
...
# compute:
while ?(str = read)
        ...
next
# clean up:
close("read")
...
# last statement executed (exit or stop):
exit
# last input line:
run
```

Input/Output examples:

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

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

SEE ALSO

ed(1), sh(1), access(2), printf(3S), stdio(3S), plot(4).

BUGS *Bs* is not extremely tolerant of some errors. Mistyping a declaration is painful, as a new definition cannot be made without doing a *clear* Using the *edit* command is the best soluction in this case.

The graphics mode is nearly useless without *tplot*, which is nor currently available. **tplot is not supported on HP–UX**

NAME

cal – print calendar

SYNOPSIS

cal [[month] year]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Cal prints a calendar for the specified year. If a month is also specified, a calendar just for that month is printed. If neither is specified, a calendar for the present month is printed. *Year* can be between 1 and 9999. The *month* is a number between 1 and 12. The calendar produced is that for England and her colonies.

Try September 1752.

BUGS

The year is always considered to start in January even though this is historically naive.

Beware that "cal 83" refers to the early Christian era, not the 20th century.

NAME

calendar – reminder service

SYNOPSIS

calendar [–]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

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

When an argument is present, *calendar* does its job for every user who has a file **calendar** in his login directory and sends him any positive results by *mail*(1). Normally this is done daily in the early morning hours under control of *cron*(8).

FILES

calendar	
/usr/lib/calprog	to figure out today's and tomorrow's dates
/etc/passwd	
/tmp/cal*	
/usr/lib/crontab	

SEE ALSO

mail(1), cron(8).

BUGS

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

NAME

cat – concatenate, copy, and print files

SYNOPSIS

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

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System V

DESCRIPTION

Cat reads each *file* in sequence and writes it on the standard output. Thus:

cat file

prints the file, and

cat file1 file2 >file3

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

If no input file is given, or if the argument – is encountered, *cat* reads from the standard input file.

The options are:

–u causes output to be unbuffered (character-by-character); normally, output is buffered.

–s makes *cat* silent about non-existent files, identical input and output, and write errors. Normally, no input file may be the same as the output file unless it is a special file.

–v causes non-printing characters (with the exception of tabs, new-lines and form-feeds) to be printed visibly. Control characters are printed ^*x* (control-*x*); the DEL character (octal 0177) is printed **^?**. Non-ASCII characters (with the high bit set) are printed as **M–***x***,** where *x* is the character specified by the seven low order bits.

–t when used with the **–v** option, causes tabs to be printed as **^I**'s.

–e when used with the **–v** option, causes a **$** character to be printed at the end of each line (prior to the new-line).

The **–t** and **–e** options are ignored if the **–v** option is not specified.

SEE ALSO

cp(1), pr(1).

WARNING

Command formats such as

cat file1 file2 >file1

overwrites the data in file1 before the concatenation begins. Therefore, take care when using shell special characters.

NAME

cb – C program beautifier, formatter

SYNOPSIS

cb [file]

HP-UX COMPATIBILITY

Level: C-Compiler - HP-UX/EXTENDED

Origin: System III

DESCRIPTION

Cb places a copy of the C program in *file* (standard input if *file* is not given) on the standard output with spacing and indentation that displays the structure of the program.

NAME

cc - C compiler

SYNOPSIS

cc [option] ... file ...

HP-UX COMPATIBILITY

Level: C Compiler – HP-UX/DEVELOPMENT

Origin: System III

DESCRIPTION

Cc is the HP-UX C compiler. It accepts several types of arguments:

Arguments whose names end with **.c** are taken to be C source programs; they are compiled, and each object program is left on the file whose name is that of the source with **.o** substituted for **.c**. The **.o** file is normally deleted, however, if a single C program is compiled and linked all in one step.

In the same way, arguments whose names end with **.s** are taken to be assembly source programs and are assembled, producing a **.o** file.

Arguments whose names end with **.o** are taken to be relocatable object files which are to be included in the link operation.

The following options are interpreted by *cc*. Options may not be concatenated. See *ld*(1) for link editor options.

–B*string* Find substitute compiler passes in the files named *string* with the suffixes **cpp**, **ccom**, **c1** and **c2**. *String* must be specified for **–B** to be meaningful.

–c Suppress the link edit phase of the compilation, and force an object (**.o**) file to be produced even if only one program is compiled. Produces a **.o** file for each **.c** file.

–E Run only the macro preprocessor on the named C programs, and send the result to the standard output. The result is compatible with the */lib/ccom* step of *cc*.

–g Cause the compiler to generate additional information needed for the use of a symbolic debugger.

–O Invoke an object-code optimizer.

–p Arrange for the compiler to produce code which counts the number of times each routine is called; also, if link editing takes place, replace the standard startoff routine by one which automatically calls *monitor*(3C) at the start and arranges to write out a **mon.out** file at normal termination of execution of the object program. An execution profile can then be generated by use of *prof*(1).

–P Similar to the **–E** option above, but the output goes to a corresponding file suffixed with **.i** which is suitable for compilation later. No compilation is done.

–S Compile the named C programs, and leave the assembler-language output on corresponding files suffixed **.s**.

–t[p012] Find only 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 **/lib/n**. Any or all of the pass designators **p**, **0**, **1**, or **2** may be specified, with the following meanings:

p – preprocessor;
0 – first pass of C compiler;
1 – second pass of C compiler;
2 – optimizer.

–v Enables verbose mode, which produces on *stdout* a step-by-step description of the compilation process.

–C
–D*name* = *def*
–D*name*
–H *nnn*
–I*dir*
–T
–U*name* These options are passed through to the C preprocessor, *cpp*. Refer to *cpp*(1) for details.

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

The Kernighan and Ritchie C text, and the various addenda to it, comprise the best available reference on C. The documents are intentionally ambiguous in some areas. HP-UX specifies some of these.

char
The **char** type is treated as signed by default. It may be declared **unsigned**.

pointers
Dereference of a NULL (zero) pointer is illegal and may cause a SIGSEGV error. This applies whether the access is for reading or writing. Some implementations may not be able to detect this error, in this case the result of such an access is undefined. Programs which rely on being able to derefrence a null pointer are not considered portable within HP-UX.

identifiers
Identifiers are significant up to (at least) 255 characters. Whether or not longer identifiers are handled is machine dependent. The assembler and loader must also support long identifiers to 255 characters.

HARDWARE DEPENDENCIES

Series 200:

The **–g** option is not currently supported.

Identifiers longer than 255 characters are not supported.

The following additional options are supported:

–a this option is passed directly to the assembler, *as*(1). Refer to *as*(1) for details.

–b causes the compiler to generate code for floating point operations that will use floating point hardware if it is installed in the computer at run-time.

–f causes the compiler to generate code for floating point operations that will use floating point hardware. This code will not run unless floating point hardware is installed at run-time.

–N< *secondary* >< *n* >
This option adjusts the size of internal compiler tables. The compiler uses fixed size arrays for certain internal tables. *Secondary* is one of the letters from the set {**dpw**}, and *n* is an integer value. *Secondary* and *n* are **not** optional. The table sizes can be re-specified using one of the secondary letters and the number *n* as follows:

d max size of the dimtab table. This table maintains information about the definitions of all structures, unions, and arrays. Default = 1000 table entries.

p max size of the parameter stack. Default = 150 table entries.

w max size of the switch table stack. Default = 250 table entries.

Two additional pass designators are available for the **–t** option. They are:

a assembler;
l linker;

Series 500:

The **–p** option is not currently supported.

An additional option, **–F**, is supported. The **–F** option causes the compiler to generate information for use by various program analysis commands.

The *ld* options **p** and **v** conflict with *cc* options, and thus cannot be accessed via *cc*.

The **–B** option is supported, but no substitute compiler passes are provided.

The file /lib/mcrt0.o is not currently supported.

Identifiers longer than 255 characters are not supported.

FILES

file.c	input file
file.o	object file
a.out	linked output
/tmp/ctm*	temporary
/lib/cpp	preprocessor
/lib/ccom	compiler, *cc*
/lib/c2	optional optimizer
/lib/crt0.o	runtime startoff
/lib/mcrt0.o	startoff for profiling
/lib/libc.a	standard library, see section 3 of this manual
/usr/include	standard directory for **#include** files

SEE ALSO

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.
as(1), ld(1), prof(1), monitor(3C).

DIAGNOSTICS

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

NAME

cd – change working directory

SYNOPSIS

cd [directory]

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

If specified, *directory* becomes the new working directory; otherwise, the value of the shell parameter **$HOME** is used. 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; therefore, it is recognized and executed by the shell.

SEE ALSO

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

NAME

cdb, fdb, pdb – C, FORTRAN, Pascal symbolic debugger

SYNOPSIS

cdb [**–d** *dir*] [**–r** *file*] [**–p** *file*] [**–a** *num*] [**–b** *num*] [**–s** *num*] [**–S** *num*] [**–u**] [*objectfile* [*corefile*]]

fdb [*cdb* options]

pdb [*cdb* options]

HP-UX COMPATIBILITY

Level: HP-UX/DEVELOPMENT

Origin: Third Eye Software

Remarks: This debugger is currently implemented on the Series 500 only.

TABLE OF CONTENTS

DESCRIPTION

Cdb, *fdb*, and *pdb* are alternate names for a source level debugger for C, HP FORTRAN, and HP Pascal programs. It provides a controlled environment for their execution.

Objectfile is an executable program file with one or more of its component modules compiled with debug option(s) turned on. The support module */usr/lib/end.o* must be included as the last object file in the list of those linked, except for libraries included with the **–l** option to *ld*(1). (Some systems automate this; see the *Hardware Dependencies* section below.) The default for *objectfile* is **a.out**.

Corefile is a core image from a failed execution of *objectfile*. The default for *corefile* is **core**.

The options are:

–d *dir* names an alternate directory where source files are located. You may have up to 16 alternate directories. They are searched in the order given. If a source file is not found in any alternate directory, the current directory is searched last.

–r *file* names a *record* file which is invoked immediately (for overwrite, not for append). See the section below entitled *Record and Playback Commands* for a description of this feature.

–p *file* names a *playback* file which is invoked immediately. See the section below entitled *Record and Playback Commands* for a description of this feature.

–a *num* sets the maximum number of assertions you can have active at once. The default *num* is 16. See the section below entitled *Assertion Control Commands* for details.

–b *num* sets the maximum number of breakpoints you can have active at once. The default *num* is 16. See the section below entitled *Breakpoint Commands* for details.

–s *num* sets the maximum number of special variables you can define during a debugging session. The default *num* is 26. See the section below entitled *Variable Name Conventions* for details.

–S *num* sets the size of the string cache to *num* bytes. The default *num* depends on the symbol table format used. The option is not available for all formats. The string cache holds data read from *objectfile*.

–u tells the debugger to expect names in the symbol table to start with an extra underscore. The option is not available for all symbol table formats.

There can only be one *objectfile* and one *corefile* per debugging session (activation of the debugger). The program (*objectfile*) is not invoked as a child process until you give an appropriate command (see the *Job Control Commands* section below). The same program may be restarted, as different child processes, many times during one debugging session.

This debugger is a complex, interactive tool with many synergistic and combinatoric features. What you can do with it is often limited only by your imagination. Remember, however, that the debugger is only a "window" on the world consisting mostly of the program being debugged and the system it runs on. If something puzzling happens, you may need to consult a manual which describes the program or the system, in order to understand the behavior.

CONVENTIONS

The debugger remembers the current file, current procedure, current line, and current data location. They are a function of what you have been viewing (not necessarily executing) most recently. Many commands use these current locations as defaults, and many commands set them as a side effect. It is important to keep this in mind when deciding what a command does in any particular situation.

For example, if you stop in procedure "abc", then view procedure "def", then ask for the value of local variable "xyz", the debugger assumes that the variable belongs to procedure "def".

Notational Conventions

Most commands are of the form "[*modifier*] *command-letter* [*options*]". Numeric modifiers before and after commands can be any numeric expression. They need not be just simple numbers. A blank is required before any numeric *option*. Multiple commands on one line must be separated by ";".

These are common modifiers and other special notations:

(A | B | C) Any one of A or B or C is required.

[A | B | C] Any one of A or B or C is optional.

file A file name.

proc A procedure (or function, or subroutine) name.

var A variable name.

number A specific, constant number (e.g. "9", not "4+5"). Floating point (real) numbers may be used any place a constant is allowed.

expr Any expression, but with limitations stated below.

depth A stack depth, as printed by the "**t**" command. The top procedure is at a *depth* of zero. A negative *depth* acts like a *depth* of zero. Stack depth usually means "exactly at the specified depth", not "the first instance at or above the specified depth".

format A style for printing data. See the *Data Viewing Commands* section below for details.

commands A series of debugger commands, separated by ";", entered on the command line or saved with a breakpoint or assertion. Semicolons are ignored (as commands) so they can be freely used as command separators. Commands may be grouped with "{}" for the "**a**", "**b**", "**if**", and "**!**" commands. In all other cases commands inside "{}" are ignored.

Variable Name Conventions

Variables are referenced exactly as they are named in your source file(s). Case sensitivity is controlled by the "**Z**" command. Be careful with one letter variable names, since they can be confused with commands. If an expression begins with a variable that might be mistaken for a command, just enclose the expression in "**()**" (e.g. "(k)"), or eliminate any white space between the variable and the first operator (use "k= 9" instead of "k = 9").

If you are interested in the value of some variable *var*, there are a number of ways of getting it, depending on where and what it is:

var Search the stack for the most recent instance of the current procedure. If found, see if *var* is a parameter or local variable of that procedure. If not, search for a global variable named either *var* or *_var*, in that order.

proc.var Search the stack for the most recent instance of *proc*. If found, see if it has a parameter or local variable named *var*, as before.

proc.depth.var
Use the instance of *proc* that is at depth *depth* (exactly), instead of the most recent instance. This is very useful for debugging recursive procedures where there are multiple instances on the stack.

:var Search for a global (not local) variable named either *var* or *_var*, in that order.

. *Dot* is shorthand for the last thing you viewed (see the *Data Viewing Commands* section below). It has the same size it did when you last viewed it. For example, if you look at a **long** as a **char**, then "." is considered to be one byte long. This is useful for treating things in unconventional ways, like changing the second highest byte of a **long** without changing the rest of the **long**. *Dot* may be treated like any other variable.

NOTE: "." is the *name* of this magic location. If you use it, it is dereferenced like any other name. If you want the *address* of something that is, say, 30 bytes farther on in memory, do not say ".+30". That would take the contents of *dot* and add 30 to it. Instead, say "&.+30", which adds 30 to the *address* of *dot*.

Special variables are names for things that are not normally directly accessible. Special variables include:

$*var* The debugger has room in its own address space for a number of user-created special variables. There are 26 of them by default (this number is adjustable using the **–s** invocation option). They are all of type **long**, and do not take on the type of any expression they are assigned to. Names are defined when they are first seen. For example, saying "$xyz = 3*4" creates special symbol "$xyz", and assigns to it the value 12. Special variables may be used just like any other variables.

$pc, **$fp**, **$sp**, **$r0**, etc.
These are the names of the program counter, the frame pointer, the stack pointer, the registers, etc. To find out which names are available on your system, use the "**l r**" (list registers)

command. All registers act as type **integer**.

$result This is used to reference the return value from the last procedure exit. Where possible, it takes on the type of the procedure. **$short** and **$long** are available as alternate ways of looking at **$result**.

$signal This lets you see and modify the current child process signal number.

$lang This lets you see and modify the current language (0 for C, 1 for FORTRAN, or 2 for Pascal).

$line This lets you see and modify the current source line number, which is also settable with a number of different commands.

$malloc
This lets you see the current amount of memory (bytes) allocated at run-time for use by the debugger itself.

$cBad This lets you see and modify the number of machine instructions the debugger will step while in a non-debuggable procedure before setting an up-level breakpoint and free-running to it. Setting it to a small value can improve debugger performance, at the risk of taking off free-running after missing the up-level break for some reason.

To see all the special variables, including the predefined ones, use the "**l s**" (list specials) command.

You can also look up code addresses with

proc#*line*

which searches for the given procedure name and line number (which must be an executable line within *proc*) and uses the code address of that line. Just referring to a procedure *proc* by name uses the code address of the entry point to that procedure.

Expression Conventions

Every expression has a value, even simple assignment statements, as in C. "Naked" expression values (those which aren't command modifiers) are always printed unless the next token is "**;**" (command separator) or "}" (command block terminator). Thus breakpoint and assertion commands (see the appropriate sections below) are normally silent. To force an expression result to be printed, follow the expression with "**/n**" (print in normal format; see below).

Integer constants may begin with "**0**" for octal or "**0x**" or "**0X**" for hexadecimal. They are **int** if they fit in two bytes, **long** otherwise. If followed immediately by "**l**" or "**L**", they are forced to be of type **long** (this is useful on systems where **int** is two bytes).

Floating point constants must be of the form *digits.digits*[**e**|**E**|**d**|**D**[+|-]*digits*], for example, "1.0", "3.14e8", or "26.62D-31". One or more leading digits is required to avoid confusion with "**.**" (*dot*). A decimal point and one or more following digits is required to avoid confusion for some command formats. If the exponent doesn't exactly fit the pattern shown, it is not taken as part of the number, but as separate token(s). The "**d**" and "**D**" exponent forms are allowed for compatibility with FORTRAN. However, all floating point constants are taken as doubles, regardless.

Character constants must be entered in **'** and are treated as **integer**s. String constants must be entered in " " and are treated like "**char ***" (e.g. pointer to **char**). Character and string constants may contain the standard backslashed escapes understood by the C compiler and the *echo*(1) command, including "**\b**", "**\f**", "**\n**", "**\r**", "**\t**", "****", "**\'**", and "*****nnn*". However, "****<newline>" is not supported, neither in quotes nor at the end of a command line.

Expressions are composed of any combination of variables, constants, and C operators. If the debugger is invoked as *cdb*, the C operator "**sizeof**" is also available. If the debugger is invoked as *fdb*, FORTRAN operators are also available and FORTRAN meanings take precedence where there is a conflict. The same is true for Pascal if the debugger is invoked as *pdb*.

If there is no active child process and no *corefile*, you can only evaluate expressions containing constants.

Expressions approximately follow the C rules of promotion, e.g. **char**, **short**, and **int** become **long**, and **float** becomes **double**. If either operand is a **double**, floating math is used. If either operand is **unsigned**, unsigned math is used. Otherwise, normal (integer) math is used. Results are then cast to proper destination types for assignments.

If a floating point number is used with an operator that doesn't normally permit it, the number is cast to **long** and used that way. For example, the C binary operator "~" (bit invert) applied to the constant "3.14159" is the same as "~3".

Note that " = " means "assign" except for Pascal; use " = = " or ".**EQ.**" for FORTRAN. In Pascal, " = " is a comparison operator; use ":= " for assignments. For example, if you invoke the debugger as **cdb**, then set "$lang = 2" (Pascal), you must say "$lang := 0" to return to C.

Use "//" for division, instead of "/", to distinguish from display formatting (see the *Data Viewing Commands* section below).

The special unary operator "**$in**" (not to be confused with debugger local variables) evaluates to 1 (true) if the operand is an address inside a debuggable procedure and **$pc** (the current child process program location) is also in that procedure, else it is 0 (false). For example, "**$in** main" is true if the child process is stopped in main().

If the first expression on a line begins with " + " or "−", use "()" around it to distinguish from the " + " and "−" commands (see the *Data Viewing Commands* section below). Parentheses may also be needed to distinguish an expression from a command it modifies.

You can attempt to dereference any constant, variable, or expression result using the C "*" operator. If the address is invalid, an error is given.

Whenever an array variable is referenced without giving all its subscripts, the result is the address of the lowest element referenced. For example, consider an array declared as "x[5][6][7]" in C, "x(5,6,7)" in FORTRAN, or "x[1..5,2..6,3..7]" in Pascal. Referencing it simply as "x" is the same as just "x" in C, the address of "x(1,1,1)" in FORTRAN, or the address of "x[1,2,3]" in Pascal. Referencing it as "x[4]" is the same as "& (x[4][0][0])" in C, the address of "x(1,1,4)" in FORTRAN, or the address of "x[4,2,3]" in Pascal.

If a not-fully-qualified array reference appears on the left side of an assignment, the value of the right-hand expression is stored into the element at the address specified.

Except for C, array indices are checked and must be within declared bounds.

String constants are stored in a magic buffer in the file */usr/lib/end.o*, which you link with your program. The debugger starts storing strings at the beginning of this buffer, and moves along as more assignments are made. If the debugger reaches the end of the buffer, it goes back and reuses it from the beginning. In general this doesn't cause any problems. However, if you use very long strings, or if you assign a string constant to a global pointer, problems could arise. To fix them, you can edit and compile a personal copy of */usr/lib/end.c* to increase the size of the buffer. (Some systems don't support this; see the *Hardware Dependencies* section below.)

Procedure Call Conventions

Procedures may be invoked from the command line, even within expressions. For example:

xyz = $abc * (3 + def (ghi − 1, jkl, "Hi Mom"))

calls procedure "def" when its value is needed in the expression.

Any breakpoints encountered during command line procedure invocation are handled as usual. However, the debugger has only one active command line at a time. If it stops in a called procedure for any reason, the remainder (if any) of the old command line is tossed, with notice given.

If you attempt to call a procedure when there is no active child process, one is started for you as if you gave a single-step command first. Unfortunately, this means that the data in *corefile* (if any) may disappear or be reinitialized.

If you send signal SIGINT (e.g., hit the BREAK key) while in a called procedure, the debugger aborts the procedure call and returns to the previous stopping point (the start of the main program for a new process).

You can call any procedure that is in your *objectfile*, even if it is not debuggable (was not compiled with debug on). For example, assume that you reference "printf()" in your program, so the code for it is in your *objectfile*. Then you can enter on the command line:

```
printf ( "This works! %d %c\n", 9, '?');
```

If you wonder what procedures are available, do a list labels command ("**l l**"). If you want to have some library routines available for debugging, but they aren't referenced anywhere in your code (so they aren't linked), you can modify a personal copy of */usr/lib/end.c* to reference them. (Some systems don't support this; see the *Hardware Dependencies* section below.) It is not necessary to have correct calls. For example, just "printf()" works fine, since you never execute the statements in *end.c*.

Note that procedure name "_end_" is declared in *end.c*.

COMMANDS

The debugger has a large number of commands for viewing and manipulating the program being debugged. They are explained below, grouped by functional similarity.

File Viewing Commands

These commands may change the current viewing position, but they do not affect the next statement to be executed in the child process, if any.

e Show the current file, procedure, line number, and source line (e.g. "test.c: testit: 28: a = 1;"). Commands that show the file and/or procedure with a source line skip (do not print) any leading white space from the source line.

e (*file* | *proc*)
Enter (view) *file* or *proc* and print its first line. *File* can be any file, not necessarily one of the source files for *objectfile* (but be careful not to view object code). For procedures, the "first line" is the procedure's first executable line, not its line of declaration.

[*depth*] **E** Like "**e**", but it sets the viewing location to the current location in the *proc* on the stack at depth *depth* (which is not necessarily the first executable line of the procedure). *Depth* defaults to zero, which is where the program is currently stopped.

L This is a synonym for **0E**.

line Print source line number *line* in the current file.

[*line*] **p** [*count*]
Print one (or *count*) lines starting at the current line (or line number *line*). If more than one line is printed, the current line is marked with a " = " in the leftmost print position.

+ [*lines*] Move to *lines* (default one) lines after the current line.

–[*lines*] Move to *lines* (default one) lines before the current line.

[*line*] **w** [*size*]
Print a window of text, *size* (default 11) lines big, centered about the current line (or *line*), which is marked with a " = " in the leftmost print position if more than one line is printed.

[*line*] **W** [*size*]
Same as "**w**", but *size* defaults to 21 lines.

+w [*size*]
+W [*size*]
Print a window of text, of the given or default *size*, beginning at the end of the previous window, if the previous command was a window command, or at the current line otherwise.

–w [*size*]
–W [*size*] Print a window of text, of the given or default *size*, ending at the beginning of the previous window, if the previous command was a window command, or at the current line otherwise.

If after any window command you give a "**w**" or "**W**" command with no *line* specified, the debugger prints the following window of source text, or the previous window if the previous window command was "**-w**" or "**-W**", using the given *size* (or the default if none). A simple carriage-return after any window command does the same thing, but uses the previous *size* as well.

/[*string*] Search forward through the current file, from the line after the current line, for *string*.

?[*string*] Search backward for *string*, from the line before the current line.

Searches wrap around the end or beginning of the file, respectively. If *string* is not specified, the previous one is used. Wild cards and regular expressions are not supported; *string* must be literal.

n Repeat the previous "/" or "**?**" command using the same *string* as previously.

N The same as "**n**", but the search goes in the opposite direction as specified by the previous "/" or "**?**" command.

Display Formats

A *format* is of the form "[*][*count*]*formchar*[*size*]".

"*" means "use alternate address map" (if maps are supported).

Count is the number of times to apply the format style *formchar*. It must be a *number*.

Size is the number of bytes to be formatted for each *count*, and overrides the default *size* for the format style. It must be a positive decimal *number* (except short hand notations, see below). *Size* is disallowed with those *formchar*s where it makes no sense.

For example, "abc/4x2" prints, starting at the location of "abc", four two-byte numbers in hexadecimal.

The formats which print numbers allow an upper-case character to be used instead, for the same results as appending "**l**" (see below). For example, "**O**" prints in long octal. These formats, which are useful on systems where **integer** is shorter than **long**, are noted below. The following formats are available:

n Print in the "normal" format, based on the type. Arrays of **char** and pointers to **char** are interpreted as strings, and structures are fully dumped.

(**d** | **D**) Print in decimal (as **integer** or **long**).

(**u** | **U**) Print in unsigned decimal (as **integer** or **long**).

(**o** | **O**) Print in octal (as **integer** or **long**).

(**x** | **X**) Print in hexadecimal (as **integer** or **long**).

(**b** | **B**) Print a byte in decimal (either way).

(**c** | **C**) Print a character (either way).

(**e** | **E**) Print in "e" floating point notation (as **float** or **double**) (see *printf*(3)). Remember that floating point constants are always doubles!

(**f** | **F**) Print in "f" floating point notation (as **float** or **double**).

(**g** | **G**) Print in "g" floating point notation (as **float** or **double**).

a Print a string using *expr* as the address of the first byte.

s Print a string using *expr* as the address of a pointer to the first byte. This is the same as saying "***expr*/a**", except for arrays.

t Show the type of *expr* (usually a variable or procedure name). For true procedure types you must actually call the procedure, e.g. "def(2)/t".

p Print the name of the procedure containing address *expr*.

S Do a formatted dump of a structure (only with symbol tables which support it). Note that *expr* must be the address of a structure, not the address of a pointer to a structure.

There are some short hand notations for *size*:

b 1 byte (**char**).

s 2 bytes (**short**).

l 4 bytes (**long**).

These can be appended to *formchar* instead of a numeric *size*. For example, "abc/xb" prints one byte in hexadecimal.

If you view an object with a *size* (explicitly or implicitly) less than or equal to the size of a **long**, the debugger changes the basetype to something appropriate for that *size*. This is so "." (*dot*) works correctly for assignments. For example, "abc/c2" sets the type of "." to **short**. One side effect is that if you look at a **double** using a **float** format, *dot* loses accuracy or has the wrong value.

Data Viewing Commands

expr If *expr* does not look like anything else (such as a command), it is handled as if you had typed "*expr*/**n**" (print expression in normal format), unless followed by ";" or "}", in which case nothing is printed.

expr/*format*
Print the contents (value) of *expr* using *format*. For example, "abc/x" prints the contents of "abc" as an **integer**, in hexadecimal.

expr?*format*
Print the address of *expr* using *format*. For example, "abc?o" prints the address of "abc" in octal.

^[[/]*format*]
Back up to the preceding memory location (based on the size of the last thing displayed). Use *format* if supplied, or the previous *format* if not. Note that no "/" is needed after the "^". Also note that you can reverse direction again (e.g. start going forward) by entering "." (*dot*), which is always an alias for the current location, followed by carriage returns.

l [*proc*[.*depth*]]
List all parameters and local variables of the current procedure (or of *proc*, if given, at the specified *depth*, if any). Data is displayed using "/**n**" format, except that all arrays and pointers are shown simply as addresses, and only the first word of any structure is shown.

l (**a** | **b** | **d** | **z**)
List all **a**ssertions, **b**reakpoints, **d**irectories (where to search for files), or **z**ignals (signal actions).

l (**f** | **g** | **l** | **p** | **r** | **s**) [*string*]
List all **f**iles (source files which built *objectfile*), **g**lobal variables, **l**abels (program entry points known to the linker), **p**rocedure names, **r**egisters, or **s**pecial variables (except registers). If *string* is present, only those things with the same initial characters are listed.

Stack Viewing Commands

[*depth*] **t**

Trace the stack for the first *depth* (default 20) levels.

[*depth*] **T**

The same as "**t**", but local variables are also displayed, using "/**n**" format (except that all arrays and pointers are shown simply as addresses, and structures as first words only).

Job Control Commands

The parent (debugger) and child (*objectfile*) processes take turns running. The debugger is only active while the child process is stopped due to a signal, including hitting a breakpoint, or terminated for whatever reason.

r [*arguments*]

Run a new child process with the given argument list (if any). The existing child process, if any, is terminated first. If no *arguments* are given, the ones used with the last "**r**" command are used again (none if "**R**" was used last).

Arguments may contain "<" and ">" for redirecting standard input and standard output. ("<" does an *open*(2) of file descriptor 0 for read-only; ">" does a *creat*(2) of file descriptor 1 with mode 0666). *Arguments* cannot contain shell variables, quote marks, or other special syntax. They cannot be enclosed in "{}" as with other commands, so "**r**" cannot be safely saved with a breakpoint or assertion.

R Run a new child process with no argument list.

k Terminate (kill) the current child process, if any.

[*count*] **c** [*line*]

Continue after a breakpoint or a signal, ignoring the signal, if any. If *count* is given, the current breakpoint, if any, has its *count* set to that value. If *line* is given, a temporary breakpoint is set at that line number, with a *count* of –1 (see the *Breakpoint Commands* section below).

[*count*] **C** [*line*]

Continue just like "**c**", but allow the signal (if any) to be received. This is fatal to the child process if it doesn't catch or ignore the signal!

[*count*] **s** Single step 1 (or *count*) statements. Successive carriage-returns repeat with a *count* of 1. If *count* is less than one, the child process is not stepped. Note that the child process continues with the current signal, if any! (You can set "$signal = 0" to prevent this.)

If you accidently step down into a procedure you don't care about, use the "**bU**" command to set a temporary up-level breakpoint, and then continue using "**c**".

[*count*] **S** Single step like "**s**", but treat procedure calls as single statements (don't follow them down). If a breakpoint is hit in such a procedure, or in one that it calls, its *commands* are executed. This is usually all right, but beware if there is a "**c**" command in that breakpoint's command list!

The debugger has no knowledge about or control over child processes forked in turn by the process being debugged. Also, it gets very confused (leading to "Bad access" messages) if the process being debugged executes a different program via *exec*(2).

Child process output may be (and usually is) buffered. Hence it may not appear immediately after you step through an output statement such as *printf*(3). It may not appear at all if you kill the process.

Breakpoint Commands

The debugger provides a number of commands for setting and deleting breakpoints. A breakpoint has three attributes associated with it:

address All the commands which set a breakpoint are simply alternate ways to specify the breakpoint address. The breakpoint is then encountered whenever *address* is about to be executed, regardless of the path taken to get there. Only one breakpoint at a time (of any type or count)

may be set at a given *address*. Setting a new breakpoint at *address* replaces the old one, if any.

count The number of times the breakpoint is encountered prior to recognition. If *count* is positive, the breakpoint is "permanent", and *count* decrements with each encounter. Each time *count* goes to zero, the breakpoint is recognized and *count* is reset to one (so it stays there until explicitly set to a different value by a "**c**" or "**C**" command).

If *count* is negative, the breakpoint is "temporary", and *count* increments with each encounter. Once *count* goes to zero, the breakpoint is recognized, then deleted.

A *count* of zero is used internally by the debugger and means that the breakpoint is deleted when the child process next stops for any reason, whether it hit that breakpoint or not. Commands saved with such breakpoints are ignored. Normally you never see these sorts of breakpoints.

Note that *count* is set to either –1 (temporary) or 1 (permanent) for any new breakpoint. It can then be modified only by the "**c**" or "**C**" command.

commands

Actions to be taken upon recognition of a breakpoint before waiting for command input. These are separated by ";" and may be enclosed in "{}" to delimit the list saved with the breakpoint from other commands on the same line. If the first character is anything other than "{", or if the matching "}" is missing, the rest of the line is saved with the breakpoint.

Remember that the results of expressions followed by ";" or "}" are not printed unless you specify a print format. You can use "/**n**" (normal format) to simply force printing of a result.

Saved commands are not parsed until the breakpoint is recognized. If *commands* are nil then, after recognition of the breakpoint, the debugger just waits for command input.

The debugger has only one active command line at a time. When it begins to execute breakpoint commands, the remainder (if any) of the old command line is tossed, with notice given.

Here are the breakpoint commands:

l b
B Both forms list all breakpoints in the format "*num*: count: *nnn proc*: *ln*: *contents*", followed by "{*commands*}", e.g.:

```
1: count: –1 (temporary)  sortall: 12: abc + = 1;
  {t;i/D}
2: count: 5  fixit: 29: def = abc >> 4;
  {Q;if *argv = = –1 {"Oops"}{c}}
```

The leftmost number is an index number for use with the "**d**" (delete) command.

[*line*] **b** [*commands*]
Set a permanent breakpoint at the current line (or at *line* in the current procedure). When the breakpoint is hit, *commands* are executed. If there are none, the debugger pauses for command input. If immediate continuation is desired, finish the command list with "**c**" (see breakpoint 2 in the example above).

For example, suppose you want to set a breakpoint in some file or procedure other than where you are at the moment. First, use the "**e**" command to get you to the right file or procedure. Look around for the line where you want the break to occur (using searches, or just by printing the lines). Once you are there, you can just say "**b**" to set a breakpoint on that line.

[*expr*] **d** Delete breakpoint number *expr*. If *expr* is absent, delete the breakpoint at the current line, if any. If there is none, the debugger executes a "**B**" command instead.

D [**b**] Delete all breakpoints. The "**b**" is optional.

For the following commands, if the second character is upper case, e.g. "**bU**" instead of "**bu**", then the breakpoint is temporary (*count* is –1), not permanent (*count* is 1).

[*depth*] **bb** [*commands*]
[*depth*] **bB** [*commands*]
Set a breakpoint at the beginning (first executable line) of the procedure at the given stack *depth*. If *depth* is not specified, it uses the current procedure, which might not be the same as the one at *depth* zero.

[*depth*] **bx** [*commands*]
[*depth*] **bX** [*commands*]
Set a breakpoint at the exit (last executable line) of the procedure at the given stack *depth*. If *depth* is not specified, it uses the current procedure, which might not be the same as the one at *depth* zero. The breakpoint is set at a point such that all returns of any kind go through it.

[*depth*] **bu** [*commands*]
[*depth*] **bU** [*commands*]
Set an up-level breakpoint. The breakpoint is set immediately after the return to the procedure at the specified stack *depth* (default one, not zero). A *depth* of zero means "current location", e.g. "**0bU**" is a way to set a temporary breakpoint at the current value of **$pc**.

[*depth*] **bt** [*proc*] [*commands*]
[*depth*] **bT** [*proc*] [*commands*]
Trace current procedure (or procedure at *depth*, or *proc*). This command sets breakpoints at both the entrance and exit of a procedure. By default, the entry breakpoint *commands* are "**Q;2t;c**", which shows the top two procedures on the stack and continues. The exit breakpoint is always set to execute "**Q;$result/n;c**", which prints the procedure's return value and continues.

If *depth* is given, *proc* must be absent or it is taken as part of *commands*. If *depth* is missing but *proc* is specified, the named procedure is traced. If both *depth* and *proc* are omitted, the current procedure is traced, which might not be the same as the one at *depth* zero.

If *commands* are present, they are used for the entrance breakpoint, instead of the default shown above.

address **ba** [*commands*]
address **bA** [*commands*]
Set a breakpoint at the given code address. Note that *address* can be the name of a procedure or an expression containing such a name. Of course, if the child process is stopped in a non-debuggable procedure, or in prologue code (before the first executable line of a procedure), things may seem a little strange.

The next few commands, while not strictly part of the breakpoint group, are used almost exclusively as arguments to breakpoints (or assertions).

if [*expr*] {*commands*} [{*commands*}]
If *expr* evaluates to a non-zero value, the first group of commands (the first "{}" block) is executed, else it (and the following "{", if any) is skipped. In general, all other "{}" blocks are always ignored (skipped), except when given as an argument to an "**a**", "**b**", or "**!**" command. The "**if**" command is nestable, and may be abbreviated to "**i**".

Q If the "quiet" command appears as the first command in a breakpoint's command list, the normal announcement of "*proc: line: text*" is not made. This allows quiet checks of variables, etc. to be made without cluttering up the screen with unwanted output. The "**Q**" command is ignored if it appears anywhere else.

"*any string you like*"

Print the given string, which may have the standard backslashed character escapes in it, including "\n" for newline. This command is useful for labelling output from breakpoint commands.

Assertion Control Commands

Assertions are lists of commands that are executed *before every statement.* This means that, if there is even one active assertion, the program is single stepped at the machine instruction level. In other words, it runs very slowly. The primary use for assertions is tracking down nasty bugs, such as when someone corrupts a global variable. Some examples follow the command descriptions.

Each assertion is individually active or suspended, plus there is an overall assertions mode. If any assertion is added or activated, or if all assertions become suspended, the global mode follows suit.

a *commands*

Create a new assertion with the given command list, which is not parsed until it's executed. As with breakpoints, the command list may be enclosed in "{}" to delimit it from other commands on the same line. Do an "**l a**" command to list all current assertions and the overall mode.

expr **a** (**a** | **d** | **s**)

Modify the assertion numbered *expr*: **a**ctivate it, **d**elete it, or **s**uspend it. Suspended assertions continue to exist, but have no effect until reactivated.

A Toggle the overall state of the assertions mechanism between *active* and *suspended.*

D a Delete all assertions.

[*flag*] **x** Force an exit from assertions mode. If *flag* is absent, or if it evaluates to zero, exit immediately. Otherwise, finish executing the current assertion first. If any assertion executes an "**x**" command, the child process stops and the assertion doing the "**x**" is identified.

The debugger has only one active command line at a time. When it begins to execute assertion commands, the remainder (if any) of the old command line is tossed, with notice given.

Certain commands ("**r**", "**R**", "**c**", "**C**", "**s**", "**S**", and "**k**") are not allowed while assertions are running. They must appear after the "**x**", if at all.

A useful assertion might be:

```
a L
```

This just traces execution a line at a time until "something" happens (e.g., you hit the BREAK key).

Another example:

```
a L; if (xyz > (def - 9) * 10) {A; 1 x; c} {abc -= 10}
```

This assertion prints the line just executed, then checks the condition. If it is false, "abc" is decremented by 10. If it is true, assertions are suspended, assertion mode is exited, and the program continues at normal speed. Without the number before the "**x**" command, the "**c**" command is not executed.

Another example:

```
a if (abc != $abc) {$abc = abc; abc/d; if (abc > 9) {x}}
```

This command sets up an assertion to report the changing value of some global variable ("abc"), and stop if it ever exceeds some value. It uses a debugger local variable ("$abc") to keep track of the old value of "abc".

Signal Control Commands

The debugger catches all signals bound for the child process before the child process sees them. (This is a function of the *ptrace*(2) mechanism.) For many signals, this is a reasonable thing to do. Most processes are not set up to handle segmentation errors, etc. However, some processes do quite a bit with signals and the constant need to continue from a signal catch can be tedious.

[*signal*] **z** [**i**][**r**][**s**][**Q**]

Maintains the "zignal" (signal) handling table. *Signal* is a valid signal number (the default is the current signal). The options (which must be all one word) toggle the state of the appropriate flag: **i**gnore, **r**eport, or **s**top. If "**Q**" is present, the new state of the signal is not printed.

Do an "**l z**" command to list the current handling of all signals. Note that just "**z**" with no options tells you the state of the current or selected signal.

For example, assuming a start up state of (don't ignore, don't report, don't stop), the command "**14z sr**" sets the alarm clock signal (at least for System III) to **stop** (but still don't **ignore**) and **report** that it occurred. Doing "**14z sr**" again toggles the flags back to the original state.

When the child process stops or terminates on a signal it is always reported, except for the breakpoint signal when the breakpoint commands start with "**Q**".

When the debugger ignores a signal, the "**c**" command then does not know about it. The signal is never ignored when the child process terminates, only when it stops.

Record and Playback Commands

The debugger supports a record-and-playback feature to help recreate program states and to record all debugger output. It is particularly useful for bugs requiring long setups.

The commands are:

>*file* Set or change recordfile to *file* and turn recording on. This rewrites *file* from the start. Only commands are recorded to this file.

>>*file* This is the same, but appends to *file* instead of overwriting.

>@*file*
>>@*file*

Set or change record-all file to *file*, for overwriting or appending. The record-all file may be opened or closed independently of (in parallel with) the recordfile. All debugger standard output is copied to the record-all file, including prompts, commands entered, and command output. However, child process output is not captured.

>(**t** | **f** | **c**)

Turn recording on ("**t**") or off ("**f**"), or close the recording file ("**c**"). When recording is resumed, it appends after commands recorded earlier. In this context, ">>" is the same as ">".

>@(**t** | **f** | **c**)

Turn record-all on, off, or close the record-all file. In this context, ">>@" is the same as ">@".

> Tell the current recording status. ">>" does the same thing.

>@ Tell the current record-all status. ">>@" does the same thing.

<*file* Start playback from *file*.

<<*file* Start playback from *file*, using the single-step feature of playback. Each command line from the playback file is presented before it is executed. A simple menu lets you execute ("<cr>") or skip ("**S**") the line, execute more than one line ("<num>"), continue ("**C**") or quit ("**Q**") single stepping, or ask for help ("**?**").

Only command lines read from the keyboard or a playback file are recorded in the recordfile. For example, if recording is turned on in an assertion, it doesn't "take effect" until assertion execution stops.

Command lines beginning with ">", "<", or "**!**" are not copied to the current recordfile (but they are copied to the record-all file). You can override this by beginning such lines with blanks.

NOTE: The debugger can of course be invoked with standard input, standard output, and/or standard error redirected, independent of record and playback. If the debugger encounters an end of file while standard input is redirected from anything other than a terminal, it prints a message to standard output and exits, returning zero.

Miscellaneous Commands

<carriage-return>
~ An empty line or a "~" command causes the debugger to repeat the last command, if possible, with an appropriate increment, if any. Repeatable commands are those which print a line, print a window of lines, print a data value, single step, and single step over procedures. Note that <carriage-return> is saved in a *record* file as a "~" command, to distinguish from **^D**.

^D Control-D is like <carriage-return>, but repeats the previous command ten times. Note that this command is saved in a *record* file as an empty line.

! [*command-line*]
This shell escape invokes a shell program. If *command-line* is present, it is executed via *system*(3). Otherwise, the environment variable SHELL gives the name of the shell program to invoke with a **–i** option, also using *system*(3). If SHELL is not found, the debugger executes "/bin/sh –i". In any case, the debugger then waits for the shell or *command-line* to complete.

As with breakpoints, *command-line* may be enclosed in "{}" to delimit it from other (debugger) commands on the same line. For example,

14b {!{date};c}; t; l a

sets a breakpoint at line 14 that calls *date*(1), then continues; then (after setting the breakpoint), the debugger does a stack trace, then lists assertions.

f ["*printf-style-format*"]
Set address printing format, using *printf*(3) format specifications (**not** debugger format styles). Only the first 19 characters are used. If there is no argument, the format is set to a system-dependent default. All addresses are assumed to be of type **long**, so you should handle all four bytes to get something meaningful.

F Find and fix bug (a useless but humorous command).

g *line* Go to an address in the procedure on the stack at *depth* zero (not necessarily the same as the current procedure). This changes the program counter so *line* is the next line to be executed.

h
help Print the debugger help file (command summary) using *more*(1).

I Print information (inquire) about the state of the debugger.

M Print the current text (*objectfile*) and core (*corefile*) address maps.

M (**t** | **c**) [*expr*; [*expr*;...]]
Set the **t**ext (*objectfile*) or **c**ore (*corefile*) address map. The first zero to six map values are set to the *expr*s given.

q Quit the debugger. To be sure you don't lose a valuable environment, this command requests confirmation.

Z Toggle case sensitivity in searches. This affects everything: File names, procedure names, variables, and string searches! The debugger starts out as **not** case sensitive.

HARDWARE DEPENDENCIES

The "**bx**" (break on exit) command requires that compilers support it by funneling all exits through one point. The breakpoint is always set at the last line of the procedure, which should be, but may not be, the sole exit point.

Series 200:

The debugger is not supported.

Series 500:

"**bx**" works, except for FORTRAN multiple returns. The compilers emit a special source line symbol for this exit point, after the last "visible" source line.

Series 500 supports two types of string formats in addition to null-terminated C strings. FORTRAN **character** variables consist of four-word (16-byte) string markers, where the second word plus the third word plus three is the byte address of the string itself, and the fourth word is the length of the string. Pascal **string** variables consist of a four-byte, word-aligned length word followed by the string characters.

If the current language is FORTRAN, or if you use "**/s**" format with **fdb** or **pdb**, the debugger interprets the variable (or expression) as a string marker (or address thereof), which is a null pointer if the second word of the marker is zero. Multiple-count formats show a series of fixed-length strings, beginning with the first one pointed to by the marker. Using "<cr>" or "^" to go forward or backward in memory uses the four words after or before the current string marker as the new marker.

If the current language is Pascal, or if you use "**/a**" format with **fdb** or **pdb**, the debugger interprets the variable (or expression) as a Pascal **string** (or address thereof). Multiple-count formats show a series of random-length strings, using successive length words, skipping any wasted bytes in the last word of the previous string. Likewise, using "<cr>" or "^" to go through memory skips the total bytes consumed in the last display.

FORTRAN arrays of **character** variables use only one string marker, but the debugger doesn't know this. You can see all the strings using a multiple-count format, e.g. "str[1]/6s", but indexing into any element other than 1 results in junk.

There is no easy way to assign into a FORTRAN or Pascal **string** (nor, for that matter, into a Pascal **packed array of char**, which looks like a simple array). Only one word is copied into the first word of the string marker or into the length word, regardless of the type of the expression result.

When a C parameter is declared as an array of anything, the highest type qualifier (array) shows up as a pointer instead. For example, "int x[]" looks like "int *x", and "char (*x)[]" looks like "char **x", but "char *x[]" is treated correctly as "pointer to array of **char**".

There is limited support for command-line calls of functions which return structures. The debugger interprets the start of heap as a structure of the return type. However, a call such as "abc()/t" displays the return type correctly.

There is never a *corefile*, so all features which depend on it don't work. Also, there are no address maps in the usual sense, so the "**M**" command is not supported.

$short and **$long** are available in addition to **$result**. However, **$result** is only set to (valid as) the return value from the last procedure called from the command line. If the procedure returns a **double**, **$result** is set to the value cast to **long**.

If a child process receives a signal and you then step with the "**s**" command (or run with assertions active), the process free-runs through the signal handler procedure (if any) before pausing (or doing assertions).

The source file *end.c* is not supported, so you can't customize */usr/lib/end.o*. The buffer size is fixed at 200 bytes. However, to force linking of library routines not otherwise referenced, you can use the **–u** option to *ld*(1).

All compiler front ends (*cc*(1), *fc*(1), and *pc*(1)) automatically tell the linker to include */usr/lib/end.o* for you if you give the **–g** (debug) option. (They don't know to do it if you instead

use debug options in source code.)

Both code and data pointers in *objectfile* contain segment numbers. At *exec*(2) time, all such pointers are mapped from *ld*(1) pseudo-values to real values based on actual segment numbers allocated. The debugger operates in "pseudo-address-space", so you won't notice anything unusual most of the time. All addresses look the same each time you invoke a new child process. For example, the heap always begins at "broken" address zero (0).

WARNING: The debugger's interaction with a child process is somewhat complicated, due to the "fixing" of pointer values written to the child and the "breaking" of pointers read from the child. If you tell the debugger to treat a pointer as a non-pointer, it may get confused, with unpredictable results. In particular, if you set a debugger special variable equal to a pointer value, then attempt to dereference that special variable, you will either get garbage or cause an access error.

In the rare case where maxheap is set very large (greater than ~70Mb) and your program uses shared EMS segments (from *memallc*(2)), the debugger may confuse pointers into the EMS segments with large addresses in the heap.

Addresses of unknown (non-debuggable) procedures are shown as call-type pointers, not data pointers. They can be distinguished because the high bit is set (e.g., the decimal value looks negative). Pointers of this form are not usable for anything; you can't dereference them nor set breakpoints based on them.

SYMBOL TABLE DEPENDENCIES

Series 500 compilers use the HP9000 Symbol Table Format.

The **–u** (unique names) option is only available for System III symbol table versions (e.g. not on Series 500).

HP9000 Symbol Table Format:

When you try to display a variable which is a FORTRAN format label, a Pascal file-of-text, or a Pascal set, with no display format or with normal format ("/**n**"), the value is shown as "{format-label}", "{file-of-text}", or "{set}", respectively. You can use other formats, such as "/**x**", to display the contents of such variables.

Procedures in FORTRAN and Pascal may have alias names in addition to normal names. Aliases are shown by the "**l p**" (list procedures) command. They can be used in place of the normal name, as desired.

The procedure name "_MAIN_" is used as the alias name for the main program (main procedure) in all supported languages. Do not use it for any debuggable procedures.

When a compiler does not know array dimensions, such as for some C and FORTRAN array parameters, it uses 0:MAXINT or 1:MAXINT, as appropriate. The "/**t**" format shows such cases with "[]" (no bounds specified), and subscripts from 0 (or 1) to MAXINT are allowed in expressions.

Even though the symbol table supports C structure, union, and enumeration tags, C typedefs, and Pascal types, the debugger does not know how to search for them, even for the "/**t**" format. They are "invisible".

The debugger does not know about (search for) Pascal variant record tag fields nor variant fields.

Some variables are indirect, so a child process must exist in order for the debugger to know their addresses. When there is no child process, the address of any such variable is shown as 0xfffffffe.

The optional pattern given with the "**l g**" (list globals) command must be an exact match, not just a leading pattern.

The string cache (see the **–S** option) defaults to 1Kbyte in size. This cache holds data read from the Value Table.

Do not include executable source lines (from a separate file) within any procedure. If you do, such source lines appear to belong to the other source file, e.g. they don't belong to the file that contains the procedure declaration, so they can't be found. In the worst case if the first executable line of the main procedure is in a different source file than the procedure declaration, the debugger fails during start up.

If you do any includes within a procedure, even just of declarations (no code), the debugger may get confused and show you a wrong procedure name in some cases.

Symbol names in the Value Table are never preceded by underscores, so the debugger never bothers to search for names of that form. The only time a prefixed underscore is expected is when searching the Linker Symbol Table for names of non-debuggable procedures.

FILES

a.out Default *objectfile* to debug.

core Default *corefile* to debug.

/usr/lib/cdb.help
Text file listed by the "**help**" command.

/usr/lib/cdb.error
Text file which explains debugger error and warning messages.

/usr/lib/end.o
Object file to link with all debuggable programs.

/usr/lib/end.c Source file for end.o.

SEE ALSO

cc(1), echo(1), ld(1), more(1), creat(2), exec(2), fork(2), open(2), printf(3), system(3), a.out(5).

On some systems any of the following may exist: adb(1), fc(1), pc(1), sdb(1), ptrace(2), core(5), symtab(5), user(5).

DIAGNOSTICS

Most errors cause a reasonably accurate message to be given. Normal debugger exits return zero and error exits return one. All debugger output goes to standard output except error messages given just before non-zero exits, which go to standard error.

Debugger errors are preceded by "panic: ", while user errors are not. If any error occurs during initialization, the debugger then prints "cannot continue" and quits. If any error happens after initialization, the debugger attempts to reset itself to an idle state, waiting for command input. If any error occurs while executing a procedure call from the command line, the context is reset to that of the normal program.

Child process (program) errors result in signals which are communicated to the debugger via the *ptrace*(2) mechanism. If a program error occurs while executing a procedure call from the command line, it is handled like any other error (i.e. you can investigate the called procedure). To recover from this, or to abort a procedure call from the command line, type DEL, BREAK, ^C, or whatever your interrupt character is.

For more information, see the text file */usr/lib/cdb.errors*.

WARNINGS

Code that is not debuggable or does not have a corresponding source file is dealt with in a half-hearted manner. The debugger shows "unknown" for unknown file and procedure names, cannot show code locations or interpret parameter lists, etc. However, the linker symbol table provides procedure names for most procedures, even if not debuggable. The main procedure (main program) must be debuggable

and have a corresponding source file.

On some systems, if the debugger is run on a shared *objectfile* you cannot set breakpoints. (This may only apply if someone else is also executing the program.) This may be indicated by the error "Bad access" when you attempt to start a child process. If another person starts running *objectfile* while you are debugging, they and you may have some interesting interactions.

If the *address* given to a "**ba**" command is a not a code address in the child process, strange results or errors may ensue.

If you set the address printing format to something *printf*(3) doesn't like, you may get an error (usually memory fault) each time you try to print an address, until you fix the format with another "**f**" command.

Do not use the "**z**" command to manipulate the SIGTRAP signal. If you change its state you had better know what you are doing or be a very good sport!

If you single step or run with assertions through a call to *longjmp*(3), the child process will probably take off free-running as the debugger sets but never hits an up-level breakpoint.

Do not modify any file while the debugger has it open. If you do, the debugger gets confused and may display garbage.

Although the debugger tries to do things reasonably, it is possible to confuse the recording mechanism. Be careful about trying to playback from a file currently open for recording, or vice versa; strange things can happen.

Many compilers only issue source line symbols at the end of each logical statement or physical line, *whichever is greater*. This means that, if you are in the habit of saying "a = 0; b = 1;" on one line, there is no way to put a breakpoint after the assignment to "a" but before the assignment to "b".

Multi-line statements, such as a multi-line *if*, may only have a line symbol generated at the end of the list of conditions. If you try to set a breakpoint on any but the last line of this statement, the breakpoint will actually be set *on the preceding statement.* Also, if you try to set a breakpoint before the first executable line of a procedure, it may be set at the last line of the previous procedure. You can detect this because the debugger tells you what line it really set the breakpoint on.

Some statements do not emit code where you would expect it. For example, assume:

```
 99:for (i = 0; i < 9; i++) {
100:xyz (i);
101:}
```

A breakpoint placed on line 99 will be hit only once in some cases. The code for incrementing is placed at line 101. Each compiler is a little different; you must get used to what your particular compiler does. A good way of finding out is to use single stepping to see in what order the source lines are executed.

The output of some program generators, such as *yacc*(1), have compiler line number directives in them that can confuse the debugger. It expects source line entries in the symbol table to appear in sorted order. Removal of line directives fixes the problem, but makes it more difficult to find error locations in the original source file. The following script, run after *yacc*(1) and before *cc*(1), comments out line number changes in C programs:

```
sed "/# *line/s/^.*$/\/*&*\//" y.tab.c >temp.c
```

In general, line number directives (or compiler options) are only safe so long as they never set the number backwards.

BUGS

The C operators "++", "--", and "**?:**" are not available. The debugger always understands all the other C operators, except "**sizeof**", if the default language is FORTRAN or Pascal.

For FORTRAN, only the additional operators "**.NE.**", "**.EQ.**", "**.LT.**", "**.LE.**", "**.GT.**", and "**.GE.**" are supported.

For Pascal, only the operators "**:=**", "**<>**", "**^**", "**^.**" (as in "x^.y"), "**and**", "**or**", "**not**", "**div**", and "**mod**" are added.

The "**/t**" format mostly knows how to print type information using only C syntax. Also, it's confused about "array of pointer", e.g., "int *x[]" is shown as "int x[](*)". However, it does show FORTRAN array subscripts correctly (e.g. right to left).

Multiple array dimensions must always be given separately, as in C, for example, "x[3][2][4]". FORTRAN array indices must be given using "**[]**", not "**()**".

There is no support for FORTRAN entry points. They don't show up with the "**l p**" (list procedures) command, and you can't call them or reference them by name.

There is no support for FORTRAN **complex** variables, except as a series of two separate **float**s or **double**s.

The debugger doesn't understand C type casts.

The C operators "**&&**" and "||" aren't short circuit evaluated as in the compiler. All parts of expressions involving them are evaluated, with any side-effects, even if it's not necessary.

The debugger doesn't understand C pointer arithmetic. "*(a+n)" is not the same as "a[n]" unless "a" has an element size of 1.

There is no support for C local variables declared in nested blocks, nor for any local overriding a parameter with the same name. When looking up a local by name, parameters come first, then locals in the order of the "}"s of the blocks in which they are declared. When listing all locals, they are shown in the same order. When there is a name overlap, the address or data shown is that of the first variable with that name.

There is no support for Pascal intermediate variables. To reference a variable local to an enclosing procedure, you must specify the procedure name and stack depth in the usual way (*proc.depth.var*).

There is no support for Pascal packed arrays where the element size is not a whole number of bytes. Any reference into such an array may produce garbage or a bad access.

Pascal WITH statements are not understood. To access any variable you must specify the complete "path" to it.

If you set the maximum allowed number of breakpoints, then do a "**r**" or "**R**" (run) command, you get a "Too many breakpoints" error because the debugger needs to set one internally. This can happen for similar reasons at other times, too.

The debugger supports call-by-reference only for known parameters of known (debuggable) procedures. If the object to pass lives in the child process, you can fake such a call by passing "& object", i.e. the address of the object.

Array parameters are always passed to command-line procedure calls by address. This is correct except for Pascal call-by-value parameters. Structure parameters are passed by address or value, as appropriate, but only a maximum of eight bytes is passed, which can totally confuse the called procedure. FORTRAN string markers are never passed correctly. Only the first number of a complex pair is passed as a parameter. Functions which return complex numbers are are not called correctly; insufficient stack space is allocated for the return area, which can lead to overwriting the parameter values.

Assignments into objects greater than four bytes in size, from debugger special variables, result in errors or invalid results.

Case-insensitive searches are done in a crude way which equates some non-letters with other non-letters. For example, "[" and "{" are equal, as are "@" and "`".

Command lines longer than 1024 bytes are broken into pieces of that size. This may be relevant if you run the debugger with playback or with input redirected from a file.

NAME

cdc – change the delta commentary of an SCCS delta

SYNOPSIS

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

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Cdc changes the *delta commentary*, for the *SID* specified by the **–r** keyletter, of each named SCCS file.

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

If a directory is named, *cdc* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read (see *WARNINGS*); each line of the standard input is taken to be the name of an SCCS file to be processed.

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

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

–r*SID* Used to specify the *S*CCS *ID*entification (*SID*) string of a delta for which the delta commentary is to be changed.

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

MR entries are added to the list of **MR**s 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 **MR**s, it is removed and changed into a "comment" line. A list of all deleted **MR**s is placed in the comment section of the delta commentary and preceded by a comment line stating that they were deleted.

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

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

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

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

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

The exact permissions necessary to modify the SCCS file are documented in the article indicated under SEE ALSO. Simply stated, they are either (1) if you made the delta, you can change its delta commentary; or (2) if you own the file and directory you can modify the delta commentary.

EXAMPLES

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

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

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

does the same thing.

FILES

x-file (see *delta*(1))
z-file (see *delta*(1))

SEE ALSO

admin(1), delta(1), get(1), help(1), prs(1), sccsfile(5).
SCCS User's Guide in *HP-UX Concepts and Tutorials.*

DIAGNOSTICS

Use *help*(1) for explanations.

WARNINGS

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

NAME

chatr – change program's internal attributes

SYNOPSIS

/lbin/chatr [+ **c**|–**c**] [+ **g**|–**g**] [+ **h**|–**h**] [–**m***n*] [+ **n**|–**n**] [+ **p**|–**p**] [–**s**] [+ **z**|–**z**] file ...

HP-UX COMPATIBILITY

Level: HP-UX/OPTIONAL

Origin: HP

Remarks: *Chatr* is implemented on the Series 500 only.

DESCRIPTION

Chatr, by default, prints each *file*'s magic number and file attributes to the standard output. With one or more optional arguments, *chatr* performs the following operations:

c set (+) or clear (–) the virtual bit for each code segment.

g set (+) or clear (–) the virtual bit of the global data segment.

h set (+) or clear (–) the virtual bit for the heap of a two data segment program.

–**m***n* change the maximum heap size to *n* bytes.

n mark code as shareable (+) (magic number = SHARE_MAGIC), or unshareable (–) (magic number = EXEC_MAGIC).

p set (+) or clear (–) the paged and virtual bits for the heap of a two data segment program.

–**s** perform action silently.

z set (+) or clear (–) the demand load bit for each segment.

Upon completion, *chatr* prints the file's old and new values to the standard output file, unless –**s** is in effect.

RETURN VALUE

Chatr returns zero on success. If the call to *chatr* is syntactically incorrect, or one or more of the specified files cannot be acted upon, *chatr* returns the number of files whose attributes could not be modified. If no files are specified, *chatr* returns decimal 255.

SEE ALSO

ld(1), a.out(5), magic(5).

DIAGNOSTICS

Chatr generates an error message for the following conditions:

no arguments are supplied – in this case the syntax is printed to the standard error file;

cannot open a file;

a request is made to modify a file which is not EXEC_MAGIC or SHARE_MAGIC.

Chatr generates a warning message for the following conditions:

the + **p**, –**p**, + **h**, or –**h** option is specified for a file which is a one data segment program;

the –**m** option is specified for a file which is a one data segment program, or a file for which the data is unpaged.

NAME

chmod – change mode

SYNOPSIS

chmod mode file ...

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

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

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

A symbolic *mode* has the form:

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

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

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

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

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

Only the owner of a file (or the super-user) may change its mode. Only the super-user may set or clear the sticky (save text) bit.

EXAMPLES

The first example denies write permission to others, and the second makes a file executable (using symbolic mode):

```
chmod o–w file
chmod +x file
```

The first example below assigns read and execute permission to everybody, and sets the set-user-id bit. The second assigns read and write permission to the file owner, and read permission to everybody else (using absolute mode):

```
chmod 4555 file
chmod 644 file
```

SEE ALSO

ls(1), chmod(2).

NAME

chown, chgrp – change file owner or group

SYNOPSIS

chown owner file ...

chgrp group file ...

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

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

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

In order to change the owner or group, you must own the file or be the super-user.

FILES

/etc/passwd
/etc/group

SEE ALSO

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

NAME

chroot – change root directory for a command

SYNOPSIS

chroot newroot command

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

The given command is executed *relative to the new root*. The meaning of any initial slashes (/) in path names is changed for a command and any of its children to *newroot*. Furthermore, the initial working directory is *newroot*.

Notice that:

```
chroot newroot command >x
```

will create the file **x** relative to the original root, not the new one.

Command includes both the command name and any arguments.

This command is restricted to the super-user.

Chroot does not search **PATH** for the location of *command*, so the absolute path name of *command* must be given.

The new root path name is always relative to the current root. Even if a *chroot* is currently in effect, the *newroot* argument is relative to the current root of the running process.

SEE ALSO

chdir(2).

BUGS

Command cannot be in a shell script.

NAME

chsh – change default login shell

SYNOPSIS

chsh name [shell]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: UCB 4.2

DESCRIPTION

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

An example use of this command is:

```
chsh bill /bin/csh
```

SEE ALSO

csh(1), passwd(1), passwd(5).

NAME

cmp – compare two files

SYNOPSIS

cmp [**–l**] [**–s**] file1 file2

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

The two files are compared. (If *file1* is –, the standard input is used.) Under default options, *cmp* makes no comment if the files are the same; if they differ, it announces the byte and line number at which the difference occurred. If one file is an initial subsequence of the other, that fact is noted.

The options are:

–l Print the byte number (decimal) and the differing bytes (octal) for each difference. (Byte numbering begins at 1, rather than at 0 as is common.)

–s Print nothing for differing files; return codes only.

SEE ALSO

comm(1), diff(1).

DIAGNOSTICS

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

NAME

col – filter reverse line-feeds and backspaces

SYNOPSIS

col [**–bflpx**]

HP-UX COMPATABILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Col reads from the standard input and writes onto the standard output. It performs the line overlays implied by reverse line-feeds (ASCII code **ESC-7**), and by forward and reverse half-line-feeds (**ESC-9** and **ESC-8**). It also removes backspaces in favor of multiply overstruck lines. *Col* is particularly useful for filtering multi-column output made with the **.rt** command of *nroff*(1) and output resulting from use of the *tbl*(1) preprocessor.

If the **–b** option is given, *col* assumes that the output device in use is not capable of backspacing. In this case, if two or more characters are to appear in the same place, only the last one read will be output.

If the **–l** option is given, *col* assumes the output device is a line printer (rather than a character printer) and removes backspaces in favor of multiply overstruck full lines. It generates the minimum number of print operations necessary to generate the required number of overstrikes. (All but the last print operation on a line are separated by carriage returns (\r); the last print operation is terminated by a newline (\n).)

Although *col* accepts half-line motions in its input, it normally does not emit them on output. Instead, text that would appear between lines is moved to the next lower full-line boundary. This treatment can be suppressed by the **–f** (fine) option; in this case, the output from *col* may contain forward half-line-feeds (**ESC-9**), but will still never contain either kind of reverse line motion.

Unless the **–x** option is given, *col* will convert white space to tabs on output wherever possible to shorten printing time.

The ASCII control characters **SO** (\017) and **SI** (\016) are assumed by *col* to start and end text in an alternate character set. The character set to which each input character belongs is remembered, and on output **SI** and **SO** characters are generated as appropriate to ensure that each character is printed in the correct character set.

On input, the only control characters accepted are space, backspace, tab, return, new-line, **SI**, **SO**, **VT** (\013), and **ESC** followed by **7**, **8**, or **9**. The **VT** character is an alternate form of full reverse line-feed, included for compatibility with some earlier programs of this type. All other non-printing characters are ignored.

Normally, *col* will ignore any unrecognized escape sequences found in its input; the **–p** option may be used to cause *col* to output these sequences as regular characters, subject to overprinting from reverse line motions. The use of this option is highly discouraged unless the user is fully aware of the textual position of the escape sequences.

Note that the input format accepted by *col* matches the output produced by *nroff*(1) with either the **–T37** or **–Tlp** options. Use **–T37** (and the **–f** option of *col*) if the ultimate disposition of the output of *col* will be a device that can interpret half-line motions, and **–Tlp** otherwise.

SEE ALSO

nroff(1), tbl(1).

BUGS

Cannot back up more than 128 lines.

Allows at most 800 characters, including backspaces, on a line.

Local vertical motions that would result in backing up over the first line of the document are ignored. As a result, the first line must not have any superscripts.

NAME

comm – select or reject lines common to two sorted files

SYNOPSIS

comm [– [**123**]] file1 file2

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Comm reads *file1* and *file2*, which should be ordered in ASCII collating sequence (see *sort*(1)), and produces a three-column output: lines only in *file1*; lines only in *file2*; and lines in both files. The file name – means the standard input.

Flags 1, 2, or 3 suppress printing of the corresponding column. Thus

comm –12

prints only the lines common to the two files;

comm –23

prints only lines in the first file but not in the second;

comm –123

is a no-op.

SEE ALSO

cmp(1), diff(1), sort(1), uniq(1).

NAME

cp, ln, mv – copy, link or move files

SYNOPSIS

cp file1 [file2 ...] target
ln [**–f**] file1 [file2 ...] target
mv [**–f**] file1 [file2 ...] target

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

File1 is copied (linked, moved) to *target*. Under no circumstance can *file1* and *target* be the same. If *target* is a directory, then one or more files are copied (linked, moved) to that directory. If two or more files are specified for any of these commands (not counting *target*), then *target* must be a directory.

Only *mv* will allow *file1* to be a directory, in which case the directory rename will occur only if the two directories have the same parent.

Ln and *mv* will ask for permission if *target* already exists and is not writable. This is done by printing the mode (see *chmod*(2)), and reading one line from the standard input (if the standard input is a terminal). If the line (which you type in) begins with y, the operation will take place. Any other response will abort it. (Note that this will not occur if you are the super-user, since all files are considered writable by the super-user. *Cp* behaves similarly, in that the super-user is allowed to overwrite an existing file, while ordinary users are not.) The **–f** option will force these operations to occur without your intervention.

You cannot use *mv* to perform the following operations:

rename either the current working directory or its parent directory using the "." or ".." notation;

rename a directory such that its new name is the same as the name of a file contained in that directory.

SEE ALSO

cpio(1), link(1), rm(1), chmod(2).

BUGS

If *file1* and *target* lie on different file systems, *mv* must copy the file and delete the original. In this case the owner name becomes that of the copying process and any linking relationship with other files is lost.

Ln will not link across file systems.

You cannot use *mv* to rename a directory when its name ends in a slash (/).

NAME

cpio – copy file archives in and out

SYNOPSIS

cpio –o [**acBvx**]

cpio –i [**BcdmPrstuvx6**] [patterns]

cpio –p [**adlmuvx**] directory

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Cpio –o (copy out) 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.

Cpio –i (copy in) extracts from the standard input (which is assumed to be the product of a previous **cpio –o**) the names of files selected by zero or more *patterns* given in the name-generating notation of *sh*(1). In *patterns*, metacharacters **?**, *****, and [...] match the slash / character. Multiple *patterns* may be specified. If no *patterns* are specified, the default is ***** (i.e., select all files). The extracted files are conditionally created and copied into the current directory tree based upon the options described below.

Cpio –p (pass) reads the standard input to obtain a list of path names of files that are conditionally created and copied into the destination *directory* tree based upon the options described below.

The meanings of the available options are:

a Reset access times of input files after they have been copied.

B Input/output is to be blocked 5120 bytes to the record (does not apply to the *pass* option; recommended only with data directed to or from /**dev/rmt?**).

d *Directories* are to be created as needed.

c Write *header* information in ASCII character form for portability.

P Read a file written on a **PDP–11** or **VAX** system (with byte swapping) that did not use the **–c** option. Only useful with **–i** (copy in). Only bytes contained in the header are swapped. Non-ascii files will probably need further processing to be readable; this processing requires knowledge of the content of the file and thus cannot be done by this program. (**PDP–11** and **VAX** are registered trademarks of Digital Equipment Corporation).

r Interactively *rename* files. If the user types a null line, the file is skipped.

s Identical to the **P** option, except that all bytes in the file are swapped (including the header).

t Print only a *table of contents* of the input. No files are created, read, or copied.

u Copy *unconditionally* (normally, an older file will not replace a newer file with the same name).

x Save or restore device special files. *Mknod(2)* will be used to recreate these files on a restore, and thus –ix can only be used by the super-user. Restoring device files onto a different system can be very dangerous. This is intended for intrasystem (backup) use.

v *Verbose*: causes a list of file names to be printed. When used with the **t** option, the table of contents looks like the output of an **ls –l** command (see *ls*(1)).

l Whenever possible, link files rather than copying them. Usable only with the **–p** option.

m Retain previous file modification time. This option is ineffective on directories that are being copied.

6 Process an old (i.e., UNIX *Sixth* Edition format) file. Only useful with **–i** (copy in).

When the end of the tape is reached, *cpio* will prompt the user for a new special file and continue.

Note that *cpio* archives created using a raw device file must be read using a raw device file.

If you want to pass one or more metacharacters to *cpio* without the shell expanding them, be sure to precede each of them with a backslash (\).

Device files written with the **–ox** option (e.g. /dev/tty03) will not transport to other implementations of HP-UX.

HARDWARE DEPENDENCIES

Series 200/500:

All files with i-nodes greater than or equal to 65535 are unlinkable with the **–i** option. A separate copy of each file is made instead.

The number of blocks reported by *cpio* is always in units of 512-byte blocks, regardless of the block size of the initialized media.

The **–B** option ***must*** be used when writing directly (i.e. without using *tcio*(1)) to a CS-80 cartridge tape unit (HP 88140L/S). Warning: using *cpio* to write directly to a cartridge tape unit can severely damage the tape drive in a short amount of time, and is therefore strongly discouraged. The recommended method of writing to the cartridge tape unit is to use *tcio*(1) in conjunction with *cpio* (note that **–B** must ***not*** be used when *tcio*(1) is used). *Tcio*(1) buffers data into larger pieces, yielding better system performance and less wear and tear on the media and tape drive. A minimum buffer size of 64K bytes is recommended. Note that the **–B** option also must ***not*** be used when performing raw I/O to the internal miniature flexible disc drive (HP 9130K), if the I/O requires more than one volume.

EXAMPLES

The first example below copies the contents of a directory into an archive; the second duplicates a directory hierarchy:

```
ls | cpio –o >/dev/mt0

cd olddir
find . –print | cpio –pdl newdir
```

The trivial case "find . –print | cpio –oB >/dev/rmt0" can be handled more efficiently by:

```
find . –cpio /dev/rmt0
```

SEE ALSO

ar(1), find(1), tar(1), tcio(1), cpio(5).

WARNING

Do not redirect the output of *cpio* to a named *cpio* archive file which resides in the same directory as the original files which are part of that *cpio* archive. This can cause loss of data.

If data has been written out to a medium using a raw device file, then a raw device file must be used in reading that data back in.

BUGS

Path names are restricted to 128 characters. If there are too many unique linked files, the program runs out of memory to keep track of them and, thereafter, linking information is lost. Only the super-user can copy special files.

Cpio tapes written on HP machines with the **–ox[c]** options can mislead (non-HP) versions of *cpio* which do not support the **–x** option. If a non-HP (and non-Bell) version of *cpio* happens to be modified so that (HP) *cpio* recognizes it as a device special file, a spurious device file could be created.

If /dev/tty is not accessible, *cpio* issues a complaint, or refuses to work.

The **–pd** option will not create the directory typed on the command line.

The –**idr** option will not make empty directories.

Cpio will fail while restoring files from a backup tape (**cpio –i**) if the following conditions are met:

> your working directory during the restore is **not** the root directory (/), **and** the files being restored have multiple links, **and** their path names begin with slash (/).

If these conditions are met, the following occurs:

(1) The first file on the backup tape is restored correctly;
(2) The second file is removed, and the restore fails.

Note that the second file is removed before the restore fails!

Cpio then writes the message "Cannot link *file1* & *file2*" to *stderr*, but also writes "*file1* linked to *file2*" on *stdout*, as if everything went fine. The correct message is that written to *stderr*.

There are two work-arounds for this bug, either of which will solve the problem. The first is to make sure that your working directory is the root directory during the restore process. The second is to use relative file names (path names not beginning with slash) in your backup.

NAME

cpp – C language preprocessor

SYNOPSIS

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

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Cpp is the C language preprocessor which is invoked as the first pass of any C compilation using the *cc*(1) command. Its purpose is to process **include** and conditional compilation instructions, and macros. The output of *cpp* is designed to be in a form acceptable as input to the next pass of the C compiler. As the C language evolves, *cpp* and the rest of the C compilation package will be modified to follow these changes. Therefore, the use of *cpp* other than in this framework is not suggested. The preferred way to invoke *cpp* is through the *cc*(1) command, since the functionality of *cpp* may someday be moved elsewhere.

Cpp optionally accepts two file names as arguments. *Ifile* and *ofile* are respectively the input and output for the preprocessor. They default to standard input and standard output if not supplied.

The following *option*s are recognized:

–C By default, *cpp* strips C-style comments. If the **–C** option is specified, all comments (except those found on *cpp* directive lines) are passed along.

–D*name*
–D*name* = *def*
Define *name* as if by a **#define** directive. If no = *def* is given, *name* is defined as 1.

–H *nnn* Change the internal macro definition table to be *nnn* bytes in size. The macro symbol table is increased proportionally. The default table size is 36 000 bytes. This option serves to eliminate the "too many defines" and "too much defining" errors.

–I*dir* Change the algorithm for searching for **#include** files whose names do not begin with / to look in *dir* before looking in the directories on the standard list. Thus, **#include** files whose names are enclosed in " " will be searched for first in the directory of the *ifile* argument, then in directories named in **–I** options, and last in directories on a standard list. For **#include** files whose names are enclosed in < >, the directory of the *ifile* argument is not searched. However, the directory *dir* is searched.

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

–T Forces *cpp* to use only the first eight characters for distinguishing different preprocessor names. This behavior is the same as previous preprocessors with respect to the length of names, and is included for backward compatibility.

–U*name*
Remove any initial definition of *name*, where *name* is a reserved symbol that is predefined by the particular preprocessor. The current list of these possibly reserved symbols includes:

operating system:	mert, ibm, gcos, os, tss, unix
hardware:	hp9000s500, hp9000s200, interdata, pdp11, u370, u3b, vax
UNIX System variant:	RES, RT, TS, PWB

Two special names are understood by *cpp*. The name **__LINE__** is defined as the current line number (as a decimal integer) as known by *cpp*, and **__FILE__** is defined as the current file name (as a C string) as known by *cpp*. They can be used anywhere (including in macros) just as any other defined name.

All *cpp* directives start with lines begun by #. The directives are:

#define *name token-string*
Replace subsequent instances of *name* with *token-string* (*token-string* may be null).

#define *name*(*arg*, ... , *arg*) *token-string*
Replace subsequent instances of *name*, followed by (, a list of comma separated tokens, and a), by *token-string*, where each occurrence of an *arg* in the *token-string* is replaced by the corresponding token in the comma separated list. Note that there must be no space between *name* and (.

#undef *name*
Cause the definition of *name* (if any) to be forgotten from now on.

#include "*filename*"
#include <*filename*>
Include at this point the contents of *filename* (which will then be run through *cpp*). When the <*filename*> notation is used, *filename* is only searched for in the standard places. See the **–I** option above for more detail.

#line *integer-constant* "*filename*"
Cause *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 from which it comes. If "*filename*" is not given, the current file name is unchanged.

#endif <*text*>
Ends a section of lines begun by a test directive (**#if**, **#ifdef**, or **#ifndef**). Each test directive must have a matching **#endif**. Any *text* occurring on the same line as the **endif** is ignored and thus may be used to mark matching **if-endif** pairs, making it easier to match up **endif**s with their associated **if**s.

#ifdef *name*
The lines following will appear in the output if and only if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**, or if it is a currently defined reserved symbol.

#ifndef *name*
The lines following will not appear in the output if and only if *name* has been the subject of previous **#define** without being the subject of an intervening **#undef**.

#if *constant-expression*
Lines following will appear in the output if and only if the *constant-expression* evaluates to non-zero. All binary non-assignment C operators, the **?:** operator, the unary –, **!**, and ~ operators are all 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. Thus, if lines previous to this directive are ignored, the following lines will appear in the output, and vice-versa.

The test directives and the possible **#else** directives can be nested.

Cpp supports *name*s up to 255 characters long.

FILES

/usr/include	standard directory for **#include** files

SEE ALSO

cc(1).

DIAGNOSTICS

The error messages produced by *cpp* are intended to be self-explanatory. The line number and filename where the error occurred are printed along with the diagnostic.

NAME

crypt – encode/decode files

SYNOPSIS

crypt [password]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Crypt reads from the standard input and writes on the standard output. The *password* is a key that selects a particular transformation. If no *password* is given, *crypt* demands a key from the terminal and turns off printing while the key is being typed in. *Crypt* encrypts and decrypts with the same key:

```
crypt key <clear >cypher
crypt key <cypher | pr
```

will print the clear.

Files encrypted by *crypt* are compatible with those treated by the editor *ed* in encryption mode.

The security of encrypted files depends on three factors: the fundamental method must be hard to solve; direct search of the key space must be infeasible; "sneak paths" by which keys or clear text can become visible must be minimized.

Crypt implements a one-rotor machine designed along the lines of the German Enigma, but with a 256-element rotor. Methods of attack on such machines are known, but not widely; moreover the amount of work required is likely to be large.

The transformation of a key into the internal settings of the machine is deliberately designed to be expensive, i.e. to take a substantial fraction of a second to compute. However, if keys are restricted to (say) three lower-case letters, then encrypted files can be read by expending only a substantial fraction of five minutes of machine time.

Since the key is an argument to the *crypt* command, it is potentially visible to users executing *ps*(1) or a derivative. To minimize this possibility, *crypt* takes care to destroy any record of the key immediately upon entry. The choice of keys and key security are the most vulnerable aspect of *crypt*.

FILES

/dev/tty for typed key

SEE ALSO

ed(1), makekey(8).

BUGS

If output is piped to *nroff*(1) and the encryption key is *not* given on the command line, *crypt* can leave terminal modes in a strange state (see *stty*(1)).

Due to legal restrictions, *crypt* is currently not available on systems that may be sold outside the United States.

NAME

csh – a shell (command interpreter) with C-like syntax

SYNOPSIS

csh [**–cefinstvVxX**] [command file] [argument list ...]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: Berkeley 4.1

DESCRIPTION

Csh is a command language interpreter incorporating a command history buffer and a C-like syntax.

The command options are interpreted as follows:

–c The first argument in the *argument list* is a command file. Commands are read and executed from that file. The command file must exist. Any arguments in the argument list are copied into the shell variable *argv*.

–e The shell exits if any invoked command terminates abnormally or yields a non-zero exit status.

–f Suppress execution of the *.cshrc* file in your home directory, thus speeding up shell start-up time.

–i Forces *csh* to respond interactively when called from a device other than a computer terminal, such as another computer. *Csh*'s normally responds non-interactively. If *csh* is called from a computer terminal, it always responds interactively, no matter which options are selected.

–n This option causes commands to be parsed, but **not** executed. This may be used in syntactic checking of shell scripts. All substitutions are performed (history, command, alias, etc.).

–s This option allows you to redirect input from a command.

–t A single line of input is read and executed. This option combines the **–n** option described above with automatic execution of the command.

–v This option causes the *verbose* shell variable to be set. This causes command input to be echoed to your standard output device after history substitutions are made.

–x This option causes the *echo* shell variable to be set. This causes all commands to be echoed to the standard output immediately before execution.

–V This option causes the *verbose* variable to be set before *.cshrc* is executed. This means all *.cshrc* commands are also echoed to the standard output.

–X This option causes the *echo* variable to be set before *.cshrc* is executed. This means all *.cshrc* commands are also echoed to the standard output.

After processing the command options, if arguments remain in the argument list, and the **–c**, **–i**, **–s**, or **–t** options were not specified, the first remaining argument is taken as the name of a file of commands to be executed.

COMMANDS

A simple command is a sequence of words, the first of which specifies the command to be executed. A sequence of simple commands separated by vertical bar (|) characters forms a pipeline. The output of each command in a pipeline is made the input of the next command in the pipeline. Sequences of pipelines may be separated by semicolons (;), and are then executed sequentially. A sequence of pipelines may be executed in background mode by following the last entry with an ampersand (&) character.

Any pipeline may be placed in parenthesis to form a simple command which in turn may be a component of another pipeline. It is also possible to separate pipelines with "||" or "&&" indicating, as in the C language, that the second pipeline is to be executed only if the first fails or succeeds, respectively.

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. The build-in commands are:

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*. Command and filename substitution are performed on *wordlist*. *Name* cannot be **alias** or **unalias**.

alloc This comand shows the amount of dynamic core in use, broken down into used and free core, and the address of the last location in the heap. With an argument, *alloc* 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.

break Causes execution to resume after the *and* 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 *directory_name*
chdir
chdir *directory_name*

Change the shell's current working directory to *directory_name*. If no argument is given, then *directory_name* defaults to your home directory.

If *directory_name* is not found as a subdirectory of the current working directory (and does not begin with "/", "./" or "../"), then each component of the variable *cdpath* is checked to see if it has a subdirectory *directory_name*. Finally, if all else fails, *csh* treats *directory_name* as a shell variable. If its 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 other *case* labels.

dirs Prints the directory stack; the top of the stack is at the left; the first directory in the stack is 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 arguments ...
(As in *sh*(1).) The arguments are read as input to the shell and the resulting command(s) executed. This is usually used to execute commands generated as the result of command or variable substitution, since parsing occurs before these substitutions.

exec command
The specified command is executed in place of the current shell.

exit
exit (*expression*)
The shell exits either with the value of the *status* variable (first form) or with the value of the specified *expression* (second form).

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* terminates it prematurely. When this command is read from the terminal, the loop is read once, prompting with '?' before any statements in the loop are executed. If you make a mistake while typing in a loop at the terminal you can then 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 perform filename expansion on a list of words.

goto *word*
The specified *word* is filename and command expanded to yield a string of the form 'label'. The shell rewinds its input as much as possible and searches for a line of the form 'label:' possibly preceded by blanks or tabs. Execution continues after the specified line.

hashstat
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*
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.

if (*expression*) *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 *expression* is false, when *command* is **not** executed (this is a bug).

if (*expression1*) **then**
...
else if (*expression2*) **then**
...
else
...

endif If the specified *expression1* is true then the commands to the first *else* are executed; otherwise if *expression2* 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 [**–l**]
Lists the active jobs; the **–l** option lists process id's in addition to the normal information.

kill % *job*
kill – *sig %job ...*
kill *pid*
kill –*sig pid ...*

kill –l Sends either the TERM (terminate) signal or the specified signal to the specified jobs or processes. Signals are either given by number or by names (as given in */usr/include/signal.h*, stripped of the "SIG" prefix – see *signal*(2)). The signal names are listed by **kill –l**. There is no default, so 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 is sent a CONT (continue) signal as well.

login Terminates a login shell, replacing it with an instance of */bin/login*. This is one way to log off, included for compatibility with *sh*(1).

logout Terminates a login shell. Especially useful if *ignoreeof* is set.

newgrp Changes the group identification of the caller; for details see *newgrp*(1). A new shell is executed by *newgrp* so that the current shell environment is lost.

nice
nice +*number*
nice *command*
nice +*number command*
The first form sets the *nice* (run command priority) for this shell to 4 (the default). The second form sets the priority to the given *number*. The final two forms run *command* at priority 4 and *number* respectively. The super-user may raise the priority by specifying negative niceness using **nice –***number .... Command* is always executed in a sub-shell, and the restrictions place on commands in simple **if** statements apply.

nohup [*command*]
Without an argument, *nohup* can be used in shell scripts to cause hangups to be ignored for the remainder of the script. With an argument, causes the specified *command* to be run with hangups ignored. All processes executed in the background with **&** are effectively *nohup*'ed.

notify [*%job ...*]
Causes the shell to notify the user asynchronously when the status of the current (no argument) or specified jobs changes; normally notification is presented before a prompt. This is automatic if the shell variable *notify* is set.

onintr [–] [*label*]
Controls the action of the shell on interrupts. With no arguments, *onintr* restores the default action of the shell on interrupts, which is to terminate shell scripts or to return to the terminal command input level. If – is specified, causes all interrupts to be ignored. If a *label* is given,

causes the shell to execute a **goto label** when an interrupt is received or a child process terminates because it was interrupted.

If the shell is running in the background and interrupts are being ignored, *onintr* has no effect; interrupts continue to be ignored by the shell and all invoked commands.

popd [+*n*]

Pops the directory stack, returning to the new top directory. With an argument, discards the *n*th entry in the stack. The elements of the directory stack are numbered from 0 starting at the top.

pushd [*name*] [+*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 **set** 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 filename expanded.

These arguments may be repeated to set multiple values in a single *set* command. Note, however, that variable expansion happens for all arguments before any setting occurs.

setenv *name value*

Sets the value of environment variable *name* to be *value*, a single string. The most commonly used environment variables USER, TERM, and PATH are automatically imported to and exported from the *csh* variables *user*, *term*, and *path*; there is no need to use *setenv* for these.

shift [*variable*]

With no argument, the members of *argv* are shifted to the left, discarding *argv*[1]. An error occurs if *argv* is not set or has less than two strings assigned to it. With an argument, *shift* performs the same function on the specified *variable*.

source *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. Input during *source* commands is **never** placed on the history list.

stop [%*job* . . .]

Stops the current (no argument) or specified job which is executing in the background.

switch (*string*)
case *str1*:
...
breaksw
...
default:
...
breaksw

endsw Each *case* label (*str1*) is successively matched against the specified *string* which is first command and filename expanded. The file metacharacters *, ?, and [...] may be used in the *case* labels, which are variable expanded. If none of the labels match before a **default** label is found, then the execution begins after the **default** label. Each *case* label and the **default** label must appear at the beginning of a line. The command *breaksw* causes execution to continue after the *endsw*. Otherwise, control may fall through *case* labels and **default** labels as in C. If no label matches and there is no default, execution continues after the *endsw*.

time [*command*]
With no argument, a summary of time used by this shell and its children is printed. If an argument is 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 [*value*]
The current file creation mask is displayed (no argument) or set to the specified *value*. The mask is given in octal. Common values for the mask are 002, which gives all permissions to the owner and group, and read and execute permissions to all others, or 022, which gives all permissions to the owner, and read permission only to the group and all others.

unalias *pattern*
All aliases whose names match the specified *pattern* are discarded. Thus, all aliases are removed by **unalias** *. No error occurs if *pattern* is omitted.

unhash
Use of the internal hash table to speed location of executed programs is disabled.

unset *pattern*
All variables whose names match the specified *pattern* are removed. Thus, all variables are removed by **unset** *; this has noticeably distasteful side-effects. No error occurs if *pattern* is omitted.

unsetenv *pattern*
Removes all variables whose names match the specified *pattern* from the environment. See also the *setenv* command above and *printenv*(1).

wait All background jobs are waited for. If the shell is interactive, then an interrupt can disrupt the wait, at which time the shell prints names and job numbers of all jobs known to be outstanding.

while (*expression*)
...
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.) If the input is a terminal (i.e. not a script), prompting occurs the first time through the loop as for the *foreach* statement.

@
@ *name* = *expression*
@ *name*[*index*] = *expression*

The first form prints the values of all the shell variables. The second form sets the specified *name* to the value of *expression*. If the expression contains "<", ">", "&" or "|", then at least this part of the expression must be placed within parentheses. The third form assigns the value of *expression* 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. White space may optionally separate the *name* from the assignment operator. However, spaces are mandatory in separating components of *expression* which would otherwise be single words.

Special postfix "++" and "--" operators increment and decrement *name*, respectively (i.e. @ **i**++).

Non-Built-In Command Execution

When a command to be executed is not a built-in command, the shell attempts to execute the command via *exec*(2). Each word in the variable *path* names a directory in which the shell attempts to find the command (if the command does not begin with "/"). If neither –**c** nor –**t** is given, the shell hashes the names in these directories into an internal table so that an *exec* is attempted only in those directories where the command might possibly reside. 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 –**c** or –**t** was given, or if any directory component of *path* does not begin with a '/', the shell concatenates the directory name and 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)

prints the *home* directory but leaves you where you were.

cd ; pwd

does the same thing, but 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 file, then it is assumed to be a shell script, and a new shell is spawned to read it.

If there is an *alias* for *shell* then the words of the alias are inserted at the beginning of the argument list to form the shell command. The first word of the *alias* should be the full path name of the shell (e.g. "$shell"). Note that this is a special, late-occurring case of *alias* substitution, which inserts words into the argument list without modification.

Command Substitution

Command substitution is indicated by a command enclosed in single quotes (' . . . '). The output from such a command is normally broken into separate words at blanks, tabs and newlines, with null words being discarded, this text then replacing the original string. Within double quotes, only newlines force new words; blanks and tabs are preserved.

In any case, the single final newline does not force a new word. Note that it is thus possible for a command substitution to yield only part of a word, even if the command outputs a complete line.

History Substitutions

History substitutions enable you to use words from previous commands as portions of new commands, repeat commands, repeat arguments of a previous command in the current command, and fix spelling mistakes in the previous command.

History substitutions begin with an exclamation point (!). Substitutions may begin anywhere in the input stream, but may **not** be nested. The exclamation point can be preceded by a backslash to prevent its special meaning. For convenience, an exclamation point is passed to the parser unchanged when it is followed by a blank, tab, newline, equal sign or right parenthesis. Any input line which contains history substitution is echoed on the terminal before it is executed for verification.

Commands input from the terminal which consist of one or more words are saved on the history list. The history substitutions reintroduce sequences of words from these saved commands into the input stream. The number of previous commands saved is controlled by the *history* variable. The previous command is always saved, regardless of its value. Commands are numbered sequentially from 1.

You can refer to previous events by event number (such as **!10** for event 10), relative event location (such as **!–2** for the second previous event), full or partial command name (such as **!d** for the last event using a command with initial character d), and string expression (such as **!?mic?** referring to an event containing the characters **mic**).

These forms, without further modification, simply reintroduce the words of the specified events, each separated by a single blank. As a special case, **!!** is a re-do; it refers to the most previous command.

To select words from a command you can follow the event specification by a colon (:) and a designator for the desired words. The words of a input line are numbered from zero. The basic word designators are:

0 selects the first word (i.e. the command name itself).

n selects the *n*th word.

$ selects the last word.

a-*b* selects the range of words from *a* to *b*. Special cases are **–y**, which is an abbreviation for "word 0 through word *y*", and **x–**, which stands for "word *x* up to, but not including, word **$**".

***** indicates the range from the second word to the last word.

% used with a search sequence to substitute the immediately preceding matching word.

The colon separating the command specification from the word designator can be omitted if the argument selector begins with a ^, $, *, –, or %.

After each word designator, you can place a sequence of modifiers, each preceded by a colon. The following modifiers are defined:

h Use only the first component of a pathname by removing all following components.

r Use the root file name by removing any trailing suffix (.xxx).

e Use the file name's trailing suffix (.xxx) by removing the root name.

s/*l*/*r* substitute the value of *r* for the value *l* in the indicated command.

t Use only the final file name of a pathname by removing all leading pathname components.

& Repeat the previous substitution.

p Print the new command but do not execute it.

q Quote the substituted words, preventing further substitutions.

x Like **q**, but break into words at blanks, tabs and newlines.

g **g**lobal command; used as a prefix to cause the specified change to be made globally (all words in the command are changed).

Unless preceded by a **g**, the modification is applied only to the first modifiable word. You get an error if a substitution is attempted and cannot be completed (i.e. if you have a history buffer of 10 commands and ask for a substitution of **!11**).

The left hand side of substitutions are not regular expressions in the sense of the HP-UX editors, but rather strings. Any character may be used as the delimiter in place of a slash (/); a backslash 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 newline follows immediately, as may the trailing **?** in a contextual scan.

A history reference may be given without an event specification (e.g. **!$**). In this case the reference is to the previous command unless a previous history reference occurred on the same line, in which case this form repeats the previous reference. Thus

!?foo?^ !$

gives the first and last arguments from the command matching "?foo?".

A special abbreviation of a history reference occurs when the first non-blank character of an input line is a caret (^). This is equivalent to "!:s^", providing a convenient shorthand for substitutions on the text of the previous line. Thus "^lb^lib" fixes the spelling of "lib" in the previous command.

Finally, a history substitution may be surrounded with curly braces { } if necessary to insulate it from the characters which follow. Thus, after

ls –ld ~paul

we might execute **!{l}a** to do

ls –ld ~paula

while **!la** would look for a command starting with "la".

Quoting with Single and Double Quotes

The quotation of strings by backslash (\) and double quotes (") can be used to prevent all or some of the remaining substitutions. Strings enclosed in backslashes are protected from any further interpretation. Strings enclosed in double quotes are still variable and command expanded as described below.

In both cases the resulting text becomes (all or part of) a single word; only in one special case does a double-quoted string yield parts of more than one word; single-quoted strings never do.

Alias Substitution

The shell maintains a list of aliases which can be established, displayed and modified by the *alias* and *unalias* commands. After a command line is scanned, it is parsed into distinct commands and the first word of each command, left-to-right, is checked to see if it has an alias. If it does, then the text which is the alias for that command is reread with the history mechanism available as though that command were the previous input line. The resulting words replace the command and argument list. If no reference is made to the history list, then the argument list is left unchanged.

Thus, if the alias for **ls** is **ls –l**, the command **ls /usr** maps to **ls –l /usr**, leaving the argument list undisturbed. Similarly, if the alias for **lookup** was **grep !^ /etc/passwd, then lookup bill** maps to **grep bill /etc/passwd**.

If an alias is found, the word transformation of the input text is performed and the aliasing process begins again on the re-formed input line. Looping is prevented if the first word of the new text is the same as the old by flagging it to prevent further aliasing. Other loops are detected and cause an error.

Note that the mechanism allows aliases to introduce parser metasyntax. Thus we can execute

alias print 'pr \!* | lpr'

to make a command which uses *pr*(1) to print its arguments on the line printer.

Expressions

A number of the built-in commands 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 (shown in order of increasing precedence):

```
|| && | ^ & == != =~ !~ <= >= < > << >> + - * / % ! ~ ( )
```

The following list shows the grouping of these operators. The precedence decreases from top to bottom in the list:

```
* / %
+ -
<< >>
<= >= < >
== != =~ !~
```

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 *'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. These components should be surrounded by spaces except when adjacent to components of expressions which are syntactically significant to the parser – &, |, <, >, (, and).

Also available in expressions as primitive operands are command executions enclosed in curly braces { } and file enquiries of the form "*–l filename*", 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 *filename* is command and filename expanded and then tested to see if it has the specified relationship to the real user. If the file does not exist or is inaccessible then all enquiries return false (0). Command executions succeed, returning true, if the command exits with status 0; otherwise they fail, returning false. If more detailed status information is required then the command should be executed outside of an expression and the variable *status* examined.

Control of the Flow (one)

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 parts of 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 succeed on non-seekable inputs.)

Signal Handling

The shell normally ignores *quit* signals. Jobs running in background mode 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 Line Parsing

Csh splits input lines into words at blanks and tabs. The following exceptions (parser metacharacters) are considered separate words:

&	ampersand;
|	vertical bar;
;	semicolon;
<	less-than sign;
>	greater-than sign;
(	left parenthesis;
)	right parenthesis;
&&	double ampersand;
||	double vertical bar;
<<	double less-than sign;
>>	double greater-than sign;

The backslash (\) removes the special meaning of these parser metacharacters. A parser metacharacter preceded by a backslash is interpreted as its ASCII value. A newline character (ASCII 10) preceded by a backslash is equivalent to a blank.

Strings enclosed in single or double quotes form parts of a word. Metacharacters in these strings, including blanks and tabs, do not form separate words. Within pairs of backslashs or quotes, a newline preceded by a backslash gives a true newline character.

When the shell's input is not a terminal, the pound sign (#) introduces a comment terminated by a newline.

CSH VARIABLES

Csh maintains a set of variables. Each variable has a value equal to zero or more strings (words). Variables have names consisting of up to 20 letters and digits starting with a letter. The underscore character is considered a letter. The value of a variable may be displayed and changed by using the *set* and *unset* commands. Some of the variables are boolean, that is, the shell does not care what their value is, only whether they are set or not.

Some operations treat variables numerically. The at sign (@) command permits numeric calculations to be performed and the result assigned to a variable. The null string is considered to be zero, and any subsequent words of multi-word values are ignored.

After the input line is aliased and parsed, and before each command is executed, variable expansion is performed keyed by the dollar sign ($) character. Variable expansion can be prevented by preceding the dollar sign with a backslash character (\) except within double quotes (") where substitution **always** occurs. Variables are never expanded if enclosed in single quotes. Strings quoted by single quotes are interpreted later (see *Command Substitution*) so variable substitution does not occur there until later, if at all. A dollar sign 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.

Unless enclosed in double quotes or given the **:q** modifier, the results of variable substitution may eventually be command and filename substituted. Within double quotes, a variable whose value consists of multiple words expands to a portion of a single word, with the words of the variable's value

separated by blanks. When the **:q** modifier is applied to a substitution, the variable expands to multiple words with each word separated by a blank and quoted to prevent later command or filename substitution.

The following metasequences are provided for introducing variable values into the shell input. Except as noted, it is an error to reference a variable which is not set.

$variable_name
${variable_name}

When interpreted, this sequence is replaced by the words of the value of the variable *variable_name*, each separated by a blank. Braces insulate *variable_name* from following characters which would otherwise be interpreted to be part of the variable name itself.

If *variable_name* is not a *csh* variable, but is set in the environment, then that value is used. Non-*csh* variables cannot be modified as shown below.

$variable_name[selector]
${variable_name[selector]}

This modification allows you to select only some of the words from the value of *variable_name*. The selector is subjected to variable substitution and may consist of a single number or two numbers separated by a dash. The first word of a variable's 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 the total number of words in the variable ($#variable_name). An asterisk metacharacter used as a selector selects all words.

$#variable_name
${#variable_name}

This form gives the number of words in the variable. This is useful for forms using a [*selector*] option.

$0 This form substitutes the name of the file from which command input is being read. An error occurs if the filename is not known.

$number
${number}

This form is equivalent to an indexed selection from the variable *argv* ($argv[number]).

$* This is equivalent to selecting all of *argv* ($argv[*]).

The modifiers **:h**, **:t**, **:r**, **:q** and **:x** may be applied to the substitutions above, as may **:gh**, **:gt** and **:gr**. If curly braces { } appear in the command form then the modifiers must appear within the braces. *The current implementation allows only one* : *modifier on each* $ *expansion.*

The following substitutions may not be modified with **:** modifiers.

$?variable_name
${?variable_name}

Substitutes the string **1** if *variable_name* is set, **0** if it is not. *Variable_name* must be a boolean variable.

$?0 Substitutes **1** if the current input filename is known, **0** if it is not.

$$ Substitutes the (decimal) process number of the (parent) shell.

$< Substitutes a line from the standard input, with no further interpretation thereafter. It can be used to read from the keyboard in a shell script.

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 (initial execution of *csh*); these variables are not modified unless modified explicitly by the user.

Csh copies the HP-UX environment variable USER into the shell variable *user*, the environment variable TERM into *term*, the environment variable HOME into *home*, and PATH into *path*. *Csh* copies these values back into the environment whenever the *csh* variables are reset. The HP-UX environment variable PATH could be set in the shell script **.login**, except that commands through *net*(1) would not see that value.

argv This variable is set to the arguments of the *csh* command statement. It is from this variable that positional parameters are substituted, i.e. **$1** is replaced by **$argv[1]**, etc.

cdpath This variable gives a list of alternate directories searched to find subdirectories in *chdir* commands.

cwd This variable contains the absolute pathname of your current working directory. Whenever you change directories (using *cd*), this variable is updated.

echo This variable is set by the **–x** command line option. If set, all built-in commands and their arguments are echoed to your standard output device just before being executed. Built-in commands are echoed before command and filename substitution, since these substitutions are then done selectively. For non-built-in commands, all expansions occur before echoing.

history This variable is used to create your command history buffer and to set its size. If this variable is not set, you have no command history and can do no history substitutions. Very large values of **history** may run your shell out of memory. Values of 10 or 20 are normal. All commands, executable or not, are saved in your command history buffer.

home This variable contains the absolute pathname to your home directory. **Home** is initialized from the HP-UX environment. The filename expansion of tilde (~) refers to this variable.

ignoreeof If set, *csh* ignores end-of-file characters from input devices which are terminals. This prevents your processes from accidentally being killed by control-D's.

mail This variable contains a list of the files where *csh* checks for your mail. *Csh* periodically (default is 10 minutes) checks this variable after a command completion which results in a prompt. If the variable contains a filename that has been modified since the last check (resulting from mail being put in the file), *csh* prints *You*have*new*mail.

If the first word of the value of *mail* is numeric, that number specifies a different mail checking interval in seconds.

If multiple mail files are specified, then the shell says *New*mail*in*file_name, where *file_name* is the file containing the mail.

noclobber This variable places restrictions on output redirection to insure that files are not accidentally destroyed, and that commands using append redirection (>>) refer to existing files.

noglob If set, filename expansion is inhibited. This is most useful in shell scripts which are not dealing with filenames, or after a list of filenames has been

obtained and further expansions are not desirable.

nonomatch If set, it is no longer an error for a filename expansion to not match any existing files. If there is no match, the primitive pattern is returned. It is still an error for the primitive pattern to be malformed, i.e. '**echo [**' still gives an error.

notify If set, *csh* notifies you immediately (through your standard output device) of background job completions. The default is **unset** (indicate 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 your current working directory. If there is no *path* variable then only full path names can be executed. When *path* is not set and when users do not specify full pathnames, *csh* searches for the command through the directories **.** (your current directory), */bin*, */lbin*, */usr/bin*, and */usr/lbin*. A *csh* which is given neither the **–c** nor the **–t** option normally hashes 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 is necessary to execute *rehash* for *csh* to access these new commands.

prompt This variable lets you select your own prompt character string. The prompt is printed before each command is read from an interactive terminal input. If a **!** appears in the string it is replaced by the current command history buffer event number unless a preceding \ is given. The default prompt is the percent sign (%) for users and the pound sign (#) for the super-user.

shell This variable contains the name of the file in which the *csh* program resides. This variable is used in forking shells to interpret files which have their execute bits set, but which are not executable by the system. (See the description of *Non-built-In Command Execution*).

status This variable contains the status value returned by the last command. If the command terminated abnormally, then 0200 is added to the status variable's value. Built-in commands which terminated abnormally return exit status **1**, and all other built-in commands set status to **0**.

time This variable contains a numeric value which controls the automatic timing of commands. If set, then *csh* prints, for any command which takes more than the specified number of cpu seconds, a line of information to your standard output device giving user, system, and real execution times plus a utilization percentage. The utilization percentage is the ratio of user plus system times to real time. This message is printed after the command finishes exection.

verbose This variable is set by the **–v** command line option. If set, the words of each command are printed on the standard output device after history substitutions have been made.

Command and Filename Substitution

The remaining substitutions, command and filename substitution, are applied selectively to the arguments of built-in commands. This means that portions of expressions which are not evaluated are not subjected to these expansions. For commands which are not internal to the shell, the command name is substituted separately from the argument list. This occurs very late, after input-output redirection is performed, and in a child of the main shell.

Filename Substitution

If a word contains any of the characters *, ?, [, or {, or begins with the character ~, then that word is a candidate for filename substitution, also known as *globbing*. This word is then regarded as a pattern, and replaced with an alphabetically sorted list of file names which match the pattern. In a list of words specifying filename substitution it is an error 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, while the characters ~ and { are more like abbreviations.

In matching filenames, the character . at the beginning of a filename or immediately following a /, as well as the character / itself, 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 the square brackets, a pair of characters separated by – matches any character lexically between and including the two.

The tilde character (~) at the beginning of a filename is used to refer to home directories. By itself, the tilde expands to your 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 **/users/ken** and **~ken/chmach** to **/usr/ken/chmach**. If the ~ is followed by a character other than a letter or /, or appears somewhere other than 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,

might expand to

(Note that "memo" was not sorted with the results of matching ***box**.) As a special case, {, }, and {} are passed undisturbed.

Input/Output

The standard input and standard output of a command may be redirected with the following syntax:

< *name* Open file *name* (which is first variable, command and filename expanded) as the standard input.

<< *word*

Read the shell input up to a line which is identical to *word*. *Word* is not subjected to variable, filename or command substitution, and each input line is compared to *word* before any substitutions are done on this input line. Unless a quoting \, ", ', or ` appears in *word*, variable and command substitution is performed on the intervening lines, allowing \ to quote $ and \. Commands which are substituted have all blanks, tabs, and newlines preserved, except for the final newline which is dropped. The resultant text is placed in an anonymous temporary file which is given to the command as standard input.

> *name*
>! *name*
>& *name*
>&! *name*

The file *name* is used as standard output. If the file does not exist then it is created; if

the file exists, it is truncated, and its previous contents are lost.
If the variable *noclobber* is set, then the file must not exist or be a character special file (e.g. a terminal or */dev/null*) or an error results. This helps prevent accidental destruction of files. In this case the exclamation point (!) forms can be used to suppress this check.
The forms involving the ampersand character (&) route the standard error into the specified file as well as the standard output. *Name* is expanded in the same way as < input filenames are.

>> *name*
>>& *name*
>>! *name*
>>&! *name*

Uses file *name* as standard output like >, but appends output to 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, it is 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 executed from a shell script 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 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 is notified.

Diagnostic output may be directed through a pipe with the standard output. Simply use the form "|&" rather than just "|".

CSH UTILITIES

File Name Completion

In typing file names as arguments to commands, it is no longer necessary to type a complete name, only a unique abbreviation is necessary. When you want the system to try to match your abbreviation, press your ESCAPE key. The system then completes the filename for you, echoing the full name on your terminal. If the abbreviation doesn't match an available filename, the terminal's bell is sounded. The file name may be partially completed if the prefix matches several longer file names. In this case, the name is extended up to the ambiguous deviation, and the bell is sounded.

File name completion works equally well when other directories are addressed. In addition, the tilde (~) convention for home directories is understood in this context.

Viewing a File or Directory List

At any point in typing a command, you may request "what files are available" or "what files match my current specification". Thus, when you have typed:

% cd ~speech/data/bench/fritz/

you may wish to know what files or subdirectories exist (in **~speech/data/bench/fritz**), without aborting the command you are typing. Typing **Control-D** or **Control-F** at this point lists the files available. The files are listed in multicolumn format, sorted column-wise. Directories and executable files are indicated with a trailing / and *, respectively. Once printed, the command is re-echoed for you to complete. Additionally, you may want to know which files match a prefix, the current file specification so far. If you had typed:

% cd ~speech/data/bench/fr

followed by a **control-D**, all files and subdirectories whose prefix was "fr" in the directory **~speech/data/bench** would be printed. Notice that the example before was simply a degenerate case

of this with a null trailing file name. (The null string is a prefix of all strings.) Notice also that a trailing slash is required to pass to a new sub-directory for both file name completion and listing. Note that the degenerate case

% ~^D

prints a full list of login names on the current system.

Command Name Recognition

Command name recognition and completion works in the same manner as file name recognition and completion above. The current value of the environment variable PATH is used in searching for the command. For example

% newa [Control]-[D]

might expand to

% newaliases

Also,

% new [Control]-[D]

lists all commands (along PATH) that begin with "new". As an option, if the shell variable *listpathnum* is set, then a number indicating the index in PATH is printed next to each command on a [Control]-[D] listing.

Autologout

A new shell variable has been added called *autologout*. If the terminal remains idle (no character input) at the shell's top level for a number of minutes greater than the value assigned to *autologout*, you are automatically logged off. The *autologout* feature is temporarily disabled while a command is executing. The initial value of *autologout* is 60. If unset or set to 0, *autologout* is entirely disabled.

Sanity

The shell now restores your terminal to a sane mode if it appears to return from some command in raw, cbreak, or noecho mode.

Saving Your History Buffer

Csh has the facility to save your history list between login sessions. If the shell variable *savehist* is set to a number, then that number of command events from your history list are saved. For example, placing the line

set history = 10 savehist = 10

in your **.cshrc** file maintains a history buffer of length 10 and saves the entire list when you logout. When you log back in, the entire buffer is restored. The commands are saved in the file **.history** in your login directory.

FILES

/bin/sh	standard shell, for shell scripts not starting with a #;
/tmp/sh*	temporary file for <<;
/etc/passwd	source of home directories for ~name.

LIMITATIONS

Words can be no longer than 1024 characters.
The system limits argument lists to 10240 characters.
The number of arguments to a command which involves filename expansion is limited to 1/6'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* substititutions on a single line to 20.

SEE ALSO

sh(1), access(2), exec(2), fork(2), pipe(2), umask(2), wait(2), tty(4), a.out(5), environ(7).

BUGS

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 then immediately executes 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 into a subshell, i.e. (a ; b ; c).

Because of the signal handling required by csh, interrupts are disabled just before a command is executed and restored as the command begins execution. There may be a few seconds delay between when a command is given and when interrupts are recognized.

If you do a kill job command on a pipeline, only the first process in the pipeline is killed.

Control over tty output after processes are started is primitive; perhaps this will inspire someone to work on a good virtual terminal interface. In a virtual terminal interface much more interesting things could be done with output control.

Alias substitution is most often used to clumsily simulate shell procedures; shell procedures should be provided rather than aliases.

Commands within loops, prompted for by ?, are not placed in the *history* list. Control structure should be parsed rather than 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 : modifiers on the output of command substitutions. All and more than one : modifier should be allowed on $ substitutions.

Your terminal type is only examined the first time you attempt recognition.

To list all commands on the system along PATH, enter [SHIFT]-[CNTL]-[D].

The csh metasequence !~ does not work.

NAME

ctags – create a tags file

SYNOPSIS

ctags [options] name ...

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: UCB

DESCRIPTION

Ctags 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) in a group of files. Each line of the tags file contains the function name, the file in which it is defined, and a scanning pattern used to find the function definition. These are given in separate fields on the line, separated by blanks or tabs. Using the *tags* file, *ex* can quickly find these function definitions.

The options are as follows:

–B use backward searching patterns (?...?).

–F use forward searching patterns (/.../) (default).

–a Add the information from the files to the *tags* file. Unlike re-building the *tags* file from the original files, this can cause the same symbol to be entered twice in the *tags* file. This option should be used with **caution**.

–t create tags for typedefs.

–u causes the specified files to be updated in tags. That is, all references to them are deleted, and the new values are appended to the file. This option is rather slow; it is usually faster to simply rebuild the tags file.

–v A page index is prepared on the standard output. This listing contains the function name, file name, and page number within that file (assuming 56 lines pages to match *pr*(1)). Since the output is sorted into lexicographic order, it may be desirable to run the output through **sort –f**, as in

```
ctags –v files | sort –f >index
```

–w suppresses warning diagnostics.

–x causes *ctags* to print a simple function index. This is done by assembling a list of function names, file names on which each function is defined, the line numbers where each function name occurs, and the text of each line. The list is then printed on the standard output.

Files whose name ends in **.c** or **.h** are assumed to be C source files and are searched for C routine and macro definitions. 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.

The tag *main* is treated specially in C programs. The tag formed is created by attaching *M* to the beginning of the file name, 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.

FILES

tags output tags file

SEE ALSO

ex(1), vi(1).

BUGS

Recognition of **functions**, **subroutines** and **procedures** for FORTRAN and Pascal is done is a very simple-minded way.

No attempt is made to deal with block structure. If you have two Pascal procedures in different blocks with the same name, invalid data can result.

The method of deciding whether to look for C or Pascal and FORTRAN functions is unreliable.

It does not know about #ifdefs and Pascal types.

It relies on the input being well formed to detect *typedefs*.

If more than one function definition appears on a single line, only the first definition will be indexed.

NAME

cu – call another UNIX system; terminal emulator

SYNOPSIS

cu [–s*speed*] [–a*acu*] [–l*line*] [**–h**] [**–q**] [**–o**|**–e**] *telno*
cu [–s*speed*] [**–h**] [**–q**] [**–o** | **–e**] **–l***line dir*

HP-UX COMPATIBILITY

Level: Data Communications - HP-UX/STANDARD

Origin: System III

DESCRIPTION

Cu calls up another UNIX system, a terminal, or possibly a non-UNIX system. It manages an interactive conversation with possible transfers of ASCII files. *Speed* gives the transmission speed (110, 150, 300, 1200, 4800, 9600); 300 is the default value. Most modems restrict you to choose between 300 and 1200.

When using a direct-connect line, the **–s** option has no effect. The first line which matches the **–l** option is used, and its speed is taken from L-devices.

The **–a** and **–l** options may be used to specify device names for the ACU and communications line devices. They can be used to override searching for the first available ACU with the right speed.

The **–h** option emulates local echo, supporting calls to other computer systems which expect terminals to be in half-duplex mode.

The **–q** option invokes the use of ENQ/ACK handshake.

The **–e** (**–o**) option designates that even (odd) parity is to be generated for data sent to the remote.

Telno is the telephone number, with equal signs for secondary dial tone or minus signs for delays, at appropriate places. The string **dir** for *telno* must be used for directly connected lines, and implies a null ACU.

Cu will try each line listed in the file **/usr/lib/uucp/L-devices** until it finds an available line with appropriate attributes or runs out of entries. After making the connection, *cu* runs as two processes: the *transmit* process reads data from the standard input and, except for lines beginning with ~, passes it to the remote system; the *receive* process accepts data from the remote system and, except for lines beginning with ~, passes it to the standard output. Normally, an automatic DC3/DC1 protocol is used to control input from the remote so the buffer is not overrun. Lines beginning with ~ have special meanings.

The *transmit* process interprets the following:

~.	terminate the conversation.
~!	escape to an interactive shell on the local system.
~!*cmd . . .*	run *cmd* on the local system (via **sh –c**).
~$*cmd . . .*	run *cmd* locally and send its output to the remote system.
~%take *from* [*to*]	copy file *from* (on the remote system) to file *to* on the local system. If *to* is omitted, the *from* argument is used in both places.
~%put *from* [*to*]	copy file *from* (on local system) to file *to* on remote system. If *to* is omitted, the *from* argument is used in both places.
~~*. . .*	send the line ~*. . .* to the remote system. If you use *cu* on the remote system to access a third remote system, send **~~.** to cause the second remote *cu* to exit.
~nostop	turn off the DC3/DC1 input control protocol for the remainder of the session. This is useful in case the remote system is one which does not respond properly to the DC3 and DC1 characters,

The *receive* process normally copies data from the remote system to its standard output. A line from the remote that begins with ~> initiates an output diversion to a file. The complete sequence is:

~>[>]: *file*
zero or more lines to be written to *file*
~>

Data from the remote is diverted (or appended, if >> is used) to file. The trailing ~> terminates the diversion.

The use of **~%put** requires *stty*(1) and *cat*(1) on the remote side. It also requires that the current erase and kill characters on the remote system be identical to the current ones on the local system. Backslashes are inserted at appropriate places.

The use of **~%take** requires the existence of *echo*(1) and *cat*(1) on the remote system. Also, **stty tabs** mode should be set on the remote system if tabs are to be copied without expansion.

FILES

/usr/lib/uucp/L-devices
/usr/spool/uucp/LCK..(tty-device)
/dev/null

SEE ALSO

cat(1), echo(1), stty(1), uucp(1C), tty(4).

DIAGNOSTICS

Exit code is zero for normal exit, non-zero (various values) otherwise.

BUGS

There is an artificial slowing of transmission by *cu* during the **~%put** operation so that loss of data is unlikely.

NAME

cut – cut out selected fields of each line of a file

SYNOPSIS

cut –**c**list [file1 file2 ...]
cut –**f**list [–**d** char] [–**s**] [file1 file2 ...]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Use *cut* to cut out columns from a table or fields from each line of a file; in data base parlance, it implements the projection of a relation. The fields as specified by *list* can be fixed length, i.e., 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). *Cut* can be used as a filter; if no files are given, the standard input is used.

The meanings of the options are:

list A comma-separated list of integer field numbers (in increasing order), with optional – to indicate ranges as in the **–o** option of *nroff/troff* for page ranges; e.g., **1,4,7**; **1–3,8**; –**5,10** (short for **1–5,10**); or **3–** (short for third through last field).

–**c***list* The *list* following –**c** (no space) specifies character positions (e.g., –**c1–72** would pass the first 72 characters of each line).

–**f***list* The *list* following –**f** is a list of fields assumed to be separated in the file by a delimiter character (see –**d**); e.g. , –**f1,7** copies the first and seventh field only. Lines with no field delimiters will be passed through intact (useful for table subheadings), unless –**s** is specified.

–**d***char* The character following –**d** is the field delimiter (–**f** option only). Default is *tab*. Space or other characters with special meaning to the shell must be quoted.

–**s** Suppresses lines with no delimiter characters in case of –**f** option. Unless specified, lines with no delimiters will be passed through untouched.

Either the –**c** or –**f** option must be specified.

Hints

Use *grep*(1) to make horizontal "cuts" (by context) through a file, or *paste*(1) to put files together column-wise (i.e., horizontally). To reorder columns in a table, use *cut* and *paste*.

EXAMPLES

cut –d: –f1,5 /etc/passwd	mapping of user ID to names
name = `who am i | cut –f1 –d" "`	to set **name** to current login name.

SEE ALSO

grep(1), paste(1).

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.

NAME

cxref – generate C program cross-reference

SYNOPSIS

cxref [options] files

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

Remarks: *cxref* is implemented on the Series 500 only.

DESCRIPTION

Cxref analyzes a collection of C files and attempts to build a cross-reference table. *Cxref* utilizes a special version of *cpp* to include **#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.

In addition to the **–D**, **–I** and **–U** options (which are identical to their interpretation by *cc*(1)), the following *options* are interpreted by *cxref*:

–c Print a combined cross-reference of all input files.

–w*num* Width option, which formats output no wider than *num* (decimal) columns. This option defaults to 80 if *num* is not specified or is less than 51.

–o file Direct output to the named *file*.

–s Operate silently; do not print input file names.

–t Format listing for 80-column width.

FILES

/usr/lib/xcpp special version of C-preprocessor.

/usr/lib/xpass special version of C-compiler front-end.

SEE ALSO

cc(1).

DIAGNOSTICS

Error messages are unusually cryptic, but usually mean that you cannot compile these files.

BUGS

Cxref considers a formal argument in a *#define* macro definition to be a declaration of that symbol. For example, a program that *#include*s **ctype.h** will contain many declarations of the variable **c**.

Since *cxref* operates by running special versions of the C-preprocessor and the C-compiler front-end, if a file cannot be compiled, it usually cannot be processed by *cxref*.

NAME

date – print and set the date

SYNOPSIS

date [mmddhhmm[yy]] [+format]

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

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 system); the second *mm* is the minute number; *yy* is the last 2 digits of the year number and is optional. For example:

```
date 10080045
```

sets the date to Oct 8, 12:45 AM. The current year is the default if no year is mentioned. The system operates in GMT. *Date* takes care of the conversion to and from local standard and daylight time.

Attempting to set the date backwards generates a warning, and requires an extra confirmation from the (super-)user.

If the argument begins with +, the output of *date* is under the control of the user. The format for the output is similar to that of the first argument to *printf*(3S). All output fields are of fixed size (zero padded if necessary). Each field descriptor is preceded by **%** and will be replaced in the output by its corresponding value. A single **%** is encoded by **%%**. All other characters are copied to the output without change. The string is always terminated with a new-line character.

Date writes an accounting record on the file /usr/adm/wtmp.

Field Descriptors:

n	insert a new-line character
t	insert a tab character
m	month of year – 01 to 12
d	day of month – 01 to 31
y	last 2 digits of year – 00 to 99
D	date as mm/dd/yy
H	hour – 00 to 23
M	minute – 00 to 59
S	second – 00 to 59
T	time as HH:MM:SS
j	Julian date – 001 to 366
w	day of week – Sunday = 0
a	abbreviated weekday – Sun to Sat
h	abbreviated month – Jan to Dec
r	time in AM/PM notation

HARDWARE DEPENDENCIES

Series 500:

The file /dev/kmem is not used.

Do not change the date and/or time in the BASIC language system if your machine also runs HP-

UX. The two operating systems' date and time are incompatible.

EXAMPLE

```
date '+DATE: %m/%d/%y%nTIME: %H:%M:%S'
```

would generate as output:

```
DATE: 08/01/76
TIME: 14:45:05
```

FILES

/dev/kmem
/usr/adm/wtmp

SEE ALSO

ctime(3C).

DIAGNOSTICS

No permission if you aren't the super-user and you try to change the date;

bad conversion if the date set is syntactically incorrect;

bad format character if the field descriptor is not recognizable.

NAME

dc – desk calculator

SYNOPSIS

dc [file]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Dc is an arbitrary precision arithmetic package. Ordinarily it operates on decimal integers, but you may specify an input base, output base, and a number of fractional digits to be maintained. (See *bc*(1), a preprocessor for *dc* that provides infix notation and a C-like syntax that implements functions. *Bc* also provides reasonable control structures for programs.) 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. An end of file on standard input or the q command stop dc. 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 or A–F. 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 and a warning generated.

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. Strings may be nested by using nested pairs of brackets.

<x >x =x — The top two elements of the stack are popped and compared. Register *x* is evaluated if they obey the stated relation.

v — replaces the top element on the stack by its square root. Any existing fractional part of the argument is taken into account, but otherwise the scale factor is ignored.

! — interprets the rest of the line as an HP-UX system command, unless the next character is <, >, or =, in which case the appropriate relational operator is used.

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.
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.
K	pushes the scale factor on the top of the stack.
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.
; :	are used by .I bc for array operations.

The input base may be any number, but only the digits 0–9 and A–F are available for input, thus limiting the usefulness of bases outside the range 1–16. All 16 possible digits may be used in any base; they always take their conventional values.

The output base may be any number. Bases in the range of 2–16 generate the "usual" results, with letters A–F representing the values from 10 through 15. Base 1 generates a string of 1's whose length is the value of the number. Base 0 generates a similiar string consisting of d's. Other bases have each "digit" represented as a (multi-digit) decimal number giving the ordinal of that digit. Each "digit" is signed for negative bases. Given the definition of output base, the command Op will always yield "10", regardless of the base; 01-p yields useful information about the output bases.

SEE ALSO

bc(1).

DIAGNOSTICS

x is unimplemented
: where *x* is an octal number.

stack empty
: when there are not enough elements on the stack to do what was asked.

Out of space
: when the free list is exhausted (too many digits).

Out of headers
: when there are too many numbers being kept around.

Out of pushdown
: when there are too many items on the stack.

Nesting Depth
: when there are too many levels of nested execution.

NAME

dd – convert, reblock, translate, and copy a (tape) file

SYNOPSIS

dd [option = value] ...

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Dd copies the specified input file to the specified output with possible conversions. The standard input and output are used by default. The input and output block size may be specified to take advantage of raw physical I/O.

option	*values*
if = *file*	input file name; standard input is default.
of = *file*	output file name; standard output is default.
ibs = *n*	input block size. *n* bytes (default 512).
obs = *n*	output block size (default 512).
bs = *n*	set both input and output block size, superseding *ibs* and *obs*; also, if no conversion is specified, it is particularly efficient since no in-core copy need be done.
cbs = *n*	conversion buffer size.
skip = *n*	skip *n* input records before starting copy.
seek = *n*	seek *n* records from beginning of output file before copying.
count = *n*	copy only *n* input records.
conv = **ascii**	convert EBCDIC to ASCII.
ebcdic	convert ASCII to EBCDIC.
ibm	slightly different map of ASCII to EBCDIC.
lcase	map alphabetics to lower case.
ucase	map alphabetics to upper case.
swab	swap every pair of bytes.
noerror	do not stop processing on an error.
sync	pad every input record to *ibs*.
...,...	several comma-separated conversions.

Where sizes are specified, a number of bytes is expected. A number may end with **k**, **b**, or **w** to specify multiplication by 1024, 512, or 2 respectively; a pair of numbers may be separated by **x** to indicate a product.

Cbs is used only if *ascii* or *ebcdic* conversion is specified. In the former case *cbs* characters are placed into the conversion buffer, converted to ASCII, and trailing blanks are trimmed and a new-line is added before sending the line to the output. In the latter case ASCII characters are read into the conversion buffer, converted to EBCDIC, and blanks added to make up an output record of size *cbs*.

After completion, *dd* reports the number of whole and partial input and output blocks.

EXAMPLE

This command will read an EBCDIC tape blocked ten 80-byte EBCDIC card images per record into the ASCII file **x**:

```
dd if=/dev/rmt0 of=x ibs=800 cbs=80 conv=ascii,lcase
```

Note the use of raw magtape. *Dd* is especially suited to I/O on the raw physical devices because it allows reading and writing in arbitrary record sizes.

SEE ALSO

cp(1), tr(1).

DIAGNOSTICS

f+p records in(out) numbers of full and partial records read (written)

WARNING

You may experience trouble writing directly to or reading directly from a cartridge tape. For best results, use *tcio*(1) as an input or output filter. For example, use

... | dd... | tcio –ovVS 256 /dev/rct

for output to a cartridge tape, and

tcio –ivS 256 /dev/rct | dd... | ...

for input from a cartridge tape.

BUGS

The ASCII/EBCDIC conversion tables are taken from the 256 character standard in the CACM Nov, 1968. The *ibm* conversion, while less widely accepted as a standard, corresponds better to certain IBM print train conventions. There is no universal solution.

New-lines are inserted only on conversion to ASCII; padding is done only on conversion to EBCDIC. These should be separate options.

NAME

delta – make a delta (change) to an SCCS file

SYNOPSIS

delta [–**r**SID] [–**s**] [–**n**] [–**g**list] [–**m**[mrlist]] [–**y**[comment]] [–**p**] files

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Delta is used to permanently introduce into the named SCCS file changes that were made to the file retrieved by *get*(1) (called the *g-file*, or generated file).

Delta makes a delta to each named SCCS file. If a directory is named, *delta* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read (see *WARNINGS*); each line of the standard input is taken to be the name of an SCCS file to be processed.

Delta may issue prompts on the standard output depending upon certain keyletters specified and flags (see *admin*(1)) that may be present in the SCCS file (see –**m** and –**y** keyletters below).

Keyletter arguments apply independently to each named file.

–**r***SID* Uniquely identifies which delta is to be made to the SCCS file. The use of this keyletter is necessary only if two or more outstanding *get*s for editing (**get –e**) on the same SCCS file were done by the same person (login name). The SID value specified with the –**r** keyletter can be either the SID specified on the *get* command line or the SID to be made as reported by the *get* command (see *get*(1)). A diagnostic results if the specified SID is ambiguous, or, if necessary and omitted on the command line.

–**s** Suppresses the issue, on the standard output, of the created delta's SID, as well as the number of lines inserted, deleted and unchanged in the SCCS file.

–**n** Specifies retention of the edited *g-file* (normally removed at completion of delta processing).

–**g***list* Specifies a *list* (see *get*(1) for the definition of *list*) of deltas which are to be *ignored* when the file is accessed at the change level (SID) created by this delta.

–**m**[*mrlist*] If the SCCS file has the **v** flag set (see *admin*(1)) then a Modification Request (**MR**) number *must* be supplied as the reason for creating the new delta.

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

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

Note that if the **v** flag has a value (see *admin*(1)), it is taken to be the name of a program (or shell procedure) which will validate the correctness of the **MR** numbers. If a non-zero exit status is returned from **MR** number validation program, *delta* terminates (it is assumed that the **MR** numbers were not all valid).

–**y**[*comment*] Arbitrary text used to describe the reason for making the delta. A null string is considered a valid *comment*.

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

–p Causes *delta* to print (on the standard output) the SCCS file differences before and after the delta is applied in a *diff*(1) format.

FILES

All files of the form *?*-file are explained in the *Source Code Control System User's Guide*. The naming convention for these files is also described there. All files below except the g-file are created in the same directory as the s-file. The g-file is created in the user's working directory.

g-file	Existed before the execution of *delta*; removed after completion of *delta* (unless **–n** was specified).
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), get(1), help(1), prs(1), sccsfile(5).
SCCS User's Guide in *HP-UX Concepts and Tutorials*.

DIAGNOSTICS

Use *help*(1) for explanations.

WARNINGS

Lines beginning with an **SOH** ASCII character (octal 001) cannot be placed in the SCCS file unless the **SOH** is escaped. This character has special meaning to SCCS (see *sccsfile*(5)) and will cause an error.

A *get* of many SCCS files, followed by a *delta* of those files, should be avoided when the *get* generates a large amount of data. Instead, multiple *get*/*delta* sequences should be used.

If the standard input (–) is specified on the *delta* command line, the **–m** (if necessary) and **–y** keyletters *must* also be present. Omission of these keyletters causes an error to occur.

NAME

deroff – remove nroff/troff, tbl, and eqn constructs

SYNOPSIS

deroff [–**w**] [–**m**x] [files]

HP-UX COMPATIBILITY

Level: Text processing - HP-UX/EXTENDED

Origin: System III

DESCRIPTION

Deroff reads each of the *files* in sequence and removes all *troff*(1) requests, macro calls, backslash constructs, *eqn*(1) constructs (between **.EQ** and **.EN** lines, and between delimiters), and *tbl*(1) descriptions, and writes the remainder of the file on the standard output. *Deroff* follows chains of included files (**.so** and **.nx** *troff* commands); if a file has already been included, a **.so** naming that file is ignored and a **.nx** naming that file terminates execution. If no input file is given, *deroff* reads the standard input.

The –**m** option may be followed by an **m**, **s**, or **l**. The resulting –**mm** or –**ms** option causes the **mm** or **ms** macros to be interpreted so that only running text is output (i.e., no text from macro lines.) The –**ml** option forces the –**mm** option and also causes deletion of lists associated with the **mm** macros.

If the –**w** option is given, the output is a word list, one "word" per line, with all other characters deleted. Otherwise, the output follows the original, with the deletions mentioned above. In text, a "word" is any string that *contains* at least two letters and is composed of letters, digits, ampersands (**&**), and apostrophes ('); in a macro call, however, a "word" is a string that *begins* with at least two letters and contains a total of at least three letters. Delimiters are any characters other than letters, digits, apostrophes, and ampersands. Trailing apostrophes and ampersands are removed from "words".

SEE ALSO

eqn(1), tbl(1), troff(1).

BUGS

Deroff is not a complete *troff* interpreter, so it can be confused by subtle constructs. Most such errors result in too much rather than too little output.

The –**ml** option does not handle nested lists correctly.

NAME

df – report number of free disk blocks

SYNOPSIS

df [**–t**] [**–f**] [file-systems]

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

Df prints out the number of free blocks and free i-nodes available for on-line file systems by examining the counts kept in the super-blocks; *file-systems* may be specified either by device name or by mounted directory name. If the *file-systems* argument is unspecified, the free space on all of the mounted file systems is printed.

The **–t** flag causes the total allocated block figures to be reported as well.

If the **–f** flag is given, only an actual count of the blocks in the free list is made (free i-nodes are not reported). With this option, *df* will report on raw devices.

HARDWARE DEPENDENCIES

Series 500:

Df cannot report on unmounted raw devices.

FILES

/dev/HP79*
/dev/HP91*
/dev/HP98*
/dev/HP829*1
/etc/mnttab

SEE ALSO

du(1), fs(5), mnttab(5), fsck(8).

NAME

diff, diffh – differential file comparator

SYNOPSIS

diff [–**efbh**] file1 file2

/**usr**/**lib**/**diffh** file1 file2

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Diff tells what lines must be changed in two files to bring them into agreement. If *file1* (*file2*) is –, the standard input is used. If *file1* (*file2*) is a directory, then a file in that directory with the name *file2* (*file1*) is used. The normal output contains lines of these forms:

n1 **a** *n3*,*n4*
n1,*n2* **d** *n3*
n1,*n2* **c** *n3*,*n4*

These lines resemble *ed* commands to convert *file1* into *file2*. The numbers after the letters pertain to *file2*. In fact, by exchanging **a** for **d** and reading backward one may ascertain equally how to convert *file2* into *file1*. As in *ed*, identical pairs where *n1* = *n2* or *n3* = *n4* are abbreviated as a single number.

Following each of these lines come all the lines that are affected in the first file flagged by <, then all the lines that are affected in the second file flagged by >.

The –**b** option causes trailing blanks (spaces and tabs) to be ignored and other strings of blanks to compare equal.

The –**e** option produces a script of *a*, *c* and *d* commands for the editor *ed*, which will recreate *file2* from *file1*. The –**f** option produces a similar script, not useful with *ed*, in the opposite order. In connection with –**e**, the following shell program may help maintain multiple versions of a file. Only an ancestral file ($1) and a chain of version-to-version *ed* scripts ($2,$3,...) made by *diff* need be on hand. A "latest version" appears on the standard output.

```
(shift; cat $*; echo '1,$p') ed – $1
```

Except in rare circumstances, *diff* finds a smallest sufficient set of file differences.

Option –**h** does a fast, half-hearted job. It works only when changed stretches of text are short and well-separated, but does work on files of unlimited length. Options –**e** and –**f** are unavailable with –**h**.

Diffh is equivalent to **diff** –**h**. It must be invoked as shown above in the synopsis, unless the **PATH** variable in your environment includes the directory /**usr**/**lib**.

FILES

/tmp/d?????
/usr/lib/diffh for –**h**

SEE ALSO

cmp(1), comm(1), diff3(1), diffmk(1), dircmp(1), ed(1), sccsdiff(1), sdiff(1).

DIAGNOSTICS

Exit status is 0 for no differences, 1 for some differences, 2 for trouble.

BUGS

Editing scripts produced under the –**e** or –**f** option are naive about creating lines consisting of a single period (.).

The specified files must contain ASCII text in the same format as that produced by the HP-UX editors (*ed*, *ex*, *vi*). That is, all lines (including the last) must end with a new-line. *Diff* will not recognize a final line if the terminating new-line is not there.

NAME

diffmk – mark differences between files

SYNOPSIS

diffmk name1 name2 name3

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Diffmk compares two versions of a file and creates a third file that includes "change mark" commands for *nroff*. *Name1* and *name2* are the old and new versions of the file. *Diffmk* 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 *.

If anyone is so inclined, *diffmk* can be used to produce listings of C (or other) programs with changes marked. A typical command line for such use is:

```
diffmk old.c new.c tmp; nroff macs tmp | pr
```

where the file **macs** contains:

```
.pl 1
.ll 77
.nf
.eo
.nc
```

The **.ll** request might specify a different line length, depending on the nature of the program being printed. The **.eo** and **.nc** requests are probably needed only for C programs.

If the characters | and * are inappropriate, a copy of *diffmk* can be edited to change them (*diffmk* is a shell procedure).

SEE ALSO

diff(1), nroff(1).

BUGS

Aesthetic considerations may dictate manual adjustment of some output. File differences involving only formatting requests may produce undesirable output, i.e., replacing **.sp** by **.sp 2** will produce a "change mark" on the preceding or following line of output.

Although unlikely, certain combinations of formatting requests may cause change marks to either disappear or to mark too much. Manual intervention may be required as the subtleties of all the various formatting macro packages and preprocessors is beyond the scope of *diffmk*.

The input to *tbl*(1) cannot tolerate .mc commads. Any .mc that would appear inside a .TS range will be silently deleted. The script can be changed if this action is inappropriate or *diffmk* can be run on the output from *tbl*(1).

diffmk uses *diff* and thus has whatever limitations on file size and performance that *diff* may impose. In particular the performance is non-linear with the size of the file, and very large files (well over 1000 lines) may take extremely long to process. Breaking the file into smaller pieces may be advisable.

diffmk also uses *ed*(1), and if the file is too large for *ed*, *ed* error messages may be imbedded in the file. Again, breaking the file into smaller pieces may be advisable.

NAME

dircmp – directory difference comparison

SYNOPSIS

dircmp [**–d**] [**–s**] dir1 dir2

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Dircmp examines *dir1* and *dir2* and generates various tabulated information about the contents of the directories.

First, *dircmp* generates a listing, with an appropriate heading, of all files which are found in *dir1* or *dir2*, but not both.

Then, if one or more files are found which are common to both directories, *dircmp* examines each file to determine its type. If the file is a directory or a special file, *dircmp* states this fact and moves on to the next file. If the file is an ordinary file, *dircmp* compares them (using *cmp*(1)), and states whether or not the files are the same.

The options are as follows:

–d run *diff*(1) on ordinary files which occur in both directories but which have different contents, and generate a print-out, with an appropriate heading, which displays the result. Without this option, *dircmp* simply states that the two files are different, and proceeds.

–s suppress all output *except* that which describes the actual differences between *dir1* and *dir2* (i.e. *dircmp* prints only a list of files unique to each directory, and a list of files common to both directories but having different contents).

FILES

/usr/tmp/dc*[a–g] temporary files

SEE ALSO

cmp(1), diff(1).

DIAGNOSTICS

Dircmp generates appropriate error messages about incorrect syntax. Also, a message is generated if *dir1* or *dir2* is not a valid directory name.

NAME

du – summarize disk usage

SYNOPSIS

du [–**ars**] [names]

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

Du gives the number of blocks contained in all files and (recursively) directories within each directory and file specified by the *names* argument. The block count includes the indirect blocks of the file. If *names* is missing, **.** is used.

The optional argument –**s** causes only the grand total (for each of the specified *names*) to be given. The optional argument –**a** causes an entry to be generated for each file. Absence of either causes an entry to be generated for each directory only.

Du is normally silent about directories that cannot be read, files that cannot be opened, etc. The –**r** option will cause *du* to generate messages in such instances.

A file with two or more links is only counted once.

BUGS

If the –**a** option is not used, non-directories given as arguments are not listed.
If there are too many distinct linked files, *du* will count the excess files more than once.
Files with holes in them will get an incorrect block count.
If multiple links are involved, *du* can give different results, depending on the order of *names*.

NAME

echo – echo (print) arguments

SYNOPSIS

echo [arg] ...

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

Echo writes its arguments separated by blanks and terminated by a new-line on the standard output. If *echo*'s arguments are not quoted, or are enclosed in double quotes (" ... "), all metacharacters are expanded according to the shell's interpretation. Thus, *echo* can be used to verify how a certain metacharacter pattern is going to be interpreted by the shell.

Echo also understands C-like escape conventions, which are listed below:

b	backspace
c	print line without new-line
f	form-feed
n	new-line
r	carriage return
t	tab
n	the 8-bit character whose ASCII code is the 1-, 2- or 3-digit octal number ***n***, which must start with a zero.

Note that these escape sequences are first interpreted by the shell before being passed to *echo*. Thus, if the arguments are unquoted, or quoted with double quotes, the backslash must be doubled to prevent premature interpretation by the shell. If the arguments are enclosed in single quotes, then the above escapes may be typed as shown.

To produce a literal backslash on the output, it must be doubled (if unquoted or quoted by double quotes). If quoted with single quotes, a single backslash suffices.

Echo is useful for producing diagnostics in command files and for sending known data into a pipe.

SEE ALSO

sh(1).

NAME

ed – text editor

SYNOPSIS

ed [–] [**–x**] [file]

HP-UX COMPATIBILITY

Level: HP-UX/DEVELOPMENT

Origin: System III

DESCRIPTION

Ed is the standard (line-oriented) text editor. If the *file* argument is given, *ed* simulates an *e* command (see below) on the named file; that is to say, the file is read into *ed*'s buffer so that it can be edited. The optional – suppresses the printing of character counts by *e*, *r*, and *w* commands, of diagnostics from *e* and *q* commands, and of the **!** prompt after a **!***shell command*. If **–x** is present, an *x* command is simulated first to handle an encrypted file. *Ed* operates on a copy of the file it is editing; changes made to the copy have no effect on the file until a *w* (write) command is given. The copy of the text being edited resides in a temporary file called the *buffer*. There is only one buffer.

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 very often be omitted.

In general, only one command may appear on a line. Certain commands allow the input of text. This text is placed in the appropriate place in the buffer. While *ed* is accepting text, it is said to be in *input mode*. In this mode, *no* commands are recognized; all input is merely collected. Input mode is left by typing a period (.) alone at the beginning of a line.

Ed supports a limited form of *regular expression* notation; regular expressions are used in addresses to specify lines and in some commands (e.g., *s*) to specify portions of a line that are to be substituted. A regular expression (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 RE*s match a *single* character:

1.1 An ordinary character (*not* one of those discussed in 1.2 below) is a one-character RE that matches itself.

1.2 A backslash (\) followed by any special character mentioned below is a one-character RE that matches the special character itself. The special characters are:

a. ., *, [, and \ (period, asterisk, left square bracket, and backslash, respectively), which are always special, *except* when they appear within square brackets ([ ]; see 1.4 below).

b. ^ (caret or circumflex), which is special at the *beginning* of an *entire* RE (see 3.1 and 3.2 below), or when it immediately follows the left of a pair of square brackets ([ ]) (see 1.4 below).

c. **$** (currency symbol), which is special at the *end* of an entire RE (see 3.2 below).

d. The character used to bound (i.e., delimit) an entire RE, which is special for that RE (for example, see how slash (/) is used in the *g* command, below.)

1.3 A period (.) is a one-character RE that matches any character except new-line.

1.4 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); e.g., **[ ]a–f]** matches either a right square bracket (]) or one of the letters **a** through **f** inclusive. The four characters listed in 1.2.a above stand for themselves within such a string of characters.

The following rules may be used to construct *RE*s from one-character REs:

2.1 A one-character RE followed by an asterisk (*) is a RE that matches *zero* or more occurrences of the one-character RE. If there is any choice, the longest leftmost string that permits a match is chosen.

2.2 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.

2.3 The concatenation of REs is a RE that matches the concatenation of the strings matched by each component of the RE.

2.4 A RE enclosed between the character sequences \(and \) is a RE that matches whatever the unadorned RE matches.

2.5 The expression *n* matches the same string of characters as was matched by an expression enclosed between \(and \) *earlier* in the same RE. Here *n* is a digit; the sub-expression specified is that beginning with the *n*-th occurrence of \(counting from the left. For example, the expression ^\(.*\)**1*$ matches a line consisting of two repeated appearances of the same string.

Finally, an *entire RE* may be constrained to match only an initial segment or final segment of a line (or both):

3.1 A circumflex (^) at the beginning of an entire RE constrains that RE to match an *initial* segment of a line.

3.2 A currency symbol ($) at the end of an entire RE constrains that RE to match a *final* segment of a line. The construction ^*entire RE*$ constrains the entire RE to match the entire line.

The null RE (e.g., //) is equivalent to the last RE encountered. See 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; e.g, **–5** is understood to mean **.–5**.
9. If an address ends with + or –, then 1 is added to or subtracted from the address, respectively. As a consequence of this rule and of rule 8 immediately above, the address – refers to the line preceding the current line. (To maintain compatibility with earlier versions of the editor, the character ^ in addresses is entirely equivalent to –.) Moreover, trailing + and – characters have a cumulative effect, so —— refers to the current line less 2.
10. For convenience, a comma (**,**) stands for the address pair **1,$**, while a semicolon (**;**) stands for the pair **.,$**.

Commands may require zero, one, or two addresses. Commands that require no addresses regard the presence of an address as an error. Commands that accept one or two addresses assume default addresses when an insufficient number of addresses is given; if more addresses are given than such a command requires, the last one(s) are used.

Typically, addresses are separated from each other by a comma (**,**). They may also be separated by a semicolon (**;**). In the latter case, the current line (**.**) is set to the first address, and only then is the second address calculated. This feature can be used to determine the starting line for forward and backward searches (see rules 5. and 6. above). The second address of any two-address sequence must correspond to a line that follows, in the buffer, the line corresponding to the first address.

In the following list of *ed* commands, the default addresses are shown in parentheses. The parentheses are *not* part of the address; they show that the given addresses are the default.

It is generally illegal for more than one command to appear on a line. However, any command (except *e*, *f*, *r*, or *w*) may be suffixed by **p** or by **l**, in which case the current line is either printed or listed, respectively, as discussed below under the *p* and *l* commands.

(.)a
<text>
.

The *a*ppend 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.

(.)c
<text>
.

The *c*hange 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 *d*elete 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 *e*dit 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* begins with **!**, 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 *E*dit 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 *f*ile-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 *g*lobal 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 *BUGS* and the last paragraph before *FILES* below.

(1,$)G/*RE*/

In the interactive *G*lobal 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 *h*elp command gives a short error message that explains the reason for the most recent **?** diagnostic.

H

The *H*elp 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 *i*nsert 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.

(.,.+1)j

The *j*oin command joins contiguous lines by removing the appropriate new-line characters. If only one address is given, this command does nothing.

(.)k*x*

The mar*k* 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 *l*ist command prints the addressed lines in an unambiguous way: a few non-printing characters (e.g., *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 *m*ove 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 *n*umber 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 *p*rint 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 *q*uit 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 *r*ead 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* begins with **!**, 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.

(.,.)s/*RE***/***replacement***/** or
(.,.)s/*RE***/***replacement***/g**

The *s*ubstitute 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 *u*ndo 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 *w*rite 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* begins with **!**, the rest of the line is taken to be a shell (*sh*(1)) command whose output is written to the specified file, or to the currently-remembered file. Such a shell command is *not* remembered as the current file name.

X

A key string is demanded from the standard input. Subsequent *e*, *r*, and *w* commands will encrypt and decrypt the text with this key by the algorithm of *crypt*(1). An explicitly empty key turns off encryption.

($)=

The 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 HP-UX 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 (e.g., **a.out**) that contain characters not in the ASCII set (bit 8 on) cannot be edited by *ed*.

If the closing delimiter of a RE or of a replacement string (e.g., /) would be the last character before a 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?

FILES

/tmp/e#	temporary; # is the process number.
ed.hup	work is saved here if the terminal is hung up.

SEE ALSO

awk(1), crypt(1), ex(1), grep(1), sed(1), sh(1), vi(1).

The Ed Editor, in *HP-UX Concepts and Tutorials*.

DIAGNOSTICS

? for command errors.

?*file* for an inaccessible file.
(use the ***h***elp and ***H***elp 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.

BUGS

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 (see *sh*(1)).
The sequence **n** in a RE does not match any character.
The *l* command mishandles DEL.
Files encrypted directly with the *crypt*(1) command with the null key cannot be edited.
Because 0 is an illegal address for the *w* command, it is not possible to create an empty file with *ed*.

On systems which do not support *crypt(1)*, the **–x** option and the **X** command are not available.

NAME

enable, disable – enable or disable LP printers

SYNOPSIS

enable printers
disable [**–c**] [**–r**[reason]] printers

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Enable activates the named *printers*, enabling them to print requests taken by *lp*(1). Use *lpstat*(1) to find the status of printers.

Disable deactivates the named *printers*, disabling them from printing requests taken by *lp*(1). By default, any requests that are currently printing on the designated printers will be reprinted in their entirety either on the same printer or on another member of the same class. Use *lpstat*(1) to find the status of printers. Options useful with *disable* are:

–c Cancel any requests that are currently printing on any of the designated printers.

–r[*reason*] Associate a *reason* with the deactivation of the printers. This reason applies to all printers mentioned up to the next **–r** option. If the **–r** option is not present or the **–r** option is given without a reason, then a default reason will be used. *Reason* is reported by *lpstat*(1).

FILES

/usr/spool/lp/*

SEE ALSO

lp(1), lpstat(1).

NAME

env – set environment for command execution

SYNOPSIS

env [–] [name = value] ... [command args]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Env 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), exec(2), profile(5), environ(7).

NAME

err – report error information on last failure

SYNOPSIS

err

HP-UX COMPATIBILITY

Level: HP-UX/NON-STANDARD

Origin: HP

Remarks: *Err* is implemented on the Series 500 only.

DESCRIPTION

Err produces error information on the standard output for the last command which failed. The *errno*, *errinfo*, and octal *trapno* values are listed.

Error information on the last child process which reported a failure is inherited across a *fork* and cleared by *exec*. The error values are also passed back from child to parent to grandparent as long as no errors were detected in the intermediate parent. Intervening commands which are executed successfully have no effect on the saved error information. If a command thinks it successfully completed, and returns an *exit* status of zero, no error information will be returned.

In general, the values reported are for a kernel intrinsic which failed, although values of *errno* or *errinfo* which are set by libraries or commands will also be reported.

SEE ALSO

errno(2), errinfo(2), trapno(2).

WARNING

This command may change in future releases of HP-UX. *Err* is intended for diagnostic purposes only.

BUGS

Information on a real error can be masked by "normal" errors caused by library routines or commands. For example, the library routine *isatty* will generate the error ENOTTY during normal operation.

NAME

ex, edit, e, expreserve, exrecover – text editor commands

SYNOPSIS

ex [–] [**–v**] [**–t**tag] [**–r**] [**–l**] [**–w**n] [**–x**] [**–R**] [+command] name ...
edit [ex options]
e [ex options]
/usr/lib/expreserve
/usr/lib/exrecover

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: UCB

DESCRIPTION

Ex is the root of a family of editors: *edit*, *ex*, *e*, *vi*, and *view*. *Ex* is a superset of *ed*, with the most notable extension being a display editing facility. Display based editing is the focus of *vi*.

If you have not used *ed*, or are a casual user, you will find that the editor *edit* is convenient for you. It avoids some of the complexities of *ex*, which is used mostly by systems programmers and persons very familiar with *ed*.

A display-based editor is often preferred with a CRT terminal. *Vi*(1) is a command that focuses on the display editing portion of *ex*.

The options have the following meanings:

–	suppresses all interactive-user feedback and is useful in processing editor scripts in command files;
–v	equivalent to using *vi* rather than *ex*;
–t *tag*	equivalent to an initial *tag* command, editing the file containing the *tag* and positioning the editor at its definition;
–r	used in recovering after an editor or system crash, retrieving the last saved version of the file *name*, or, if no file is specified, typing a list of saved files;
–l	sets the *showmatch* and *lisp* options;
–w*n*	sets the default window size to *n* lines;
–x	causes *ex* to prompt for a key, which is used to encrypt and decrypt the contents of the file (which should already be encrypted using the same key);
–R	sets the *readonly* option;
+*command*	indicates that the editor should begin by executing the specified *command*. If *command* is omitted, then it defaults to **$**, positioning the editor at the last line of the first file initially.

Expreserve saves the temporary files created during a *vi*, *ex*, *edit*, or *e* editing session that are left after a crash or editor abort. It looks in **/tmp** and moves all such files to **/usr/preserve**. Do not run *expreserve* while an editor is currently being used. *Expreserve* is normally invoked automatically by *rc*(8).

Exrecover is used by any of the above-mentioned editors when invoked with the **–r** option. It is rarely used in a stand-alone fashion.

FILES

/usr/lib/exrecover	recover command
/usr/lib/expreserve	preserve command
/etc/termcap	describes capabilities of terminals

$HOME/.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), environ(5), termcap(5).

Edit: A Tutorial, in *HP-UX Concepts and Tutorials.*
Ex Reference Manual, in *HP-UX Concepts and Tutorials.*
The Vi Editor, in *HP-UX Concepts and Tutorials.*

BUGS

The *undo* command causes all marks to be lost on lines changed and then restored if the marked lines were changed.

Undo never clears 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 do not print a name if the command line "−" option is used.

There is no easy way to do a single scan ignoring case.

The editor does not warn if text is placed in named buffers and not used before exiting the editor.

Null characters are discarded in input files, and cannot appear in resultant files.

On systems which do not support *crypt(1)*, the **−x** option is not available.

NAME

expand, unexpand – expand tabs to spaces, and vice versa

SYNOPSIS

expand [–tabstop] [–tab1,tab2,...,tabn] [file ...]
unexpand [–**a**] [file ...]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: UCB

DESCRIPTION

Expand 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. *Expand* is useful for pre-processing character files (before sorting, looking at specific columns, etc.) 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.

Unexpand puts tabs back into the data from the standard input or the named files and writes the result on the standard output. By default only leading blanks and tabs are reconverted to maximal strings of tabs. If the –**a** option is given, then tabs are inserted whenever they would compress the resultant file by replacing two or more characters.

NAME

expr – evaluate arguments as an expression

SYNOPSIS

expr arguments

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

The arguments are taken as an expression. After evaluation, the result is written on the standard output. Terms of the expression must be separated by blanks. Characters special to the shell must be escaped. Note that **0** is returned to indicate a zero value, rather than the null string. Strings containing blanks or other special characters should be quoted. Integer-valued arguments may be preceded by a unary minus sign. Internally, integers are treated as 32-bit, 2's complement numbers.

The operators and keywords are listed below. Characters that need to be escaped are preceded by \. The list is in order of increasing precedence, with equal precedence operators grouped within { } symbols.

expr \| *expr*
returns the first *expr* if it is neither null nor **0**, otherwise returns the second *expr*.

expr \& *expr*
returns the first *expr* if neither *expr* is null or **0**, otherwise returns **0**.

expr { =, = =, \>, \>=, \<, \<=, != } *expr*
returns the result of an integer comparison if both arguments are integers, otherwise returns the result of a lexical comparison (note that = and = = are identical, in that both test for equality).

expr { +, – } *expr*
addition or subtraction of integer-valued arguments.

expr { *, /, % } *expr*
multiplication, division, or remainder of the integer-valued arguments.

expr **:** *expr*
The matching operator **:** compares the first argument with the second argument which must be a regular expression; regular expression syntax is the same as that of *ed*(1), except that all patterns are "anchored" (i.e., begin with ^) and, therefore, ^ is not a special character, in that context. Normally, the matching operator returns the number of characters matched (**0** on failure). Alternatively, the \(...\) pattern symbols can be used to return a portion of the first argument.

length *expr*
The length of *expr*.

substr *expr expr expr*
Takes the substring of the first *expr*, starting at the character specified by the second *expr* for the length given by the third *expr*.

index *expr expr*
Returns the position in the first *expr* which contains a character found in the second *expr*.

match Match is a prefix operator equivalent to the infix operator **:**.

EXAMPLES

1. a=`expr $a + 1`

   adds 1 to the shell variable **a**.

2. # 'For $a equal to either "/usr/abc/file" or "file"'
   expr $a : '.*/\(.*\)' \| $a

returns the last segment of a path name (i.e., file). Watch out for / alone as an argument: *expr* will take it as the division operator (see BUGS below).

3. # ´A better representation of example 2.´
expr //$a : .*/\(.*\)

The addition of the // characters eliminates any ambiguity about the division operator and simplifies the whole expression.

4. expr $VAR : ´.*´

returns the number of characters in **$VAR**.

RETURN VALUE

As a side effect of expression evaluation, *expr* returns the following exit values:

0	if the expression is neither null nor **0**
1	if the expression *is* null or **0**
2	for invalid expressions.

SEE ALSO

ed(1), sh(1), test(1).

DIAGNOSTICS

syntax error	for operator/operand errors
non-numeric argument	if arithmetic is attempted on such a string

BUGS

After argument processing by the shell, *expr* cannot tell the difference between an operator and an operand except by the value. If **$a** is an =, the command:

expr $a = ´=´

looks lil:e:

expr = = =

as the arguments are passed to *expr* (and they will all be taken as the = operator). The following works:

expr X$a = X=

NAME

f77 – FORTRAN 77 compiler

SYNOPSIS

f77 [options] filelist

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: HP

Remarks: This manual page describes the FORTRAN 77 compiler as implemented on the Series 200 computers. Refer to *fc*(1) for a description of the FORTRAN 77 compiler as implemented on the Series 500 computers.

DESCRIPTION

F77 is the HP-UX FORTRAN 77 compiler. It accepts several types of file arguments:

(1) Arguments whose names end with **.f** are taken to be Fortran 77 source programs. They are compiled, and each object program is left in the current directory in a file whose name is that of the source, with **.o** substituted for **.f**. (The **.o** file will not be created for any source which fails to compile correctly.) In the same way, arguments whose names end with **.c** or **.s** are taken to be **C** or **assembly source** programs and are compiled or assembled, producing **.o** files.

(2) Arguments whose names end with **.r** are taken to be *ratfor*(1) source programs. These are first transformed by the *ratfor* preprocessor, and then compiled by *f77* producing **.o** files.

(3) Arguments whose names end with **.o** are passed on to the linker (*ld*(1)) to be linked into the final program.

The following options are recognized:

–b causes the compiler to generate code for floating point operations that will use floating point hardware if it is installed in the computer at run-time.

–c Compile, but do not link. The compiler produces a relocatable file (**.o**) for each file in *filelist*. This includes **.c**, **.f**, **.r**, and **.s** files.

–C enable range checking (same as **$OPTION RANGE ON**).

–f causes the compiler to generate code for floating point operations that will use floating point hardware. This code does not run unless floating point hardware is installed.

–I2 makes the default size of integers and logicals equal to one-half wordsize (same as **$OPTION SHORT**).

–I4 This is the default. This option causes the compiler to generate code to store integers and logicals in four byte quantities unless specifically declared otherwise.

–k This option forces dynamic storage for local arrays. If specified, arrays are subject to the 32K byte limitation for local data space.

–K This option forces static storage for all local variables. This is to provide a convenient path for importing FORTRAN 66 and FORTRAN 77 programs which were written to depend on static allocation of memory (i.e. variables retaining their values between invocations of the respective program units.)

–N<*secondary*><*n*>

This option adjusts the size of internal compiler tables. The compiler uses fixed size arrays for certain internal tables. *Secondary* is one of the letters from the set {**qsxcndpw**}, and *n* is an integer value. *Secondary* and *n* are **not** optional. The table sizes can be re-specified using one of the secondary letters and the number *n* as follows:

q maximum size of equivalence table (default = 150 table entries).

s maximum size of statement label table (default = 201 table entries).

x maximum size of external symbol table (default = 200 table entries).

c maximum size of control statements table (default = 200 table entries).

n maximum size of the hash table of symbols (default = 401 table entries).

d maximum size of the dimtab table. This table maintains information about the definitions of all arrays in pass 2 (default = 1000 table entries).

p maximum size of the parameter stack in pass 2 (default = 150 table entries).

w maximum size of the switch table stack in pass 2 (default = 250 table entries).

–o *outfile* name the output file from the linker *outfile* instead of *a.out*.

–onetrip causes the compiler to generate code that executes any DO loop at least once.

–p prepares object files for profiling (see *prof*(1)).

–s issue warnings for non-ANSI features (same as **$OPTION ANSI ON**).

–S compiles the named programs and leaves the assembler language output in corresponding files whose names are suffixed with **.s**. (no **.o** files are created).

–u force types of identifiers to be implicitly undeclared (same as specifying **IMPLICIT NONE**; no other **IMPLICIT** statements are permitted).

–U do not convert upper-case letters to lower-case (default is to convert to lower-case).

–v write expanded compiler and linker runstrings to *stderr*.

–w suppress warning messages (same as **$OPTION WARNINGS OFF**).

–w66 supresses warnings about FORTRAN 66 features used.

F77 also recognizes the following options, but they have no effect if they are encountered. This is for compatibility with other FORTRAN systems.

–m, –f, –O, –E, –R, –F

A warning is written to *stderr* if any of these options is used.

Any other options are taken to be arguments to *ld*, and are passed along to the linker.

FILES

file.r	input file (*ratfor* source file)
file.f	input file (FORTRAN source file)
file.s	input file (assembly source file)
file.c	input file (C source file)
file.o	object file
a.out	linked executable output file
/usr/bin/f77	mother program (linked to /usr/bin/fc)
/usr/lib/f77pass1	compiler pass 1
/lib/f1	compiler pass 2
/usr/lib/libF77.a	intrinsic function library
/urs/lib/libI77.a	FORTRAN I/O library
/usr/libc.a	C library; See Section 3 of this manual
/lib/libm.a	math library
/lib/frt0.o	run-time startoff routine
/lib/mfrt0.o	startoff with profiling

SEE ALSO

as(1), asa(1), cc(1), ld(1).

Fortran/9000 Language Reference Manual.
Fortran/9000 Language Reference Supplement.

DIAGNOSTICS

The diagnostics produced by *f77* are intended to be self-explanatory. Errors are written to *stderr*.

NAME

factor, primes – factor a number, generate large primes

SYNOPSIS

factor [number]

primes [start [stop]]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

When *factor* is invoked without an argument, it waits for a number to be typed in. If you type in a positive number, it factors the number and print its prime factors; each one is printed the proper number of times. Then it waits for another number. It exits if it encounters a zero or any non-numeric character.

If *factor* is invoked with an argument, it factors the number as above and then exits.

Maximum time to factor is proportional to sqrt(*n*) and occurs when *n* is prime or the square of a prime.

The largest number that can be dealt with by *factor* is 1.0e14.

Primes prints prime numbers between a lower and upper bound. If *primes* is invoked without any arguments, it waits for two numbers to be typed in. The first number is interpreted as the lower bound, and the second as the upper bound. All prime numbers in the resulting inclusive range are printed.

If *start* is specified, all primes greater than or equal to *start* are printed. If both *start* and *stop* are given, then all primes occurring in the inclusive range "*start* – *stop*" are printed.

Start and *stop* values must be integers represented as long integers.

If the stop value is omitted in either case, *primes* runs until either overflow occurs or it is stopped by typing *interrupt*.

The largest number that can be dealt with by *primes* is 2,147,483,647.

DIAGNOSTICS

"Ouch" when the input is out of range, for garbage input, or when *start* is greater than *stop*.

NAME

fc – FORTRAN 77 compiler

SYNOPSIS

fc [options] files

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: HP

Remarks: This manual page describes the FORTRAN 77 compiler as implemented on the Series 500 computers. Refer to *f77*(1) for a description of FORTRAN 77 as implemented on the Series 200 computers.

DESCRIPTION

Fc is the HP-UX FORTRAN 77 compiler. It accepts two types of file arguments:

(1) Arguments whose names end with **.f** are taken to be FORTRAN 77 source programs. They are compiled, and each object program is left in the current directory in a file whose name is that of the source, with **.o** substituted for **.f**. (The **.o** file will not be created for a single source which is compiled and loaded, nor for any source which fails to compile correctly.)

(2) Arguments whose names end with **.o** are passed on to the linker (*ld*(1)) to be linked into the final program.

The following options are recognized:

–c suppress linking and produce object (**.o**) files from source files;

–C enable range checking (same as **$OPTION RANGE ON**);

–D compile debug lines (source lines with a "D" or "d" in column 1 are treated as comments by default);

–e write errors to *stderr*;

–F causes the compiler to generate information used by various program analysis programs.

–g causes the compiler to generate additional information needed for the use of a symbolic debugger.

–I2 make default size of integers and logicals half a word (same as **$OPTION SHORT**);

–K automatically SAVE all local variables in all subprograms;

–L write a program listing to *stdout* during compilation;

–o outfile name the output file from the linker *outfile* instead of *a.out*;

–onetrip execute any DO loop at least once;

–Q dfile specify *dfile* as the option file;

–s issue warnings for non-ANSI features (same as **$OPTION ANSI ON**).

–u force types of identifiers to be implicitly undeclared (same as specifying **IMPLICIT NONE**; no other **IMPLICIT** statements are permitted);

–U use upper case for external names (default is lower case);

–v write expanded compiler and linker runstrings to *stderr*;

–Vc put all COMMONs in the virtual data area;

–Vd put all SAVE'd and initialized (DATA statement) variables in the virtual data area;

–Vf put all FORMAT strings in the virtual data area;

–w suppress warning messages (same as **$OPTION WARNINGS OFF**).

Any other options are taken to be arguments to *ld*, and are passed along to the linker. Note, however, that, because the **–s** option is recognized by the compiler, it is not possible to pass the strip option to the linker. Programs which are to be stripped may be compiled with the **–c** option, then linked separately.

FILES

file.f	input file (FORTRAN source file)
file.o	object file
a.out	linked executable output file
/bin/fc	mother program
/usr/lib/f77comp	compiler
/lib/frt0.o	runtime startup
/lib/libI77.a	FORTRAN I/O library
/lib/libF77.a	FORTRAN math library
/usr/tmp/*	temporary files used by the compiler; names are created by *tmpnam*(3S).

SEE ALSO

asa(1), ld(1).

FORTRAN/9000 Language Reference Manual.
Structured FORTRAN 77, by Seymour Pollack.

DIAGNOSTICS

The diagnostics produced by *fc* are intended to be self-explanatory. If a listing is requested (**–L** option), errors are written to the listing file. If no listing is being generated, errors are written to *stderr*. Errors will be written to both the listing file and *stderr* if the **–L** and **–e** options are both specified. Occasional messages may be produced by the linker.

NAME

file – determine file type

SYNOPSIS

file [**–c**] [**–f** ffile] [**–m** mfile] arg ...

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

File performs a series of tests on each argument in an attempt to classify it. If an argument appears to be ASCII, *file* examines the first 512 bytes and tries to guess its language. If an argument is an executable *a.out* file, *file* will print the version stamp, provided it is greater than 0 (see *ld*(1)).

File uses the file */etc/magic* to identify files that have some sort of ***magic number***, that is, any file containing a numeric or string constant that indicates its type. Commentary at the beginning of */etc/magic* explains its format.

The options are as follows:

–c causes *file* to check the magic file for format errors. This validation is not normally carried out for reasons of efficiency. No file classification is done under **–c**.

–f *ffile* specifies that *ffile* is a file containing a list of the files which are to be examined. *File* then classifies each file whose name appears in *ffile*.

–m *mfile*
instructs *file* to use *mfile* as the magic file.

SEE ALSO

ld(1).

NAME

find – find files

SYNOPSIS

find path-name-list expression

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Find recursively descends the directory hierarchy for each path name in the *path-name-list* (i.e., one or more path names) 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*.

–**name** *string* True if *string* matches the current file name. Normal shell argument syntax may be used if escaped (watch out for **[**, **?** and *****).

–**perm** *onum* True if the file permission flags exactly match the octal number *onum* (see *chmod*(1)). If *onum* is prefixed by a minus sign, more flag bits (017777, see *stat*(2)) become significant and the flags are compared:

(flags&onum) = = onum

–**type** *c* True if the type of the file is *c*, where *c* is **b**, **c**, **d**, **p**, or **f** for block special file, character special file, directory, fifo (a.k.a named pipe), or plain file.

–**links** *n* True if the file has *n* links.

–**user** *uname* True if the file belongs to the user *uname*. If *uname* is numeric and does not appear as a login name in the /**etc**/**passwd** file, it is taken as a user ID.

–**group** *gname* True if the file belongs to the group *gname*. If *gname* is numeric and does not appear in the /**etc**/**group** file, it is taken as a group ID.

–**size** *n* True if the file is *n* blocks long.

–**atime** *n* True if the file has been accessed in *n* days.

–**mtime** *n* True if the file has been modified in *n* days.

–**ctime** *n* True if the file has been changed in *n* days.

–**exec** *cmd* True if the executed *cmd* returns a zero value as exit status. The end of *cmd* must be punctuated by an escaped semicolon. A command argument {} is replaced by the current path name.

–**ok** *cmd* Like –**exec** except that the generated command line is printed with a question mark first, and is executed only if the user responds by typing **y**.

–**print** Always true; causes the current path name to be printed. This option must be included on the *find* command line anytime you want *find* to print the path names it has found on the standard output. If –**print** is not specified, *find* locates the files, but fails to tell you about them!

When –**print** is specified as the only *expression*, *find* prints the absolute path names of all files it finds, beginning at each directory in the *path-name-list*. If –**print** is included as the last component of an *expression*, *find* prints the absolute path names of only those files which satisfy the other primaries in the *expression*.

–**cpio** *device* Write the current file on *device* in *cpio*(5) format (5120 byte records).

–newer *file* True if the current file has been modified more recently than the argument *file*.

(*expression* **)** True if the parenthesized expression is true (parentheses are special to the shell and must be escaped).

–inum *n* True if the file has inode number *n*.

–ncpio Same as **–cpio** but adds the -c option to **cpio**.

The primaries may be combined using the following operators (in order of decreasing precedence):

1) The negation of a primary (**!** is the unary *not* operator).
2) Concatenation of primaries (the *and* operation is implied by the juxtaposition of two primaries).
3) Alternation of primaries (**–o** is the *or* operator).

EXAMPLE

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 {} \;
```

Note that the spaces delimiting the escaped parentheses are required.

The following command finds all files in */usr/lib* which are directories:

```
find /usr/lib –type d –print
```

Conversely,

```
find /usr/lib ! –type d –print
```

finds all files in */usr/lib* which are *not* directories. Finally,

```
find /users –print
```

prints every file found under */users*.

FILES

/etc/passwd
/etc/group

SEE ALSO

cpio(1), sh(1), test(1), stat(2), cpio(5), fs(5).

NAME

get – get a version of an SCCS file

SYNOPSIS

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

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Get generates an ASCII text file from each named SCCS file according to the specifications given by its keyletter arguments, which begin with –. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, *get* behaves as though each file in the directory 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 keyletter arguments is explained below as though only one SCCS file is to be processed, but the effects of any keyletter argument applies independently to each named file.

–**r***SID* The *S*CCS *ID*entification string (SID) of the version (delta) of an SCCS file to be retrieved. 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** keyletter is also used), as a function of the SID specified.

–**c***cutoff* *Cutoff* date-time, in the form:

YY[MM[DD[HH[MM[SS]]]]]

No changes (deltas) to the SCCS file which were created after the specified *cutoff* date-time are included in the generated ASCII text file. Units omitted from the date-time default to their maximum possible values; that is, –**c7502** is equivalent to –**c750228235959**. Any number of non-numeric characters may separate the various 2 digit pieces of the *cutoff* date-time. This feature allows one to specify a *cutoff* date in the form: "**–c77/2/2 9:22:25**". Note that this implies that one may use the 83/04/30 and 13:03:44 identification keywords (see below) for nested *get*s.

–**e** Indicates that the *get* is for the purpose of editing or making a change (delta) to the SCCS file via a subsequent use of *delta*(1). The –**e** keyletter used in a *get* for a particular version (SID) of the SCCS file prevents further *get*s for editing on the same SID until *delta* is executed or the **j** (joint edit) flag is set in the SCCS file (see *admin*(1)). Concurrent use of **get –e** for different SIDs is always allowed.

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

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

–**b** Used with the –**e** keyletter to indicate that the new delta should have an SID in a new branch as shown in Table 1. This keyletter is ignored if the **b** flag is not present in the file (see *admin*(1)) or if the retrieved *delta* is not a leaf *delta*. (A leaf *delta* is one that has no successors on the SCCS file tree.)

Note: A branch *delta* may always be created from a non-leaf *delta*.

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

<list> :: = <range> | <list> , <range>
<range> :: = SID | SID – SID

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

–x*list* A *list* of deltas to be excluded (forced not to be applied) in the creation of the generated file. See the **–i** keyletter for the *list* format.

–k Suppresses replacement of identification keywords (see below) in the retrieved text by their value. The **–k** keyletter is implied by the **–e** keyletter.

–l[p] Causes a delta summary to be written into an *l-file*. If **–lp** is used then an *l-file* is not created; the delta summary is written on the standard output instead. See *FILES* for the format of the *l-file*. The user must have read permission for the s-file in order to use the **–l** option.

–p Causes the text retrieved from the SCCS file to be written on the standard output. No *g-file* is created. All output which normally goes to the standard output goes to file descriptor 2 (stderr) instead, unless the **–s** keyletter is used, in which case it disappears.

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

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

–n Causes each generated text line to be preceded with the %M% identification keyword value (see below). The format is: %M% value, followed by a horizontal tab, followed by the text line. When both the **–m** and **–n** keyletters are used, the format is: %M% value, followed by a horizontal tab, followed by the **–m** keyletter generated format.

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

–t Used to access the most recently created ("top") delta in a given release (e.g., **–r1**), or release and level (e.g., **–r1.2**).

–a*seq-no*. The delta sequence number of the SCCS file delta (version) to be retrieved (see *sccsfile*(5)). This keyletter is used by the *comb*(1) command; it is not a generally useful keyletter, and users should not use it. If both the **–r** and **–a** keyletters are specified, the **–a** keyletter is used. Care should be taken when using the **–a** keyletter in conjunction with the **–e** keyletter, as the SID of the delta to be created may not be what one expects. The **–r** keyletter can be used with the **–a** and **–e** keyletters to control the naming of the SID of the delta to be created.

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

If the **–e** keyletter is used, the SID of the delta to be made appears after the SID accessed and before the number of lines generated. If there is more than one named file or if a directory or standard input is named, each file name is printed (preceded by a new-line) before it is processed. If the **–i** keyletter is used included deltas are listed following the notation "Included"; if the **–x** keyletter is used, excluded deltas are listed following the notation "Excluded".

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

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

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

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

\# Successor.

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

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

Identification Keywords

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

Keyword	*Value*
%M%	Module name: either the value of the **m** flag in the file (see *admin*(1)), or if absent, the name of the SCCS file with the leading **s.** removed.
%I%	SCCS identification (SID) (%R%.%L%.%B%.%S%) of the retrieved text.
%R%	Release.
%L%	Level.
%B%	Branch.
%S%	Sequence.
%D%	Current date (YY/MM/DD).
%H%	Current date (MM/DD/YY).

%T% Current time (HH:MM:SS).
%E% Date newest applied delta was created (YY/MM/DD).
%G% Date newest applied delta was created (MM/DD/YY).
%U% Time newest applied delta was created (HH:MM:SS).
%Y% Module type: value of the **t** flag in the SCCS file (see *admin*(1)).
%F% SCCS file name.
%P% Fully qualified SCCS file name.
%Q% The value of the **q** flag in the file (see *admin*(1)).
%C% Current line number. This keyword is intended for identifying messages output by the program such as "this 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%

HARDWARE DEPENDENCIES

Series 200/500:

Comb(1) is not currently supported.

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

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

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

a. A blank character if the delta was applied;
   * otherwise.
b. A blank character if the delta was applied or 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** keyletter).
d. Blank.
e. SCCS identification (SID).
f. Tab character.
g. Date and time (in the form YY/MM/DD HH:MM:SS) of creation.
h. Blank.
i. Login name of person who created *delta*.

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

The *p-file* is used to pass information resulting from a *get* with an **–e** keyletter along to *delta*. Its contents are also used to prevent a subsequent execution of *get* with an **–e** keyletter for the same SID until *delta* is executed or the joint edit flag, **j**, (see *admin*(1)) is set in the SCCS file. The *p-file* is created in the directory containing the SCCS file and the effective user must have write permission in that directory. Its mode is 644 and it is owned by the effective user. The format of the *p-file* is: the 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** keyletter argument if it was present, followed by a blank and the **–x** keyletter argument 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 (2 bytes) process ID of the command (i.e., *get*) that created it. The *z-file* is created in the directory containing the SCCS file for the duration of *get*. The same protection restrictions as those for the *p-file* apply for the *z-file*. The *z-file* is created mode 444.

SEE ALSO

admin(1), delta(1), help(1), prs(1), what(1), sccsfile(5).

SCCS User's Guide in *HP-UX Concepts and Tutorials.*

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

If the effective user has write permission (either explicitly or implicitly) in the directory containing the SCCS files, but the real user doesn't, then only one file may be named when the **–e** keyletter is used.

An l-file cannot be generated when **–g** is used. In other words, **–g –l** does not work.

NAME

getopt – parse command options

SYNOPSIS

**`getopt** optstring args`

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

Getopt is used to break up options in command lines for easy parsing by shell procedures, and to check for legal options. *Optstring* is a string of recognized option letters (see getopt(3C)); if a letter is followed by a colon, the option is expected to have an argument which may or may not be separated from it by white space. The special option –– is used to delimit the end of the options. *Getopt* will place –– in the arguments at the end of the options, or recognize it if used explicitly. The shell arguments ($1 $2 . . .) are reset so that each option is preceded by a – and placed in its own shell argument. Each option argument is also placed in its own shell argument.

The most common use of *getopt* is in the shell's **set** command (see the example below). There, *getopt* converts the command line to a more easily parsed form. *Getopt* writes the modified command line to the standard output.

EXAMPLE

The following code fragment shows how one might process the arguments for a command that can take the options **a** and **b**, and the option **o**, which requires an argument.

```
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; shift;;
        --)    shift;   break;;
        esac
done
```

This code will accept 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
```

SEE ALSO

sh(1), getopt(3C).

DIAGNOSTICS

Getopt prints an error message on the standard error when it encounters an option letter not included in *optstring*.

BUGS

The output of *getopt* cannot be more than 256 characters.

NAME

grep, egrep, fgrep – search an ASCII file for a pattern

SYNOPSIS

grep [**–bclnsv**] pattern [file ...]

egrep [**–bclnvef**] pattern [file ...]

fgrep [**–bclnvxef**] strings [file ...]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Commands of the *grep* family search the input *files* (standard input if no *file* is specified) for lines matching a *pattern*. Normally, each line found is copied to the standard output. *Grep patterns* are limited regular expressions in the style of *ed*(1); it uses a compact non-deterministic algorithm. *Egrep patterns* are full regular expressions; it uses a fast deterministic algorithm that sometimes needs exponential space. *Fgrep patterns* are fixed *strings*; it is fast and compact. The following *options* are recognized:

–v All lines but those matching are printed.

–x (Exact) only lines matched in their entirety are printed (*fgrep* only).

–c Only a count of matching lines is printed.

–l Only the names of files with matching lines are listed (once), separated by new-lines.

–n Each line is preceded by its relative line number in the file.

–b Each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.

–s The error messages produced for nonexistent or unreadable files are suppressed (*grep* only).

–e *expression*
Same as a simple *expression* argument, but useful when the *expression* begins with a – (does not work with *grep*).

–f *file* The regular *expression* (*egrep*) or *strings* list (*fgrep*) is taken from the *file*.

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

Fgrep searches for lines that contain one of the *strings*, each of which is separated from the next by a new-line.

Egrep accepts regular expressions as in *ed*(1), except for \(and \), with the addition of:

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

The order of precedence of operators is **[ ]**, then *** ? +**, then concatenation, then | and new-line.

EXAMPLES

The following example searches two files, finding all lines containing occurrences of any of four strings:

```
fgrep 'if
then
else
fi' script1 script2
```

Note that the single quotes are necessary to tell *fgrep* when the strings have ended and the file names have begun.

This example searches for a new-line in a file:

```
grep –v '\.' file1
```

The **–v** option causes *grep* to print those lines that do not match the expression. Since a new-line cannot be matched with dot, only lines containing a new-line are printed.

SEE ALSO

ed(1), sed(1), sh(1).

DIAGNOSTICS

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

BUGS

Lines are limited to 256 characters; longer lines are truncated.
Egrep does not recognize ranges, such as **[a–z]**, in character classes.
Grep finds lines in the input file by searching for a new-line. Thus, if there is no new-line at the end of the file, *grep* will ignore the last line of the file.

NAME

head – give first few lines

SYNOPSIS

head [–count] [file ...]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: UCB

DESCRIPTION

This filter gives the first *count* lines of each of the specified files, or of the standard input. If *count* is omitted it defaults to 10.

SEE ALSO

tail(1).

NAME

help – ask for help

SYNOPSIS

help [args]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Help finds information to explain a message from a command or explain the use of a command. Zero or more arguments may be supplied. If no arguments are given, *help* will prompt for one.

The arguments may be either message numbers (which normally appear in parentheses following messages) or command names, of one of the following types:

type 1 Begins with non-numerics, ends in numerics. The non-numeric prefix is usually an abbreviation for the program or set of routines which produced the message (e.g., **ge6**, for message 6 from the *get* command).

type 2 Does not contain numerics (as a command, such as **get**)

type 3 Is all numeric (e.g., **212**)

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

When all else fails, try "help stuck".

FILES

/usr/lib/help directory containing files of message text.

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

Only SCCS and a very few other commands currently use *help*.

NAME

hostname – set or print name of current host system

SYNOPSIS

hostname [nameofhost]

HP-UX COMPATABILITY

Level: HP-UX/STANDARD

Origin: UCB

DESCRIPTION

The *hostname* command prints the name of the current host, as given in the *uname* system call. The super-user can set the hostname by giving an argument; this is usually done in the startup script */etc/rc*.

SEE ALSO

uname(1), gethostname(2), sethostname(2), uname(2).

NAME

hp – handle special functions of HP 2640 and 2621-series terminals

SYNOPSIS

hp [**–e**] [**–m**]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Hp supports special functions of the Hewlett-Packard 2640 series of terminals, with the primary purpose of producing accurate representations of most *nroff* output. A typical use is:

nroff –h files . . . | hp

Regardless of the hardware options on your terminal, *hp* tries to do sensible things with underlining and reverse line-feeds. If the terminal has the "display enhancements" feature, subscripts and superscripts can be indicated in distinct ways. If it has the "mathematical-symbol" feature, Greek and other special characters can be displayed.

The options are as follows:

–e It is assumed that your terminal has the "display enhancements" feature, and so maximal use is made of the added display modes. Overstruck characters are presented in the Underline mode. Superscripts are shown in Half-bright mode, and subscripts in Half-bright, Underlined mode. If this flag is omitted, *hp* assumes that your terminal lacks the "display enhancements" feature. In this case, all overstruck characters, subscripts, and superscripts are displayed in Inverse Video mode, i.e., dark-on-light, rather than the usual light-on-dark.

–m Requests minimization of output by removal of new-lines. Any contiguous sequence of 3 or more new-lines is converted into a sequence of only 2 new-lines; i.e., any number of successive blank lines produces only a single blank output line. This allows you to retain more actual text on the screen.

With regard to Greek and other special characters, *hp* provides the same set as does *300*(1), except that "not" is approximated by a right arrow, and only the top half of the integral sign is shown. The display is adequate for examining output from *neqn*.

DIAGNOSTICS

"line too long", if the representation of a line exceeds 1,024 characters.

The exit codes are **0** for normal termination, and **2** for all errors.

SEE ALSO

300(1), col(1), eqn(1), greek(1), nroff(1), tbl(1).

BUGS

An "overstriking sequence" is defined as a printing character followed by a backspace followed by another printing character. In such sequences, if either printing character is an underscore, the other printing character is shown underlined or in Inverse Video; otherwise, only the first printing character is shown (again, underlined or in Inverse Video). Nothing special is done if a backspace is adjacent to an ASCII control character. Sequences of control characters (e.g., reverse line-feeds, backspaces) can make text "disappear"; in particular, tables generated by *tbl*(1) that contain vertical lines will often be missing the lines of text that contain the "foot" of a vertical line, unless the input to *hp* is piped through *col*(1).

Although some terminals do provide numerical superscript characters, no attempt is made to display them.

NAME

hyphen – find hyphenated words

SYNOPSIS

hyphen [files]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Hyphen finds all the hyphenated words ending lines in *files* and prints them on the standard output. If no arguments are given, the standard input is used; thus, *hyphen* may be used as a filter.

EXAMPLE

The following will allow the proofreading of *nroff* hyphenation in *textfile*.

```
mm textfile | hyphen
```

SEE ALSO

mm(1), nroff(1).

BUGS

Hyphen cannot cope with hyphenated *italic* (i.e., underlined) words; it will often miss them completely, or mangle them.

Hyphen occasionally gets confused, but with no ill effects other than spurious extra output.

NAME

id – print user, group IDs and names

SYNOPSIS

id

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

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

SEE ALSO

logname(1), getgid(2), getuid(2).

NAME

join – relational database operator

SYNOPSIS

join [options] file1 file2

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

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

File1 and *file2* must be sorted in increasing ASCII collating sequence on the fields on which they are to be joined, normally the first in each line.

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

Fields are normally separated by blank, tab or new-line. In this case, multiple separators count as one, and leading separators are discarded.

These options are recognized:

–a*n* In addition to the normal output, produce a line for each unpairable line in file *n*, where *n* is 1 or 2.

–e *s* Replace empty output fields by string *s*.

–j*n m* Join on the *m*th field of file *n*. If *n* is missing, use the *m*th field in each file.

–o *list* Each output line comprises the fields specifed in *list*, each element of which has the form *n.m*, where *n* is a file number and *m* is a field number.

–t*c* Use character *c* as a separator (tab character). Every appearance of *c* in a line is significant.

SEE ALSO

awk(1), comm(1), sort(1).

BUGS

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

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

NAME

kill – terminate a process

SYNOPSIS

kill [–signo] pid ...

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

Kill sends a signal to the process(es) specified by each *pid* mentioned on the command line. By default, signal 15 (SIGTERM) is sent. This normally kills processes that do not catch or ignore the signal.

If *signo* is given, then that signal is sent instead of SIGTERM (refer to *signal*(2) for a list of the defined signals). In particular, "kill –9 ... " is a sure kill.

The correct *pid* to specify can be found in two ways. The shell reports the process id of each command executed in the background (using **&**). If a pipeline is executed in the background, then only the process id of the last command in the pipeline is reported. Regardless, the reported process id is the *pid* to use. If you forget the reported process id of background tasks, or you need the process id of some other command, use the *ps*(1) command.

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

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

SEE ALSO

ps(1), sh(1), kill(2), signal(2).

BUGS

If a process becomes hung during some operation (such as I/O) so that it is never scheduled, that process will not die until it is allowed to run. Thus, such a process may never go away after the kill. Often, the system must be re-booted in such cases.

NAME

last – indicate last logins of users and teletypes

SYNOPSIS

last, lastb [–N] [name ...] [tty ...]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: UCB

DESCRIPTION

Last 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 may be given fully or abbreviated. For example **last 0** is the same as **last tty0**. If multiple arguments are given, the information which applies to any of the arguments is printed. For example **last root console** lists all of "root's" sessions as well as all sessions on the console terminal.

Last 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 still continuing or was cut short by a re-boot, *last* indicates this.

The pseudo-user **reboot** logs in at re-boots of the system, thus

```
last reboot
```

gives an indication of mean time between re-boots.

Last with no arguments prints a record of all logins and logouts, in reverse order. The **-N** option limits the report to N lines.

If *last* is interrupted, it indicates how far the search has progressed in *wtmp*. If interrupted with a quit signal (generated by a control-\), *last* indicates how far the search has progressed so far, and the search continues.

Lastb will look back in the *btmp* database to display bad login information.

FILES

/usr/adm/wtmp
: login data base

/usr/adm/btmp
: bad login database

SEE ALSO

login(1), wtmp(5).

NAME

ld – link editor

SYNOPSIS

ld [–**sulxrdnm**] [–**o** name] [–**h** name] [–**e** name] [–**R** value] [–**V** num] file ...

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: HP and System III

Remarks: This manual page describes the linker as implemented on the Series 200 computers. Refer to other *ld*(1) manual pages for information valid for other implementations.

DESCRIPTION

Ld combines several object programs into one; resolves external references; and searches libraries (as created by *ar*(1)). In the simplest case several object *files* are given, and *ld* combines them, producing an object module which can be either executed or 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 made executable if no errors occurred during the load and the –**r** flag was not specified.

The argument routines are concatenated in the order specified. The entry point of the output is the beginning of the first routine (unless the **-e** option is used).

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. 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. If the first member of a library is named **__.SYMDEF**, then it is understood to be a dictionary for the library such as produced by *ranlib*; the dictionary is searched iteratively 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 respectively. It is erroneous to define these symbols.

Ld understands several flag arguments which are written preceded by a –. Except for –**l**, they should appear before the file names.

–**d** Force definition of common storage even if the –**r** flag is present.

–**e** The following argument is taken to be the name of the entry point of the loaded program; location 0x2000 is the default.

–**h** For **-r** output, hide the named symbol by clearing the external bit in the symbol table.

–**l** This option is an abbreviation for a library name. –**l** alone stands for **/lib/libc.a**, which is the standard system library for C and assembly language programs. –**l***x* stands for **/lib/lib***x***.a**, where *x* is a string. If that does not exist, *ld* tries **/usr/lib/lib***x***.a** A library is searched when its name is encountered, so the placement of a –**l** is significant.

–**m** The names of all files and archive members used to create the output file are written to the standard output.

–**n** Arrange that when the output file is executed, the text portion will be read-only and shared among all users executing the file. This involves moving the data areas up to the first possible 512K word boundary following the end of the text.

–**o** The *name* argument after –**o** is used as the name of the *ld* output file, instead of **a.out**.

–**r** Generate 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.

–**R** The next argument is a hexadecimal number which sets the text segment origin. The default

value if 0x2000.

–s "Strip" the output, that is, remove the symbol table and relocation bits to save space (but impair the usefulness of the debugger). This information can also be removed by *strip*(1). This option is turned off if there are any undefined symbols.

–u Take the following argument as a symbol and enter it as undefined in the symbol table. This is useful for loading wholly from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

–V The *num* argument is taken as a decimal version number identifying the **a.out** that is produced. *Num* must be in the range 0–32767. The version stamp is stored in the **a.out** header; see *a.out*(5).

–x Do not preserve local (non-**.globl**) symbols in the output symbol table; only enter external symbols. This option saves some space in the output file.

FILES

/lib/lib?.a	libraries
/usr/lib/lib?.a	more libraries
a.out	output file

SEE ALSO

ar(1), as(1), cc(1), a.out(5).

NAME

ld – link editor

SYNOPSIS

ld [[option] ... [file] ...] ...

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: HP

Remarks: This manual page describes *ld* as implemented on the Series 500 computers. Refer to other *ld*(1) manual pages for information valid for other implementations.

DESCRIPTION

Ld combines one or more object files into a single file. In so doing, it resolves references between files (such as procedure calls and accesses of COMMON). The linker can produce either executable (the default) or relocatable output. An executable file is one that is loadable by the HP-UX program loader, *exec*. A relocatable file is one that is suitable as input for future re-linking (see **–r** below). The linker will not generate an output file if any errors occur during its operation.

Ld recognizes two kinds of input files: ordinary files created by the compilers or assembler (called **.o** files) and archives of these object files, called libraries (see *ar*(1)). In turn, there are two types of libraries: ordinary and indexed. An ordinary library is an archive (of *a.out* format files), with no embellishments. An indexed library is an archive generated by the *ar* command which has also been processed by *ranlib*(1). The first element of such a library is an index (or table of contents) of all the external symbols defined by object modules within the archive.

Ld processes files in the same order as they appear on the command line. The linker includes code and data from a library element if, *and only if*, that object module defines a (*currently* unresolved) reference within the user's program. *Ld* scans an ordinary library once, from beginning to end. It may be necessary to name such a library several times in order to pick up all necessary definitions from them. *Ld* scans an indexed library until it has picked up *all* definitions needed by the user's program. However, if the index is out of date (which can happen if you copy the library or insert/delete elements and don't reprocess it with *ranlib*(1)), the linker ignores the index and treats the library like an ordinary one.

Ld automatically updates any debugger support tables present in the input files (as long as **–s** does not appear on the command line). If you intend to use *cdb*(1) on a program, you should link with it the appropriate run-time start-up file as the first file and **end.o** as the last file named in the command line (see the examples below). Note that *cdb*(1) will not operate on a program marked as shared.

Ld options may occur anywhere on the command line as long as they follow the command name itself. Some options require a modifier following the option letter (e.g. **–o** *outputname*). *Ld* recognizes modifiers either as part of the word containing the option letter, or as a separate word following the option letter (the same convention used by *getopt*(3)).

Ld recognizes the following options. In these descriptions a colon following the option letter indicates that it takes an additional modifier. The colon itself is *not* literal, and must not appear on the command line.

–A keep the D-data and I-data areas apart (in separate segments).

–M: merge input code segments together. The modifier indicates an (approximate) upper bound on the size of the merged code segments in the output file.

–N mark the program as normal (not shared) executable.

–T put the D-data and I-data areas together in the same global data segment (GDS).

–V: use the specified number as a version number and store it in the output file header. This is not the same as any version number reported by the SCCS *what*(1) command.

–X: indicates the initial size for the linker's global symbol table. Thus you can improve linker performance for very large programs (i.e. those with very many external symbols). The linker expands its internal data structures as necessary no matter what the starting size of these structures.

–d force definition of COMMON storage (i.e. assign addresses and sizes), even for **–r** output.

–e: names an alternate entry point for the user program. The HP-UX loader calls this entry point whenever you run your linked program. The default is **_main**.

–h: prior to writing the output file, mark the named symbol so that it is no longer externally visible. This prevents the symbol from being treated as a global definition in future linking. Use this in conjunction with **–r**.

–l: is a shorthand form for specifying libraries in certain directories. *Ld* searches a default set of directories to locate the desired library. These directories are:

/lib
/usr/lib

The linker searches these directories in the above order, looking for an archive named **lib***xxx***.a**, where *xxx* is a string of one or more ASCII characters specified as the modifier for the **–l** option. A **–l** with no modifier (which is only valid at the end of the command line) is the same as **–l**c.

–n mark the program as shared. (This is the default; the option is here to maintain compatibility with other linkers.)

–o: specify an alternate name for the resulting output object file produced by the linker. The default is **a.out**.

–r arranges for the output file to be re-linkable. *Ld* preserves all symbol table and relocation information necessary for linking. In this case the linker does not make final definitions of COMMON, nor does it issue "undefined symbol" messages. This option overrides the actions of **–s** if both options are present. Note: the resulting file is *not* executable.

–s indicates that the output file should not contain symbol table, relocation, or debug support information. This option prevents processing of any symbolic debugger support information in a program. Note: The linker ignores this option if the **–r** option is also present. (You can also use *strip*(1) to remove this information from an existing file.)

–t print a trace of the files (including library components) that *ld* is including in the final output. This information appears twice – once for each pass the linker makes over the input files. Because *cc*(1) also has a **–t** option, you can only exercise this capability by invoking *ld* directly.

–u: specifies a name to enter in the linker's symbol table as undefined. It appears as an unresolved reference to the linker. This provides a way of loading something from a library without having an explicit reference to it in any ".o" files.

–v (verbose) causes the linker to display additional information regarding its operation. This option is not available when linking via *cc*(1) because *cc* intercepts the option as its own.

DEFAULTS

Unless otherwise noted, *ld* names its output **a.out**. The **–o** option overrides this. Similarly, the linker searches for **_main** (written as *main* in C) as the main entry point for a user program. Use the **–e** option to specify a different entry point.

Default memory management attributes for executable files are as follows: shared text, virtual code, virtual data (D-data and I-data), and paged I-data. The only exception is that debuggable programs are marked as normal, rather than shared. Use *chatr*(1) to change these attributes, or enable new ones, such as demand-load.

EXAMPLES

This example links part of a C program for later processing by *ld*. Note the ".o" suffix for the output object file. This is an HP-UX convention for indicating a linkable object file. It also marks the version number of the file as 2.

```
ld –V2 –r c1.o c2.o –o prog.o
```

This example links a simple FORTRAN program for use with the *cdb*(1) symbolic debugger. The output file name is *a.out* since there is no **-o** option in the command line.

```
ld –e start /lib/frt0.o user.o –lI77 –lF77 –lm –lc /usr/lib/end.o
```

This command links a Pascal program so that all its data resides in a single data segment and code segments are merged to be no larger than 16 000 bytes in size.

```
ld –T –M16000 –e start /lib/prt0.o main.o util.o –lpc –lm –lc
```

FILES

837u /lib/crt0.o run-time start-up for C
/lib/frt0.o
run-time start-up for FORTRAN /lib/prt0.o run-time start-up for Pascal /usr/lib/end.o special file for use by *cdb*(1)

SEE ALSO

ar(1), cc(1), cdb(1), chatr(1), fc(1), nm(1), pc(1), ranlib(1), strip(1), end(3), getopt(3), a.out(5).

DIAGNOSTICS

Ld returns the following exit codes:

0 – no errors;
1 – abort (killed by signal);
2 – error during link;

NOTES

Ld recognizes several names as having special meanings. These names – **_end**, **_edata**, and **_etext** (**end**, **edata**, and **etext** in C) – are reserved. (See section 3 of the HP-UX Reference manual for more information.) User programs must not contain (externally-visible) definitions for these names. Also, remember that there are several names already defined in the various run-time start-up files, the most common being **start** (only accessible via assembly language).

NAME

lex – generate programs for lexical analysis of text

SYNOPSIS

lex [–**rctvn**] [file] ...

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Lex generates programs to be used in simple lexical analysis of text.

The input *files* contain strings and expressions to be searched for, and C text to be executed when strings are found. Multiple files are treated as a single file. If no files are specified, the standard input is used.

A file **lex.yy.c** is generated which, when loaded with the library, copies the input to the output except when a string specified in the file is found; then the corresponding program text is executed. The actual string matched is left in *yytext*, an external character array. Matching is done in order of the strings in the file. The strings may contain square brackets to indicate character classes, as in **[abx–z]** to indicate **a**, **b**, **x**, **y**, and **z**; and the operators *****, **+**, and **?** mean respectively any non-negative number of, any positive number of, and either zero or one occurrences of, the previous character or character class. The character **.** is the class of all ASCII characters except new-line. Parentheses for grouping and vertical bar for alternation are also supported. The notation *r*{*d*,*e*} in a rule indicates between *d* and *e* instances of regular expression *r*. It has higher precedence than |, but lower than *****, **?**, **+**, and concatenation. The character ^ at the beginning of an expression permits a successful match only immediately after a new-line, and the character **$** at the end of an expression requires a trailing new-line. The character / in an expression indicates trailing context; only the part of the expression up to the slash is returned in *yytext*, but the remainder of the expression must follow in the input stream. An operator character may be used as an ordinary symbol if it is within " symbols or preceded by \. Thus **[a–zA–Z]+** matches a string of letters.

Three subroutines defined as macros are expected: **input()** to read a character; **unput(***c***)** to replace a character read; and **output(***c***)** to place an output character. They are defined in terms of the standard streams, but you can override them. The program generated is named **yylex()**, and the library contains a **main()** which calls it. The action REJECT on the right side of the rule causes this match to be rejected and the next suitable match executed; the function **yymore()** accumulates additional characters into the same *yytext*; and the function **yyless(***p***)** pushes back the portion of the string matched beginning at *p*, which should be between *yytext* and *yytext*+*yyleng*. The macros *input* and *output* use files **yyin** and **yyout** to read from and write to, defaulted to **stdin** and **stdout**, respectively.

Any line beginning with a blank is assumed to contain only C text and is copied; if it precedes **%%** it is copied into the external definition area of the **lex.yy.c** file. All rules should follow a **%%**, as in YACC. Lines preceding **%%** which begin with a non-blank character define the string on the left to be the remainder of the line; it can be called out later by surrounding it with {}. Note that curly brackets do not imply parentheses; only string substitution is done.

The flags, which must appear before any *files*, are as follows:

- **–r** indicates *ratfor*(1) actions;
- **–c** indicates C actions – this is the default;
- **–t** causes the **lex.yy.c** program to be written instead to the standard output;
- **–v** provides a one-line summary of statistics for the machine generated;
- **–n** suppresses printing of the – summary.

Certain table sizes for the resulting finite state machine can be set in the definitions section:

%p *n* number of positions is *n* (default is 2000);

%n *n* number of states is *n* (default is 500);

%t *n* number of parse tree nodes is *n* (default is 1000);

%a *n* number of transitions is *n* (default is 3000).

The use of one or more of the preceding table options automatically implies **–v**, unless **–n** is specified.

External names generated by *lex* all begin with the prefix **yy** or **YY**.

EXAMPLE

```
D           [0–9]
%%
if          printf("IF statement\n");
[a–z]+      printf("tag, value %s\n",yytext);
0{D}+       printf("octal number %s\n",yytext);
{D}+        printf("decimal number %s\n",yytext);
"++"        printf("unary op\n");
"+"         printf("binary op\n");
"/*"        {       loop:
                    while (input() != '*');
                    switch (input())
                            {
                            case '/': break;
                            case '*': unput('*');
                            default: go to loop;
                            }
                    }
```

SEE ALSO

yacc(1).

Lex – Lexical Analyzer Generator in *HP-UX Concepts and Tutorials.*

BUGS

The **–r** option is not yet fully operational.

NAME

LIF – Logical Interchange Format description

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: HP

DESCRIPTION

LIF (Logical Interchange Format) is a Hewlett-Packard standard disc format that may be used for interchange of files among various HP computer systems. A LIF volume contains a header (identifying it as a LIF volume) and a directory that defines the contents (i.e. files) of the volume. The size of the directory is fixed when the volume is initialized (see *lifinit*(1)) and sets an upper bound on the number of files that may be created on the volume.

HP-UX contains a set of utilities (referred to hereafter as *lif**(1)) that may be used to initialize a LIF volume (i.e. create a header and an empty directory), copy files to and from LIF volumes, list the contents of LIF volumes, remove LIF files, and rename LIF files.

The *lif**(1) utilities are the only utilities within HP-UX where the internal contents of a LIF volume are known. To the rest of HP-UX a LIF volume is simply a file containing some unspecified data. The term 'LIF volume' should in no way be confused with the HP-UX notion of a file system volume or mountable volume.

A LIF volume may be created on any HP-UX file (either regular disc file or device special file) that supports random access via *lseek*(2). Note that you should **not** mount the special file before using the *lif**(1) routines. See *lifinit*(1) for details. Within a LIF volume, individual files are identified by 1 to 10 character file names. File names may consist of upper-case alphanumeric characters (A through Z, 0 through 9) and the underscore character (_). The first character of a LIF file name must be a letter. The *lif**(1) utilities will accept any file name, including illegal file names generated on other systems, but will only create legal names. For example, file names containing lower-case letters will be read but not created.

LIF file names are specified to the *lif**(1) utilities by concatenating the HP-UX path name for the LIF volume with the LIF file name, separating the two with a colon (:). For example,

/dev/fd.0:ABC	specifies LIF file ABC within HP-UX device special file /dev/fd.0.
myfile:ABC	specifies LIF file ABC within HP-UX disc file 'myfile'.

Note that this file naming convention is applicable only for use as arguments to the *lif**(1) utilities and do not constitute legal path names for any other use within HP-UX.

HARDWARE DEPENDENCIES

Series 500:

You **must** use a character special file to access the media.

SEE ALSO

lifcp(1), lifinit(1), lifls(1), lifrename(1), lifrm(1).

NAME

lifcp – copy to or from LIF files

SYNOPSIS

lifcp [**–a**] [**–b**] [**–t**] file1 file2
lifcp [**–a**] [**–b**] [**–t**] file1 [file2 ...] directory

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: HP

DESCRIPTION

Lifcp copies a LIF file to an HP-UX file, an HP-UX file to a LIF file, or a LIF file to another LIF file. It also copies a list of (HP-UX/LIF) files to a (LIF/HP-UX) directory. The last name on the argument list is the destination file or directory.

The options are:

–a causes an ASCII copy. In case of HP-UX to LIF copy, a LIF ASCII file is created.

–b forces binary copy, creating LIF BINARY files. If neither of the **–a** or **–b** options are given, **lifcp** will guess whether to do an ASCII or a binary copy based on the magic number of the file. Text files are ASCII and object files are treated as binary.

–t causes HP-UX file names to be translated to a name acceptable by a LIF utility. That is, all the lower-case letters are converted to upper-case letters, and all other characters (except digits) are converted to an underscore (_). If the HP-UX file name starts with a non-letter, the file name is preceded by X. Note that if there are two files named colon (:) and semicolon (;), both of them are translated to "X_". File names are truncated to a maximum of 10 characters.

A LIF file name is recognized by the embedded colon (:) delimiter (see *lif*(1) for LIF file naming conventions). A LIF directory is recognized by a trailing colon. If an HP-UX file name containing a colon is used, the colon must be escaped with two backslash characters (\\) (the shell removes one of them). The file name '–' (dash) is interpreted to mean standard in or standard out, depending on its position in the argument list.

The LIF file naming conventions are known only by the LIF utilities. Since file name expansion is done by the shell, this mechanism cannot be used for expansion of LIF file names.

Note that the media should **not** be mounted before using *lifcp*.

HARDWARE DEPENDENCIES

Series 500:

You **must** use a character special file to access the media.

EXAMPLES

lifcp abc liffile:CDE
copy HP-UX file *abc* to LIF file *CDE* within HP-UX file *liffile* (*liffile* must have been previously initialized as a LIF volume with *lifinit*). This type of operation is commonly done to create a LIF volume within an HP-UX file, which, when completed, can be copied out to some medium in one step.

lifcp abc\\: liffile:CDE
copy LIF file *abc:* to LIF file *CDE* in *liffile*.

lifcp –t abc def liffile:
copy HP-UX files *abc* and *def* to LIF files *ABC* and *DEF* within *liffile*.

lifcp liffile:ABC .
copy LIF file *ABC* within *liffile* to HP-UX file *ABC* within the current directory.

lifcp – /dev/rfd.0:A_FILE
copy the standard input to LIF file *A_FILE* on LIF volume /dev/rfd.0.

lifcp liffile:ABC /dev/rfd.0:CDE
copy LIF file *ABC* in *liffile* to LIF file *CDE* in /dev/rfd.0.

lifcp liffile:ABC –
copy LIF file *ABC* in *liffile* to standard out.

lifcp * liffile:
copy all files within the current HP-UX directory to LIF files of the same name on LIF volume *liffile* (may cause errors if file names in the current directory do not obey LIF naming conventions!).

SEE ALSO
lif(1), lifinit(1), lifls(1), lifrename(1), lifrm(1).

DIAGNOSTICS
Lifcp returns exit code 0 if the file is copied successfully. Otherwise it prints a diagnostic and returns non-zero.

NAME

lifinit – write LIF volume header on file

SYNOPSIS

lifinit [–**v**nnn] [–**d**nnn] [–**n** string] file

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: HP

DESCRIPTION

Lifinit writes a LIF volume header on a volume or file. *Option*s may appear in any order. Their meanings are:

–**v***nnn* Sets the volume size to *nnn* bytes. If *nnn* is not a multiple of 1024, it will be rounded down to the next such multiple.

–**d***nnn* Sets the directory size to *nnn* file entries. If *nnn* is not a multiple of 8, it will be rounded up to next such multiple.

–**n** *string* sets the volume name to be *string*. If the **–n** option is omitted, the volume name is set to the last component of the path name specified by *file*. If *string* is longer than 6 characters, it is truncated.

If *file* does not exist, a regular HP-UX disc file is created and initialized.

The default values for volume size are 256K bytes for regular files, and the actual capacity of the device (as determined by *volsize*(3)) for device files.

The default directory size is a function of the volume size. A percentage of the volume size is allocated to the volume directory as follows:

VOLUME SIZE	DIRECTORY SIZE
< 2MB	~1.3%
> 2MB	~0.5%

Each directory entry occupies 32 bytes of storage. The actual directory space is subject to the rounding rules stated above.

Note that you should **not** mount the special file before using *lifinit*.

HARDWARE DEPENDENCIES

Series 200:

If your media has never been initialized, it must be initialized using *system_mi* before *lifinit* can be used. (Refer to the System Administrator Manual for details concerning *system_mi*.)

Series 500:

You **must** use a character special file to access the media.

If your media has never been initialized, it must be initialized using *sdfinit*(8) before *lifinit* can be used.

EXAMPLES

lifinit –v500000 –d10 x
lifinit /dev/rfd.0

SEE ALSO

lif(1), lifcp(1), lifls(1), lifrename(1), lifrm(1), sdfinit(8).

DIAGNOSTICS

Lifinit returns exit code 0 if the volume is initialized successfully. Otherwise it prints a diagnostic and returns non-zero.

WARNING

Do not terminate *lifinit* once it has started executing. Otherwise, your media could become corrupted.

NAME

lifls – list contents of LIF directory

SYNOPSIS

lifls [option] name

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: HP

DESCRIPTION

Lifls lists the contents of a LIF directory on *stdout*. The default output format calls for the file names to be listed in multiple columns (as is done by *ls*(1), except unsorted) if *stdout* is a character special file. If *stdout* is not a teletype, the output format is one file name per line. *Name* is a path name to an HP-UX file containing a LIF volume and optional file name. If *name* is a volume name, the entire volume is listed. If *name* is of the form *volume:file*, then only the file is listed. There are two options:

–l List in long format, giving volume name, volume size, directory start, directory size, file type, file size, file start, extension, date created, last volume and volume number.

–C Force multiple column output format regardless of *stdout* type.

Note that you should **not** mount the special file before using *lifls*.

HARDWARE DEPENDENCIES

Series 500:

You **must** use a character special file to access the media.

EXAMPLES

lifls –l ../TEST/header
lifls /dev/rfd.0

SEE ALSO

lif(1), lifcp(1), lifinit(1), lifrename(1), lifrm(1).

DIAGNOSTICS

Lifls returns exit code 0 if the directory was listed successfully. Otherwise it prints a diagnostic and returns non-zero.

NAME

lifrename – rename LIF files

SYNOPSIS

lifrename oldfile newfile

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: HP

DESCRIPTION

Oldfile is a full LIF file specifier (see *lif*(1) for details) for the file to be renamed (e.g. **liffile:A_FILE**). *Newfile* is the new name to be given to the file (only the file name portion). This operation does not include copy or delete. Old file names must match the name of the file to be renamed, even if that file name is not a legal LIF name.

Note that you should **not** mount the special file before using *lifrename*.

HARDWARE DEPENDENCIES

Series 500:

You **must** use a character special file to access the media.

EXAMPLES

lifrename liffile:A_FILE B_FILE
lifrename /dev/fd.0:ABC CDE

SEE ALSO

lif(1), lifcp(1), lifinit(1), lifls(1), lifrm(1).

DIAGNOSTICS

Lifrename returns exit code 0 if the file name is changed successfully. Otherwise it prints a diagnostic and returns non-zero.

NAME

lifrm – remove a LIF file

SYNOPSIS

lifrm file1 ... filen

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: HP

DESCRIPTION

Lifrm removes one or more entries from a LIF volume. File name specifiers are as described in *lif*(1).

Note that you should **not** mount the special file before using *lifrm*.

HARDWARE DEPENDENCIES

Series 500:

You **must** use a character special file to access the media.

EXAMPLES

lifrm liffile:MAN
lifrm /dev/rfd.0:F

SEE ALSO

lif(1), lifcp(1), lifinit(1), lifls(1), lifrename(1).

DIAGNOSTICS

Lifrm returns exit code 0 if the file is removed successfully. Otherwise it prints a diagnostic and returns non-zero.

NAME

line – read one line from user input

SYNOPSIS

line

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Line 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** and always prints at least a new-line. It is often used within shell files to read from the user's terminal.

SEE ALSO

read (documented under sh(1)), read(2).

NAME

link, unlink – exercise link and unlink system calls

SYNOPSIS

/etc/link file1 file2
/etc/unlink file

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Link and *unlink* perform their respective system calls on their arguments, abandoning most error checking. These commands may only be executed by the super-user.

RETURN VALUE

0 - successful *link*.
1 - input syntax error.
2 - *link* call failed (*unlink* will never report failure).

SEE ALSO

rm(1), link(2), unlink(2).

NAME

lint – a C program checker/verifier

SYNOPSIS

lint [**–abchnpuvxDUI**] file ...

HP-UX COMPATIBILITY

Level: C Compiler – HP-UX/STANDARD

Origin: System III

DESCRIPTION

Lint attempts to detect features of the C program *files* which are likely to be bugs, non-portable, or wasteful. It also checks type usage more strictly than the compilers. Among the things which 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 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.

It is assumed that all the *files* are to be loaded together; they are checked for mutual compatibility. By default, *lint* uses function definitions from the standard lint library **llib–lc.ln**; function definitions from the portable lint library **llib–port.ln** are used when *lint* is invoked with the **–p** option.

Any number of *lint* options may be used, in any order. The following options are used to suppress certain kinds of complaints:

–a Suppress complaints about assignments of long values to variables that are not long.

–b Suppress complaints about **break** statements that cannot be reached. (Programs produced by *lex* or *yacc* will often result in a large number of such complaints.)

–c Suppress complaints about casts that have questionable portability.

–h Do not apply heuristic tests that attempt to intuitively find bugs, improve style, and reduce waste.

–u Suppress complaints about functions and external variables used and not defined, or defined and not used. (This option is suitable for running *lint* on a subset of files of a larger program.)

–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 *lint's* behavior:

–n Do not check compatibility against either the standard or the portable lint library.

–p Attempt to check portability to other dialects of C.

The **–D**, **–U**, and **–I** options of *cc*(1) are also recognized as separate arguments.

Certain conventional comments in the C source will 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.

/*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.

Lint produces its first output on a per source file basis. Complaints regarding included files are collected and printed after all source files have been processed. Finally, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a complaint stems from a given source file or from one of its included files, the source file name will be printed followed by a question mark.

FILES

/usr/lib/lint[12]	programs
/usr/lib/llib–lc.ln	declarations for standard functions (binary format; source is in **/usr/lib/llib–lc**)
/usr/lib/llib–port.ln	declarations for portable functions (binary format; source is in **/usr/lib/llib–port**)
/usr/tmp/*lint*	temporaries

SEE ALSO

cc(1).

BUGS

Exit(2) and other functions which do not return are not understood.

NAME

login – sign on

SYNOPSIS

/etc/login [name [hangup]]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III, 4.2BSD

DESCRIPTION

The *login* command is used at the beginning of each terminal session and allows you to identify yourself to the system. It cannot be invoked explicitly (except by the super-user), but is invoked by the system when a connection is first established, or after the previous user has logged out by sending an "end-of-file" (control–D) to his or her initial shell. (See *How to Get Started* at the beginning of this volume for instructions on how to dial up initially.)

Login asks for your user name (preferably lower-case), and, if appropriate, your password. Echoing is turned off (where possible) during the typing of your password, so it does not appear on the written record of the session. Note that, if you have a password, you are asked for it whether your user name is valid or not. This is done to make it more difficult for an unauthorized user to log in on the system by trial and error.

If password aging has been invoked by the super-user on your behalf, your password may have expired. In this case, you are diverted into *passwd*(1) to change it, after which you may attempt to login again.

If you do not complete the login successfully within a certain period of time (e.g., one minute), you are likely to be silently disconnected.

After a successful login, the accounting files are updated, and the user and group id's and the working directory are initialized. *Login* then executes a command interpreter (usually *sh*(1)), according to specifications found in the **/etc/passwd** file. If the shell is executed, the profile files **/etc/profile** and **$HOME/.profile** are executed, if they exist. Depending on what these profile files contain, you are notified of mail in your mail file or any messages you may have received since your last login. Argument 0 of the command interpreter is – followed by the last component of the interpreter's path name. The *environment* (see *environ*(7)) is initialized to:

HOME = *your-login-directory*
PATH = :/bin:/usr/bin
LOGNAME = *your-login-name*

For the super-user, PATH is augmented to include **/etc**.

The presence of *name* suppresses the **login:** prompt, and uses *name* as the login name. *Hangup* is the time, in seconds, before hanging up if the login is unsuccessful. The default is 60 seconds. Zero (0) seconds indicates an indefinite wait.

If **/usr/adm/btmp** is present, all unsucessful login attempts are logged to this file. This feature is disabled if the file is not present. A summary of bad login attempts may be viewed using *lastb* (see *last*(1)).

If **/etc/securetty** is present, login security is in effect and the super-user may only login successfully on the ttys listed in this file. Ttys are listed by device name, one per line. Valid tty names are dependent on installation. Some examples could be "console", "tty01", "ttya1", etc. Note that this feature does not inhibit a normal user from using *su*.

After three unsuccessful logins attempts, a *hangup* signal is issued.

FILES

/etc/utmp	users currently logged in

/usr/adm/wtmp	history of logins, logouts, and date changes
/usr/adm/btmp	history of bad login attempts
/usr/mail/*your-name*	mailbox for user *your-name*
/etc/motd	message-of-the-day
/etc/passwd	password file – defines users, passwords, and groups
/etc/profile	system profile (initialization for all users)
/etc/securetty	list of valid ttys for root login
$HOME/.profile	personal profile (individual user initialization)

SEE ALSO

last(1), mail(1), newgrp(1), passwd(1), sh(1), su(1), passwd(5), profile(5), utmp(5), environ(7), getty(8).

DIAGNOSTICS

Login incorrect
: if the user name or the password is incorrect.

No shell, cannot open password file, no directory:
: consult your system manager.

Your password has expired. Choose a new one.
: if password aging is implemented.

No entry in utmp:
: *utmp* file exists but your terminal is not listed there. Caused by serious trouble in the file system or *ttyname*(3).

No root directory:
: attempted to log into a subdirectory that does not exist (i.e., passwd file entry had shell name "*", but the system cannot *chroot* to the given directory).

No login in /etc or /bin on root:
: same as above except sub-root login command not found.

Invalid ID:
: *setuid* or *setgid* failed.

No directory:
: cannot *chdir* to your home directory.

No shell:
: your shell (or /bin/sh if your shell name is null in /**etc**/**passwd**) could not be *exec*'d.

NAME

logname – get login name

SYNOPSIS

logname

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Logname returns the contents of the environment variable **$LOGNAME**, which is set when a user logs into the system.

FILES

/etc/profile

SEE ALSO

env(1), login(1), logname(3X), environ(7).

NAME

lorder – find ordering relation for object library

SYNOPSIS

lorder file ...

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

The input is one or more object or library archive *files* (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-line example intends to build a new library from existing **.o** files.

```
ar cr library `lorder *.o | tsort`
```

Ranlib(1) serves a similar purpose and is more efficient for libraries.

FILES

*symref, *symdef temp files

SEE ALSO

ar(1), ld(1), tsort(1), ranlib(1).

BUGS

Object files whose name do not end with **.o**, even when contained in library archives, are overlooked. Their global symbols and references are attributed to some other file.

NAME

lp, lpr, cancel – send or cancel requests to an LP line printer

SYNOPSIS

lp [–**c**] [–**d**dest] [–**m**] [–**n**number] [–**o**option] [–**s**] [–**t**title] [–**w**] files
lpr [–**c**] [–**d**dest] [–**m**] [–**n**number] [–**o**option] [–**s**] [–**t**title] [–**w**] files
cancel [ids] [printers]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Lp 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. The file name – stands for the standard input and may be supplied on the command line in conjunction with named *files*. The order in which *files* appear is the same order in which they will be printed.

Lp associates a unique *id* with each request and prints it on the standard output. This *id* can be used later to cancel (see *cancel*) or find the status (see *lpstat*(1)) of the request.

The following options to *lp* may appear in any order and may be intermixed with file names:

–**c** Make copies of the *files* to be printed immediately when *lp* is invoked. Normally, *files* will not be copied, but will be 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, in the absence of the –**c** option, any changes made to the named *files* after the request is made but before it is printed will be reflected in the printed output.

–**d***dest* Choose *dest* as the printer or class of printers that is to do the printing. If *dest* is a printer, then the request will be printed only on that specific printer. If *dest* is a class of printers, then the request will be printed on the first available printer that is a member of the class. Under certain conditions (printer unavailability, file space limitation, etc.), requests for specific destinations may not be accepted (see *accept*(8) and *lpstat*(1)). By default, *dest* is taken from the environment variable **LPDEST** (if it is set). Otherwise, a default destination (if one exists) for the computer system is used. Destination names vary between systems (see *lpstat*(1)).

–**m** Send mail (see *mail*(1)) after the files have been printed. By default, no mail is sent upon normal completion of the print request.

–**n***number* Print *number* copies (default = 1) of the output.

–**o***option* Specify printer-dependent or class-dependent *options*. Several such *options* may be collected by specifying the –**o** keyletter more than once. For more information about what is valid for *options*, see *Models* in *lpadmin*(8).

–**s** Suppress messages from *lp*(1) such as "request id is ... ".

–**t***title* Print *title* on the banner page of the output.

–**w** Write a message on the user's terminal after the *files* have been printed. If the user is not logged in, then mail will be sent instead.

Cancel cancels line printer requests that were made by the *lp*(1) command. The command line arguments may be either request *ids* (as returned by *lp*(1)) or *printer* names (for a complete list, use *lpstat*(1)). Specifying a request *id* cancels the associated request even if it is currently printing. Specifying a *printer* cancels the request which is currently printing on that printer. In either case, the cancellation of a request that is currently printing frees the printer to print its next available request.

FILES

/usr/spool/lp/*

SEE ALSO

enable(1), lpstat(1), mail(1), accept(8), lpadmin(8), lpsched(8).

NAME

lpd – line printer daemon

SYNOPSIS

/usr/lib/lpd [printername]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Lpd is the daemon for the line printer. It is automatically initiated by the line printer spooling command, *lpr*.

Printername is the name of a printer device file, without the initial "/dev/" (i.e. **lp8**). If *printername* is not specified, the default printer **lp** is used.

Lpd searches the directory **/usr/spool** for a directory of the same name as the specified *printername*. Thus, **/usr/spool/lp** is used by default. To be able to use other printers, a directory for each printer must be created in **/usr/spool** by the super-user. (Other spool directories can be specified by *lpr* with the **–d** option.)

The file **lock** is used to prevent two daemons from becoming active on the same spool directory. Several daemons can be active simultaneously, as long as they are working on different spool directories. After the program has successfully set the lock, it forks and the main path exits, thus spawning the daemon. The directory is scanned for files beginning with "df". Each such file is submitted as a job. Each line of a job file must begin with a key character to specify what to do with the remainder of the line. The key characters are:

L specifies that the remainder of the line is to be sent as a literal.
I is the same as **L**, but signals the $IDENT card which is to be mailed back by the mail option.
B specifies that the rest of the line is a file name. That file is to be printed.
F is the same as **B** except a form-feed is prepended to the file.
U specifies that the rest of the line is a file name. After the job has been transmitted, the file is unlinked.
M is followed by a user ID; after the job is sent, a message is mailed to the user via the *mail*(1) command to verify the sending of the job.
D specifies that the remainder of the line is a pathname for a specific printer.

Any error encountered will cause the daemon to give up, wait 10 seconds, and start over. This means that an improperly constructed "df" file may cause the same job to be submitted every 10 seconds.

To restart *lpd* (in the case of hardware or software malfunction), it is necessary to first kill the old daemon (if it is still alive), and remove the lock file (if present), before initiating the new daemon. This is done automatically by **/etc/rc** when the system is brought up, in case there were any jobs left in the spooling directory when the system last went down.

Lpd will pass ASCII escape sequences to the output device. This enables users to access special capabilities like expanded type fonts, alternate character sets, etc.

FILES

/usr/spool/lp/*	default spool area for line printer daemon.
/usr/spool/*printername*/*	spool area for additional printers
/etc/passwd	used to get the user's name.
/dev/lp	default line printer device.
/dev/lp*	additional printer devices.

SEE ALSO

lpr(1).

NAME

lpr – line printer spooler

SYNOPSIS

lpr [option ...] [name ...]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Lpr causes the named files to be queued for printing on a line printer. If no *name*s appear, the standard input is assumed; thus *lpr* may be used as a filter. If the **–d** option is absent, the printer /**dev**/**lp** is assumed.

The following *option*s may be given (each as a separate argument and in any order) before any file name arguments:

–c Makes a copy of the file to be sent before returning to the user.
–r Removes the file after sending it.
–m When printing is complete, reports that fact by *mail*(1).
–n Does not report the completion of printing by *mail*(1). This is the default option.
–d Is followed by a printer name, and causes output to be sent to that printer.

Note that *lpr* does not force a page-eject at the end of a printing. Thus, if printing stops in the middle of a page, the upcoming header will not begin at the appropriate place. The most common solution to this is to pipe your printing through *pr*(1).

FILES

/etc/passwd	user's identification and accounting data.
/usr/lib/lpd	line printer daemon.
/usr/spool/*printername*/*	spool area.

SEE ALSO

lp(1), lpd(1), pr(1).

WARNING

Lpr is an obsolete command. The use of *lp*(1) is encouraged instead.

NAME

lpstat – print LP status information

SYNOPSIS

lpstat [options]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Lpstat prints information about the current status of the LP line printer system.

If no *options* are given, then *lpstat* prints the status of all requests made to *lp*(1) by the user. Any arguments that are not *options* are assumed to be request *ids* (as returned by *lp*). *Lpstat* prints the status of such requests. *Options* may appear in any order and may be repeated and intermixed with other arguments. Some of the keyletters below may be followed by an optional *list* that can be in one of two forms: a list of items separated from one another by a comma, or a list of items enclosed in double quotes and separated from one another by a comma and/or one or more spaces. For example:

```
–u "user1, user2, user3"
```

The omission of a *list* following such keyletters causes all information relevant to the keyletter to be printed. For example:

```
lpstat –o
```

prints the status of all output requests.

–a[*list*] Print acceptance status (with respect to *lp*) of destinations for requests. *List* is a list of intermixed printer names and class names.

–c[*list*] Print class names and their members. *List* is a list of class names.

–d Print the system default destination for *lp*.

–o[*list*] Print the status of output requests. *List* is a list of intermixed printer names, class names, and request *ids*.

–p[*list*] Print the status of printers. *List* is a list of printer names.

–r Print the status of the LP request scheduler

–s Print a status summary, including the status of the line printer scheduler, the system default destination, a list of class names and their members, and a list of printers and their associated devices.

–t Print all status information.

–u[*list*] Print status of output requests for users. *List* is a list of login names.

–v[*list*] Print the names of printers and the pathnames of the devices associated with them. *List* is a list of printer names.

FILES

/usr/spool/lp/*

SEE ALSO

enable(1), lp(1).

NAME

ls, l, ll, lsf, lsr, lsx – list contents of directories

SYNOPSIS

ls [–**abcdfgilmnoqrstux1ACFR**] [names]
l [ls options] [names]
ll [ls options] [names]
lsf [ls options] [names]
lsr [ls options] [names]
lsx [ls options] [names]

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III and UCB

DESCRIPTION

For each directory named, *ls* lists the contents of that directory; for each file named, *ls* repeats its name and any other information requested. By default, the output is sorted alphabetically. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but file arguments are processed before directories and their contents.

If you are the super-user, *ls* defaults to listing all files except . and ...

There are three major listing formats. The format chosen depends on whether the output is going to a login device, and may also be controlled by option flags. The default format for a teletype is to list the contents of directories in multi-column format, with the entries sorted down the columns. (When individual file names (as opposed to directory names) appear in the argument list, those file names are always sorted across the page rather than down the page in columns. This is because the individual file names may be arbitrarily long.) If the standard output is not a teletype, the default format is to list one entry per line. Finally, there is a stream output format in which files are listed across the page, separated by "," characters. The –**m** flag enables this format.

There are several options:

–**a** List all entries; in the absence of this option, entries whose names begin with a period (.) are *not* listed.

–**A** The same as –**a**, except that the current directory "." and parent directory ".." are not listed. For the super-user, this flag defaults to ON, and is turned off by –**A**.

–**b** Force printing of non-graphic characters to be in \ddd notation, in octal.

–**c** Use time of last modification of the inode (mode, etc.) instead of last modification of the file for sorting (–**t**) and/or printing (–**l**).

–**C** Force multi-column output to a file or a pipe.

–**d** If argument is a directory, list only its name; often used with –**l** to get the status of a directory.

–**f** Force each argument to be interpreted as a directory and list the name found in each slot. This option turns off –**l**, –**t**, –**s**, and –**r**, and turns on –**a**; the order is the order in which entries appear in the directory.

–**F** Cause directories to be marked with a trailing "/" and executable files to be marked with a trailing "*".

–**g** The same as –**l**, except that the owner is not printed (overrides –**o** if both options are specified).

–**i** For each file, print the i-number in the first column of the report.

–**l** List in long format, giving mode, number of links, owner, group, size in bytes, and time of last modification for each file (see below). If the file is a special file, the size field will contain the

major device number in decimal and the minor device number in hexadecimal, rather than a size.

–m Force stream output format, i.e. a comma separated list.

–n The same as **–l**, except that the owner number is printed, and all group information is omitted.

–o The same as **–l**, except that the group is not printed.

–q Force printing of non-graphic charaters in file names as the character "?"; this normally happens only if the output device is a teletype.

–r Reverse the order of sort to get reverse alphabetic or oldest first, as appropriate.

–R Recursively list subdirectories encountered.

–s Give size in blocks (including indirect blocks) for each entry.

–t Sort by time of last modification (latest first) instead of by name.

–u Use time of last access instead of last modification for sorting (with the **–t** option) and/or printing (with the **–l** option).

–x Force columnar printing to be sorted across rather than down the page.

–1 The file names will be listed in single column format regardless of the output device. This will force single column format to the user's terminal.

Ls normally is known by several names which provide shorthands for the various formats:

l is equivalent to **ls –m**.
ll is equivalent to **ls –l**.
lsf is equivalent to **ls –F**.
lsr is equivalent to **ls –R**.
lsx is equivalent to **ls –x**.

The shorthand notations are implemented as links to ls. Option arguments to the shorthand versions behave exactly as if the long form above had been used with the additional arguments.

The mode printed under the **–l** option consists of 11 characters that are interpreted as follows:

The first character is:

d if the entry is a directory;
b if the entry is a block special file;
c if the entry is a character special file;
p if the entry is a fifo (a.k.a. "named pipe") special file;
– if the entry is an ordinary file.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to the owner's permissions; the next to permissions of others in the user-group of the file; and the last to all others. Within each set, the three characters indicate permission to read, to write, and to execute the file as a program, respectively. For a directory, "execute" permission is interpreted to mean permission to search the directory for a specified file.

The permissions are indicated as follows:

r if the file is readable;
w if the file is writable;
x if the file is executable;
– if the indicated permission is *not* granted.

The group-execute permission character is given as **s** if the file has set-group-ID mode; likewise, the user-execute permission character is given as **s** if the file has set-user-ID mode. The last character of the mode (normally **x** or –) is **t** if the 1000 (octal) bit of the mode is on; see

chmod(1) for the meaning of this mode. The indications of set-ID and 1000 bit of the mode are capitalized if the corresponding execute permission is *not* set.

When the sizes of the files in a directory are listed, a total count of blocks, including indirect blocks, is printed.

HARDWARE DEPENDENCIES

Series 200:

Network and SRM files are not implemented.

Series 500:

The **–a** and **–A** options perform the same function.

FILES

/etc/passwd	to get user IDs for **ls –l** and **ls –o**.
/etc/group	to get group IDs for **ls –l** and **ls –g**.

SEE ALSO

chmod(1), find(1).

BUGS

Newline and tab are considered printing characters in file names.

The output device is assumed to be 80 columns wide.

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 using this setting would make old shell scripts which used *ls* error-prone.

Column widths choices are poor for terminals which can tab.

NAME

lsdev – list device drivers in the system

SYNOPSIS

/**etc/lsdev** [major...]

HP-UX COMPATIBILITY

Level: HP-UX/NON-STANDARD

Origin: HP

Remarks: *Lsdev* is implemented on the Series 500 only.

DESCRIPTION

With no arguments, *lsdev* lists, one pair per line, the major device numbers and driver names of all device drivers configured into the system and available for invocation via special files.

If there are any arguments, they must represent major device numbers. For each, *lsdev* lists the corresponding driver name (if any).

Lsdev is simply a quick-reference aid. In some implementations, it may only read an internal list of device drivers, not the actual list in the operating system.

SEE ALSO

Section 4.

DIAGNOSTICS

Lists the drivername as "no such driver" when appropriate.

NAME

mail, rmail – send mail to users or read mail

SYNOPSIS

mail [**–rpqe**] [**–f** file]

mail persons

rmail persons

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Mail without arguments prints a user's mail, message-by-message, in last-in, first-out order. For each message, the user is prompted with a **?**, and a line is read from the standard input to determine the disposition of the message:

<new-line>	Go on to next message.
+	Same as <new-line>.
d	Delete message and go on to next message.
p	Print message again.
–	Go back to previous message.
s [*files*]	Save message in the named *files* (**mbox** is default).
w [*files*]	Save message, without its header, in the named *files* (**mbox** is default).
m [*persons*]	Mail the message to the named *persons* (yourself is default).
q	Put undeleted mail back in the *mailfile* and stop.
EOT (control-d)	Same as **q**.
x	Put all mail back in the *mailfile* unchanged and stop.
!*command*	Escape to the shell to do *command*.
*	Print a command summary.

The optional arguments alter the printing of the mail:

–r causes messages to be printed in first-in, first-out order.

–p causes all mail to be printed without prompting for disposition.

–q causes *mail* to terminate after interrupts. Normally an interrupt only causes the termination of the printing of the current message.

–f*file* causes *mail* to use *file* (e.g., **mbox**) instead of the default *mailfile*.

–e The mail is simply tested for existence and the exit code returned.
0 = mail present
1 = no mail
2 = other error

When *persons* are named, *mail* takes the standard input up to an end-of-file (or up to a line consisting of just a **.**) and adds it to each *person*'s *mailfile*. The message is preceded by the sender's name and a postmark. Lines that look like postmarks in the message, (i.e., "From . . .") are preceded with a >. A *person* is usually a user name recognized by *login*(1). If a *person* being sent mail is not recognized, or if *mail* is interrupted during input, the **dead.letter** will be saved to allow editing and resending.

To denote a recipient on a remote system, prefix *person* by the system name and exclamation mark (see *uucp*(1C)). Everything after the first exclamation mark in *persons* is interpreted by the remote system. In particular, if *persons* contains additional exclamation marks, it can denote a sequence of machines through which the message is to be sent on the way to its ultimate destination. For example, specifying **a!b!cde** as a recipient's name causes the message to be sent to user **b!cde** on system **a**. System **a** will interpret that destination as a request to send the message to user **cde** on system **b**. This might be useful, for instance, if the sending system can access system **a** but not system **b**, and system **a** has access

to system **b**.

The *mailfile* may be manipulated in two ways to alter the function of *mail*. The *other* permissions of the file may be read-write, read-only, or neither read nor write to allow different levels of privacy. If changed to other than the default, the file will be preserved even when empty to perpetuate the desired permissions. The file may also contain the first line:

Forward to *person*

which will cause all mail sent to the owner of the *mailfile* to be forwarded to *person*. This is especially useful to forward all of a person's mail to one machine in a multiple machine environment.

Rmail only permits the sending of mail; *uucp*(1C) uses *rmail* as a security precaution.

When a user logs in, he is informed of the presence of mail, if any.

FILES

/etc/passwd	to identify sender and locate persons
/usr/mail/*	incoming mail for user *; *mailfile*
$HOME/mbox	saved mail
$MAIL	*mailfile*
/tmp/ma*	temporary file
/usr/mail/*.lock	lock for mail directory
dead.letter	unmailable text

SEE ALSO

login(1), mailx(1), uucp(1C), write(1).

BUGS

Race conditions sometimes result in a failure to remove a lock file.
After an interrupt, the next message may not be printed; printing may be forced by typing a **p**.

NAME

mail – send and receive mail

SYNOPSIS

mailx [**–v**] [**–i**] [**–n**] [**–s** subject] [user ...]
mailx [**–v**] [**–i**] [**–n**] **–f** [name]
mailx [**–v**] [**–i**] [**–n**] **–u** user

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: UCB

DESCRIPTION

Mailx is a intelligent mail processing system, which has a command syntax reminiscent of *ed*(1), with lines replaced by messages.

The options common to all *mailx* invocations are as follows:

–v verbose mode; the details of mail delivery are displayed on your terminal.

–i causes tty interrupt signals to be ignored. This is particularly useful when using *mailx* on noisy phone lines.

–n inhibits the reading of */usr/lib/mailx/mailx.rc*.

Sending Mail

To send a message to one or more other people, *mailx* can be invoked with arguments which are the names of people to send to. You are then expected to type in your message, followed by an EOT (control–D) at the beginning of a line. A *subject* may be specified on the command line by using the **–s** option. (Only the first argument after the **–s** option is used as a subject, so be sure to quote ("...") subjects containing spaces.) The section below, labeled *Replying to or Originating Mail*, describes some features of *mailx* available to help you compose your letter.

Reading Mail

In normal usage, *mailx* is invoked with no arguments, causing *mailx* to check the post office for any mail for you, and then print a one-line header for each message found there. The current message is initially the first message (numbered 1) and can be printed using the **print** command (which can be abbreviated **p**). You can move among the messages much as you move between lines in *ed*, with the commands + and – moving backwards and forwards, and simple numbers typing the addressed message.

Disposing of Mail

After examining a message you can **delete** (**d**) the message or **reply** (**r**) to it. Deletion causes the *mailx* program to forget about the message. This is not irreversible – the message can be **undeleted** (**u**) by giving its number, or the *mailx* session can be aborted by giving the **exit** (**x**) command. However, deleted messages usually disappear, never to be seen again.

Specifying Messages

Commands such as **print** and **delete** can be given a list of message numbers as arguments to apply to a number of messages at once. Thus,

```
delete 1 2
```

deletes messages 1 and 2, while

```
delete 1–5
```

deletes messages 1 through 5. The special name "*" addresses all messages, and "$" addresses the last message. Thus, the command **top** which prints the first few lines of a message could be used in "top *" to print the first few lines of all messages.

Replying to or Originating Mail

You can use the **reply** command to set up a response to a message, sending it back to the person from

which it came. Text you then type in, up to an end-of-file, defines the contents of the message. While you are composing a message, *mailx* treats lines beginning with the character "~" specially. For instance, typing "~m" (alone on a line) places a copy of the current message into the response, right-shifting it by a tabstop. Other escapes set up subject fields, add and delete recipients to the message and allow you to escape to an editor to revise the message or to a shell to run some commands. (These options are given in the summary below.)

Ending a Mail Processing Session

You can end a *mailx* session with the **quit** (**q**) command. Messages which have been examined go to your *mbox* file unless they have been deleted, in which case they are discarded. Unexamined messages go back to the post office. The **–f** option causes *mailx* to read in the contents of your *mbox* (or the specified file) for processing. When you **quit**, *mailx* writes undeleted messages back to this file. The **–u** option is a short way of executing "**mailx –f /usr/mail/user**".

Personal and Systemwide Distribution Lists

It is also possible to create personal distribution lists so that, for instance, you can send mail to "cohorts" and have it go to a group of people. Such lists can be defined by placing a line like

alias cohorts bill ozalp sklower jkf mark cory:kridle

in the file **.mailrc** in your home directory. The current list of such aliases can be displayed by the **alias** (**a**) command in *mailx*. System wide distribution lists can be created by editing */usr/lib/mailx/mailx.rc*. These are kept in a slightly different syntax. In mail you send, personal aliases are expanded in mail sent to others so that they are able to **reply** to the recipients.

Network Mail – UUCP

Refer to *uucp*(1C) for a description of *uucp* network addresses.

Mailx has a number of options which can be **set** in the **.mailrc** file to alter its behavior; thus "set askcc" enables the "askcc" feature. (These options are summarized below.)

Summary

Each command is typed on a line by itself, and may take arguments following the command word. The command need not be typed in its entirety – the first command which matches the typed prefix is used. For the commands which take message lists as arguments, if no message list is given, then the next message forward which satisfies the command's requirements is used. If there are no messages forward of the current message, the search proceeds backwards, and if there are no good messages at all, *mailx* types "No applicable messages" and aborts the command.

–	Goes to the previous message and prints it out. If given a numeric argument *n*, goes to the *n*th previous message and prints it.
?	Prints a brief summary of commands.
!	Executes the HP-UX shell command which follows.
Print	(**P**) Like **print**, but also prints out ignored header fields. See also **print** and **ignore**.
Reply	(**R**) Reply to originator. Does not reply to other recipients of the original message.
Type	(**T**) Identical to the **Print** command.
alias	(**a**) With no arguments, prints out all currently-defined aliases. With one argument, prints out that alias. With more than one argument, creates a new alias or changes an old alias.
alternates	(**alt**) Useful if you have accounts on several machines. **Alt** can be used to inform *mailx* that the listed addresses are really yourself. When you **reply** to messages, *mailx* does not send a copy of the message to any of the addresses listed on the **alternates** list. If the **alternates** command is given with no argument, the current set of alternate names is displayed.

chdir (**cd**) Changes the user's working directory to that specified, if given. If no directory is given, then changes to the user's login directory.

copy (**co**) Does the same thing that **save** does, except that it does not mark the messages it is used on for deletion when you quit.

delete (**d**) Takes a list of messages as arguments and marks them all as deleted. Deleted messages are not saved in ***mbox***, nor are they available for most other commands.

dp (also **dt**) Deletes the current message and prints the next message. If there is no next message, ***mailx*** says "at EOF."

edit (**e**) Takes a list of messages and invokes the text editor for each one in turn. On return from the editor, the message is read back in.

exit (**ex** or **x**) Effects an immediate return to the shell without modifying the user's system mailbox, his ***mbox*** file, or his edit file in **–f**.

file (**fi**) Identical to **folder**.

folders List the names of the folders in your folder directory.

folder (**fo**) Switches to a new mail file or folder. With no arguments, it tells you which file you are currently reading. If you supply an argument, it writes out changes (such as deletions) you have made in the current file, and reads in the new file. Some special conventions are recognized for the name. **#** means the previous file, **%** means your system mailbox, ***%user*** means ***user***'s system mailbox, **&** means your ~/mbox file, and **+folder** means a file in your folder directory.

from (**f**) Takes a list of messages and prints their message headers.

headers (**h**) Lists the current range of headers, which is an 18 message group. If a "+" argument is given, then the next 18 message group is printed, and if a "–" argument is given, the previous 18 message group is printed.

help A synonym for ?.

hold (**ho**, also **preserve**) Takes a message list and marks each message therein to be saved in the user's system mailbox instead of in ***mbox***. Does not override the **delete** command. **ignore** (**i**) Add the list of header fields named to the ***ignored list***. Header fields in the ignore list are not printed on your terminal when you print a message. This command is very handy for suppression of certain machine-generated header fields. The **Type** and **Print** commands can be used to print a message in its entirety, including ignored fields. If **ignore** is executed with no arguments, it lists the current set of ignored fields.

mailx (**m**) Takes as arguments login names and distribution group names and sends mail to those people.

mbox Indicates that a list of messages are to be sent to ***mbox*** in your home directory when you quit. This is the default action for messages if you do ***not*** have the ***hold*** option set.

next (**n** like + or CR) Goes to the next message in sequence and types it. With an argument list, types the next matching message.

preserve A synonym for **hold**.

print (**p**) Takes a message list and types out each message on the user's terminal.

quit (**q**) Terminates the session, saving all undeleted, unsaved messages in the user's ***mbox*** file in his login directory, preserving all messages marked with **hold** or **preserve** or never referenced in his system mailbox, and removing all other messages from his system mailbox. If new mail has arrived during the session, the message "You have new mail" is given. If given while editing a mailbox file with the **–f** flag, then the edit file is rewritten. A

return to the shell is effected, unless the rewrite of edit file fails, in which case the user can escape with the **exit** command.

reply (**r**) Takes a message list and sends mail to the sender and all recipients of the specified message. The default message must not be deleted.

respond A synonym for **reply**.

save (**s**) Takes a message list and a filename and appends each message in turn to the end of the file. The filename in quotes, followed by the line count and character count, is echoed on the user's terminal. Preceded by a vertical bar (|), the filename is treated as a pipe. For example,

```
s 2 |lpr
```

submits message #2 to the line printer.

set (**se**) With no arguments, prints all variable values. Otherwise, sets options. Arguments are of the form "option=value" or "option."

shell (**sh**) Invokes an interactive version of the shell.

size Takes a message list and prints out the size in characters of each message.

source (**so**) Reads *mailx* commands from a file.

top Takes a message list and prints the top few lines of each. The number of lines printed is controlled by the variable **toplines** and defaults to five.

type (**t**) A synonym for **print**.

unalias Takes a list of names defined by **alias** commands and discards the remembered groups of users. The group names no longer have any significance.

undelete (**u**) Takes a message list and marks each one as *not* being deleted.

unset Takes a list of option names and discards their remembered values; the inverse of **set**.

visual (**v**) Takes a message list and invokes the display editor on each message.

write (**w**) A synonym for **save**.

xit (**x**) A synonym for **exit**.

z *Mailx* presents message headers in windowfuls as described under the **headers** command. You can move *mailx*'s attention forward to the next window with the **z** command. Also, you can move to the previous window by using **z–**.

Here is a summary of the tilde escapes, which are used when composing messages to perform special functions. Tilde escapes are only recognized at the beginning of lines. The name "tilde escape" is somewhat of a misnomer since the actual escape character can be set by the option **escape**.

~!*command* Execute the indicated shell *command*, then return to the message.

~c *name ...* Add the given *name*s to the list of carbon copy recipients.

~d Read the file "dead.letter" from your home directory into the message.

~e Invoke the text editor on the message collected so far. After the editing session is finished, you may continue appending text to the message.

~f *messages* Read the named *messages* into the message being sent. If no messages are specified, read in the current message.

~h Edit the message header fields by typing each one in turn and allowing the user to append text to the end, or modify the field by using the current terminal erase and kill characters.

~**m** *messages* Read the named *messages* into the message being sent, shifted right one tab. If no messages are specified, read the current message.

~**p** Print out the message collected so far, prefaced by the message header fields.

~**q** Abort the message being sent, copying the message to "dead.letter" in your home directory if **save** is set.

~**r** *filename* Read the named file into the message.

~**s** *string* Cause the named *string* to become the current subject field.

~**t** *name ...* Add the given *names* to the direct recipient list.

~**v** Invoke an alternate editor (defined by the VISUAL option) on the message collected so far. Usually, the alternate editor is a screen editor. After you quit the editor, you may resume appending text to the end of your message.

~**w** *filename* Write the message onto the named file.

~| *command* Pipe the message through the *command* as a filter. If the command gives no output or terminates abnormally, retain the original text of the message. The command *fmt*(1) is often used as *command* to rejustify the message.

~~*string* Insert the string of text in the message prefaced by a single ~. If you have changed the escape character, then you should double that character in order to send it.

Options are controlled via the **set** and **unset** commands. Options may be either binary, in which case it is only significant to see whether they are set or not, or string, in which case the actual value is of interest. The binary options include the following:

append Causes messages saved in *mbox* to be appended to the end rather than inserted at the beginning.

ask Causes *mailx* to prompt you for the subject of each message you send. If you respond with simply a new-line, no subject field is sent.

askcc Causes you to be prompted for additional carbon copy recipients at the end of each message. Responding with a new-line indicates your satisfaction with the current list.

autoprint Causes the **delete** command to behave like **dp**. Thus, after deleting a message, the next one is typed automatically.

dot The binary option **dot** causes *mailx* to interpret a period alone on a line as the terminator of a message you are sending.

hold This option is used to hold messages in the system mailbox by default.

ignore Causes interrupt signals from your terminal to be ignored and echoed as @'s.

ignoreeof Causes *mailx* to refuse to accept a control–d as the end of a message. **Ignoreeof** also applies to *mailx* command mode.

metoo Usually, when a group is expanded that contains the sender, the sender is removed from the expansion. Setting this option causes the sender to be included in the group.

nosave Normally, when you abort a message with two interrupts, *mailx* copies the partial letter to the file "dead.letter" in your home directory. Setting this binary option prevents this.

quiet Suppresses the printing of the version when first invoked.

save Causes the message collected prior to an interrupt to be saved on the file "dead.letter" in your home directory on receipt of two interrupts (or after a ~**q**).

verbose Setting this option is the same as using the **–v** option on the comand line. When ***mailx*** runs in verbose mode, the actual delivery of messages is displayed on the user's terminal.

The following options have string values:

EDITOR Pathname of the text editor to use in the **edit** command and **~e** escape. If not defined, then a default editor is used.

SHELL Pathname of the shell to use in the **!** command and the **~!** escape. A default shell is used if this option is not defined.

VISUAL Pathname of the text editor to use in the **visual** command and **~v** escape.

crt The value of **crt** is used as a threshold to determine how long a message must be before *more*(1) is used to read it.

escape If defined, the first character of this option gives the character to use in the place of ~ to denote escapes.

folder The name of the directory to use for storing folders of messages. If the directory name begins with a slash (/), ***mailx*** considers it to be an absolute pathname. Otherwise, the directory is found relative to your home directory.

record If defined, gives the pathname of the file used to record all outgoing mail. If not defined, then outgoing mail is not saved in this way.

toplines If defined, gives the number of lines of a message to be printed out with the **top** command; normally, the first five lines are printed.

FILES

/usr/mail/*	post office
~/mbox	your old mail
~/.mailrc	file giving initial mail commands
/tmp/R#	temporary for editor escape
/usr/lib/mailx/mailx.help*	help files
/usr/lib/mailx/mailx.rc	system initialization file
Message*	temporary for editing messages

SEE ALSO

mail(1), fmt(1), uucp(1C).

Mailx in HP–UX Concepts and Tutorials.

BUGS

Many more flags are documented in the reference manual listed above. In general, these flags are not useful to most users.

NAME

make – maintain, update, recompile programs

SYNOPSIS

make [–**f** makefile] [–**p**] [–**i**] [–**k**] [–**s**] [–**r**] [–**n**] [–**b**] [–**e**] [–**t**] [–**q**] [–**d**] [names]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

The following is a brief description of all options and some special names. Options may occur in any order.

–**b** Compatibility mode for old (Version 7) makefiles.

–**d** Debug mode. Print out detailed information on files and times examined. (This is intended for debugging the *make* command itself.)

–**e** Environment variables override assignments within makefiles.

–**f** *makefile* Description file name. *Makefile* is assumed to be the name of a description file. A file name of – denotes the standard input. The contents of *makefile* override the built-in rules if they are present. Note that the space between –**f** and *makefile* **must** be present.

–**i** Ignore error codes returned by invoked commands. This mode is also entered if the fake target name **.IGNORE** appears in the description file.

–**k** Abandon work on the current entry, but continue on other branches that do not depend on that entry.

–**n** No execute mode. Print commands, but do not execute them. Even lines beginning with an @ are printed.

–**p** Print out the complete set of macro definitions and target descriptions.

–**q** Question. The *make* command returns a zero or non-zero status code depending on whether the target file is or is not up-to-date.

–**r** Do not use the built-in rules.

–**s** Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name **.SILENT** appears in the description file.

–**t** Touch the target files (causing them to be up-to-date) rather than issue the usual commands.

The "built-in" dependency targets are:

.DEFAULT

If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name **.DEFAULT** are used if it exists.

.PRECIOUS

Dependents of this target will not be removed when quit or interrupt are hit.

.SILENT

Same effect as the –**s** option.

.IGNORE

Same effect as the –**i** option.

Make executes commands in *makefile* to update one or more target *names*. *Name* is typically a program. If *makefile* is –, the standard input is taken. If no –**f** option is present, the current directory is searched for a file named **makefile**, **Makefile**, **s.makefile**, or **s.Makefile**, in that order. The search

stops when a file is found whose name matches one of these four names. (Note that *make* only performs one search for a *makefile*. If one is found, it is used and, if an error occurs, *make* terminates. If a *makefile* is not found, *make* terminates immediately.) More than one **–f** makefile argument pair may appear.

Make updates a target only if it depends on files that are newer than the target. All prerequisite files of a target are added recursively to the list of targets. Missing files are deemed to be out of date.

Makefile contains a sequence of entries that specify dependencies. The first line of an entry is a blank-separated, non-null list of targets, followed by a colon (:), followed by a (possibly null) list of prerequisite files or dependencies. Text following a **;** and all following lines that begin with a tab are shell commands to be executed to update the target. The first line that does not begin with a tab or # begins a new dependency or macro definition. Shell commands may be continued across lines with the <backslash><new-line> sequence. Sharp (#) and new-line surround comments.

The following *makefile* says that **pgm** depends on two files **a.o** and **b.o**, and that they in turn depend on their corresponding source files (**a.c** and **b.c**) and a common file **incl.h**:

```
pgm: a.o b.o
        cc a.o b.o –o pgm
a.o: incl.h a.c
        cc –c a.c
b.o: incl.h b.c
        cc –c b.c
```

Command lines are executed one at a time, each by its own shell. A line is printed when it is executed unless the **–s** option is present, or the entry **.SILENT:** is in *makefile*, or unless the first character of the command is @. The **–n** option specifies printing without execution; however, if the command line has the string **$(MAKE)** in it, the line is always executed (see discussion of the **MAKEFLAGS** macro under *Environment*). Note that this feature does not work if **MAKE** is enclosed in braces, as in **${MAKE}**. The **–t** (touch) option updates the modified date of a file without executing any commands.

Commands returning non-zero status normally terminate *make*. If the **–i** option is present, or the entry **.IGNORE:** appears in *makefile*, or if the line specifying the command begins with <tab><hyphen>, the error is ignored. If the **–k** option is present, work is abandoned on the current entry, but continues on other branches that do not depend on that entry.

The **–b** option allows old makefiles (those written for the old version of *make*) to run without errors. The difference between the old version of *make* and this version is that this version requires all dependency lines to have a (possibly null) command associated with them. The previous version of *make* assumed if no command was specified explicitly that the command was null.

Interrupt and quit cause the target to be deleted unless the target depends on the special name **.PRECIOUS**.

Environment

The environment is read by *make*. All variables are assumed to be macro definitions and processed as such. The environment variables are processed before any makefile and after the internal rules; thus, macro assignments in a makefile override environment variables. The **–e** option causes the environment to override the macro assignments in a makefile.

The **MAKEFLAGS** environment variable is processed by *make* as containing any legal input option (except **–f**, **–p**, and **–d**) defined for the command line. Further, upon invocation, *make* "invents" the variable if it is not in the environment, puts the current options into it, and passes it on to invocations of commands. Thus, **MAKEFLAGS** always contains the current input options. This proves very useful for "super-makes". In fact, as noted above, when the **–n** option is used, the command **$(MAKE)** is executed anyway; hence, one can perform a **make –n** recursively on a whole software system to see what would have been executed. This is because the **–n** is put in **MAKEFLAGS** and passed to further invocations of **$(MAKE)**. This is one way of debugging all of the makefiles for a software project without actually doing anything.

Macros

Entries of the form *string1* = *string2* are macro definitions. Subsequent appearances of $(*string1*[:*subst1* = [*subst2*]]) are replaced by *string2*. The parentheses are optional if a single character macro name is used and there is no substitute sequence. The optional :*subst1* = *subst2* is a substitute sequence. If it is specified, all non-overlapping occurrences of *subst1* in the named macro are replaced by *subst2*. Strings (for the purposes of this type of substitution) are delimited by blanks, tabs, new-line characters, and beginnings of lines. An example of the use of the substitute sequence is shown under *Libraries*.

Internal Macros

There are five internally maintained macros which are useful for writing rules for building targets.

$*** The macro **$* stands for the file name part of the current dependent with the suffix deleted. It is evaluated only for inference rules.

$@** The **$@ macro stands for the full target name of the current target. It is evaluated only for explicitly named dependencies.

$<** The **$< macro is only evaluated for inference rules or the **.DEFAULT** rule. It is the module which is out of date with respect to the target (i.e., the "manufactured" dependent file name). Thus, in the **.c.o** rule, the **$<** macro would evaluate to the **.c** file. An example for making optimized **.o** files from **.c** files is:

```
.c.o:
        cc –c –O $*.c
```

or:

```
.c.o:
        cc –c –O $<
```

$?** The **$? macro is evaluated when explicit rules from the makefile are evaluated. It is the list of prerequisites that are out of date with respect to the target; essentially, those modules which must be rebuilt.

$% The **$%** macro is only evaluated when the target is an archive library member of the form **lib(file.o)**. In this case, **$@** evaluates to **lib** and **$%** evaluates to the library member, **file.o**.

Four of the five macros can have alternative forms. When an upper case **D** or **F** is appended to any of the four macros the meaning is changed to "directory part" for **D** and "file part" for **F**. Thus, **$(@D)** refers to the directory part of the string **$@**. If there is no directory part, ./ is generated. The only macro excluded from this alternative form is **$?**. The reasons for this are debatable.

Suffixes

Certain names (for instance, those ending with **.o**) have inferable prerequisites such as **.c**, **.s**, etc. If no update commands for such a file appear in *makefile*, and if an inferable prerequisite exists, that prerequisite is compiled to make the target. In this case, *make* has inference rules which allow building files from other files by examining the suffixes and determining an appropriate inference rule to use. The current default inference rules are:

```
.c .c~ .sh .sh~ .c.o .c~.o .c~.c .s.o .s~.o .y.o .y~.o .l.o .l~.o
.y.c .y~.c .l.c .c.a .c~.a .s~.a .h~.h
```

To print out the rules compiled into the *make* on any machine in a form suitable for recompilation, the following command is used:

```
make –p –f – 2>/dev/null </dev/null
```

The only peculiarity in this output is the (**null**) string which *printf*(3S) prints when handed a null string.

A tilde in the above rules refers to an SCCS file (see *sccsfile*(5)). Thus, the rule **.c~.o** would transform an SCCS C source file into an object file (**.o**). Because the **s.** of the SCCS files is a prefix it is incompatible with *make*'s suffix point-of-view. Hence, the tilde is a way of changing any file reference into an SCCS

file reference.

A rule with only one suffix (i.e. **.c:**) is the definition of how to build *x* from *x*.**c**. In effect, the other suffix is null. This is useful for building targets from only one source file (e.g., shell procedures, simple C programs).

Additional suffixes are given as the dependency list for **.SUFFIXES**. Order is significant; the first possible name for which both a file and a rule exist is inferred as a prerequisite.

The default list is:

.SUFFIXES: .o .c .y .l .s

Here again, the above command for printing the internal rules will display the list of suffixes implemented on the current machine. Multiple suffix lists accumulate; **.SUFFIXES:** with no dependencies clears the list of suffixes.

Inference Rules

The first example can be done more briefly:

```
pgm: a.o b.o
        cc a.o b.o –o pgm
a.o b.o: incl.h
```

This is because *make* has a set of internal rules for building files. The user may add rules to this list by simply putting them in the *makefile*.

Certain macros are used by the default inference rules to permit the inclusion of optional matter in any resulting commands. For example, **CFLAGS**, **LFLAGS**, and **YFLAGS** are used for compiler options to *cc*(1), *lex*(1), and *yacc*(1) respectively. Again, the previous method for examining the current rules is recommended.

The inference of prerequisites can be controlled. The rule to create a file with suffix **.o** from a file with suffix **.c** is specified as an entry with **.c.o:** as the target and no dependents. Shell commands associated with the target define the rule for making a **.o** file from a **.c** file. Any target that has no slashes in it and starts with a dot is identified as a rule and not a true target.

Libraries

If a target or dependency name contains parenthesis, it is assumed to be an archive library, the string within parenthesis referring to a member within the library. Thus **lib(file.o)** and **$(LIB)(file.o)** both refer to an archive library which contains **file.o**. (This assumes the **LIB** macro has been previously defined.) The expression **$(LIB)(file1.o file2.o)** is not legal. Rules pertaining to archive libraries have the form *.XX*.**a** where the *XX* is the suffix from which the archive member is to be made. An unfortunate byproduct of the current implementation requires the *XX* to be different from the suffix of the archive member. Thus, one cannot have **lib(file.o)** depend upon **file.o** explicitly. The most common use of the archive interface follows. Here, we assume the source files are all C type source:

```
lib:    lib(file1.o) lib(file2.o) lib(file3.o)
        @echo lib is now up to date
.c.a:
        $(CC) –c $(CFLAGS) $<
        ar rv $@ $*.o
        rm -f $*.o
```

In fact, the **.c.a** rule listed above is built into *make* and is unnecessary in this example. A more interesting, but more limited example of an archive library maintenance construction follows:

```
lib:    lib(file1.o) lib(file2.o) lib(file3.o)
        $(CC) –c $(CFLAGS) $(?:.o=.c)
        ar rv lib $?
        rm $?  @echo lib is now up to date
```

```
.c.a:;
```

Here the substitution mode of the macro expansions is used. The **$?** list is defined to be the set of object file names (inside **lib**) whose C source files are out of date. The substitution mode translates the **.o** to **.c**. (Unfortunately, one cannot as yet transform to **.c~**; however, this may become possible in the future.) Note also, the disabling of the **.c.a:** rule, which would have created each object file, one by one. This particular construct speeds up archive library maintenance considerably. This type of construct becomes cumbersome if the archive library contains a mix of assembly programs and C programs.

FILES

[Mm]akefile
s.[Mm]akefile

SEE ALSO

sh(1).

WARNING

Be wary of any file (such as an include file) whose access, modification, and last change times cannot be altered by the *make*-ing process. For example, if a program depends on an include file which in turn depends on another include file, and if one or both of these files are out-of-date, *make* will try to update these files each time it is run, thus unnecessarily re-*make*ing up-to-date files dependent on the include file. The solution is to manually update these files with the *touch*(1) command before running *make*. (Note that it is generally a bad idea to include the *touch*(1) command in your makefile, because it can cause *make* to update a program that otherwise did not need to be updated.)

BUGS

Some commands return non-zero status inappropriately; use **–i** to overcome the difficulty.

Commands that are directly executed by the shell, notably *cd*(1), are ineffectual across new-lines in *make*.

The syntax **(lib(file1.o file2.o file3.o)** is illegal.

You cannot build **lib(file.o)** from **file.o**.

The macro **$(a:.o = .c~)** doesn't work.

There is a limit of 2500 characters, including the terminating new-line, for expanded dependency lines.

Make will not properly expand a macro within another macro when string substitution is involved.

NAME

man – on-line manual command

SYNOPSIS

man **–k** keyword ...
man **–f** file ...
man [–] [**–t**] [section] title ...

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: UCB

DESCRIPTION

Man is a program which gives information from the programmers manual. It can be asked to form one line descriptions of commands specified by name, or for all commands whose description contains any of a set of keywords. It can also provide on-line access to the sections of the printed manual.

When given the option **–k** and a set of keywords, *man* prints out a one line synopsis of each manual section whose listing in the table of contents contains that keyword.

When given the option **–f** and a list of file names, *man* attempts to locate manual sections related to those files, printing out the table of contents lines for those sections.

The **–t** option causes *man* to use *troff* instead of *nroff*. **–t** is ignored when specified with **–k** or **–f**.

When neither **–k** nor **–f** is specified, *man* formats a specified set of manual pages. If the section specifier *section* is given, *man* looks in that section of the manual for the given *titles*. *Section* can be a digit (0 – 9), or one of the words **local**, **new**, or **public**. If *section* is a digit, it may be followed by a single letter classifier (i.e. **1g** indicating a graphics program in section 1). Classifiers may not be specified if a word is given for *section*. If **local**, **new**, or **public** is specified, then the manual section **manl**, **mann**, or **manp** is searched, respectively.

If *section* is omitted, *man* searches the on-line manual sub-directories in the following order: **man1**, **mann**, **manl**, **man6**, **man8**, **man2**, **man3**, **man4**, **man5**, **man7**, and finally **manp**. *Man* prints the first *title* it finds, if any.

If no *section* value is specified, or if the first attempt fails, *man* appends default section classifiers onto the given *titles* in an effort to locate the file. The list below gives the default classifiers used for each manual section, in the order in which they are used:

manual section 1: none, **h**, **m**, **c**, **g**;
manual section 2: none, **h**, **j**, **v**;
manual section 3: none, **h**, **j**, **x**, **m**, **s**, **f**, **c**;
manual sections 4-8: none, **h**.

If the standard output is a teletype, or if the flag – is given, then *man* pipes its output through *rmnl*(1) to delete useless blank lines, *ul*(1) to create proper underlines for different terminals, and through *more*(1) to stop after each page. Hit a space to continue.

If the /usr/man/cat? directory is present and the file is not in it, but the file exists in /usr/man/man?, then the page is formatted and installed in /usr/man/cat? on first access. If only the /usr/man/cat? directories are present and/or nroff is not installed then only those pages which have been preformatted are displayable.

HARDWARE DEPENDENCIES

Series 200/500:

Troff is not currently supported.

FILES

/usr/man/man?/*
/usr/man/cat?/*

SEE ALSO

more(1), rmnl(1), ul(1), whereis(1), catman(8).

BUGS

The *section* words **local**, **new**, or **public** may *not* be abbreviated by **l**, **n**, or **p**. They must be completely spelled out.

NAME

mesg – permit or deny messages to terminal

SYNOPSIS

mesg [**n**] [**y**]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Mesg with argument **n** forbids messages via *write*(1) by revoking non-user write permission on the user's terminal. *Mesg* with argument **y** reinstates permission. All by itself, *mesg* reports the current state without changing it.

FILES

/dev/tty*

SEE ALSO

write(1).

DIAGNOSTICS

Exit status is 0 if messages are receivable, 1 if not, 2 on error.

NAME

mkdir – make a directory

SYNOPSIS

mkdir dirname ...

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

Mkdir creates specified directories in mode 777, masked by the current value of *umask*. Standard entries, ., for the directory itself, and .., for its parent, are made automatically.

Mkdir requires write permission in the parent directory.

SEE ALSO

rm(1), umask(1).

DIAGNOSTICS

Mkdir returns exit code 0 if all directories were successfully made; otherwise, it prints a diagnostic and returns non-zero.

NAME

mm – print documents formatted with MM macros

SYNOPSIS

mm [options] [files]

HP-UX COMPATIBILITY

Level: Text Processing - HP-UX/EXTENDED

Origin: System III

DESCRIPTION

Mm can be used to type out documents using *nroff*(1) and the MM text formatting macro package. It has options to specify preprocessing by *tbl*(1) and/or *neqn*(1) and postprocessing by various terminal-oriented output filters. The proper pipelines and the required arguments and flags for *nroff*(1) and MM are generated, depending on the options selected.

Options for *mm* are given below. Any other arguments or flags (e.g., **–rC3**) are passed to *nroff*(1) or to MM, as appropriate. Such options can occur in any order, but they must appear before the *files* arguments. If no arguments are given, *mm* prints a list of its options.

–T*term* Specifies the type of output terminal; for a list of recognized values for *term*, type **help term2**. If this option is *not* used, *mm* will use the value of the shell variable **$TERM** from the environment (see *profile*(5) and *environ*(7)) as the value of *term*, if **$TERM** is set; otherwise, *mm* will use **450** as the value of *term*. If several terminal types are specified, the last one takes precedence.

–12 Indicates that the document is to be produced in 12-pitch. May be used when **$TERM** is set to one of **300**, **300s**, **450**, and **1620**. (The pitch switch on the DASI 300 and 300s terminals must be manually set to **12** if this option is used.)

–c Causes *mm* to invoke *col*(1); note that *col*(1) is invoked automatically by *mm* unless *term* is one of **300**, **300s**, **450**, **37**, **4000A**, **382**, **4014**, **tek**, **1620**, and **X**.

–e Causes *mm* to invoke *neqn*(1).

–t Causes *mm* to invoke *tbl*(1).

–E Invokes the **–e** option of *nroff*(1).

–y Causes *mm* to use the non-compacted version of the macros (see *mm*(7)).

As an example (assuming that the shell variable **$TERM** is set in the environment to **450**), the two command lines below are equivalent:

```
mm –t –rC3 –12 ghh*
tbl ghh* | nroff –cm –T450 –12 –h –rC3
```

Mm reads the standard input when – is specified instead of any file names. (Mentioning other files together with – leads to disaster.) This option allows *mm* to be used as a filter, e.g.:

```
cat dws | mm –
```

The following helpful hints should aid you in using these macros:

1. *Mm* invokes *nroff*(1) with the **–h** flag. With this flag, *nroff*(1) assumes that the terminal has tabs set every 8 character positions.
2. Use the **–o***list* option of *nroff*(1) to specify ranges of pages to be output. Note, however, that *mm*, if invoked with one or more of the **–e**, **–t**, and – options, *together* with the **–o***list* option of *nroff*(1) may cause a harmless "broken pipe" diagnostic if the last page of the document is not specified in *list*.
3. If you use the **–s** option of *nroff*(1) (to stop between pages of output), use line-feed (rather than return or new-line) to restart the output. The **–s** option of *nroff*(1) does not work with the **–c** option of *mm*, or if *mm* automatically invokes *col*(1) (see **–c** option above).
4. If you lie to *mm* about the kind of terminal its output will be printed on, you'll get (often subtle) garbage; however, if you are redirecting output into a file, use the **–T37** option, and then use

the appropriate terminal filter when you actually print that file.

SEE ALSO

col(1), nroff(1), tbl(1), profile(5), mm(7), term(7).
MM– Memorandum Macros in *HP-UX Concepts and Tutorials.*

DIAGNOSTICS

"mm: no input file" if none of the arguments is a readable file and *mm* is not used as a filter.

NAME

more, page – file perusal filter for crt viewing

SYNOPSIS

more [**–cdflsu**] [–n] [+linenumber] [+/pattern] [name ...]

page [*more* options]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: UCB

DESCRIPTION

More is a filter which allows examination of continuous text, one screenful at a time, on a soft-copy terminal. It normally pauses after each screenful, printing **––More––** at the bottom of the screen. If you then type a carriage return, one more line is displayed. If you hit a space, another screenful is displayed. Other possibilites are enumerated later.

The command line options are:

–*n* An integer which is the size (in lines) of the window which *more* will use instead of the default.

–c *More* will draw each page by beginning at the top of the screen and erasing each line just before it draws on it. This avoids scrolling the screen, making it easier to read while *more* is writing. This option will be ignored if the terminal does not have the ability to clear to the end of a line.

–d *More* will prompt the user with the message "Hit space to continue, Rubout to abort" at the end of each screenful. This is useful if *more* is being used as a filter in some setting, such as a class, where many users may be unsophisticated.

–f This causes *more* to count logical lines, rather than screen lines. That is, long lines are not folded. This option is recommended if *nroff* output is being piped through *ul*, since the latter may generate escape sequences. These escape sequences contain characters which would ordinarily occupy screen positions, but which do not print when they are sent to the terminal as part of an escape sequence. Thus *more* may think that lines are longer than they actually are, and fold lines erroneously.

–l Do not treat ^L (form feed) specially. If this option is not given, *more* will pause after any line that contains a ^L, as if the end of a screenful had been reached. Also, if a file begins with a form feed, the screen will be cleared before the file is printed.

–s Squeeze multiple blank lines from the output, producing only one blank line. Especially helpful when viewing *nroff* output, this option maximizes the useful information present on the screen.

–u Normally, *more* will handle underlining and bold such as produced by *nroff* in a manner appropriate to the particular terminal: if the terminal can perform underlining or has a stand-out mode, *more* will output appropriate escape sequences to enable underlining, else stand-out mode, for underlined information in the source file. If the terminal can perform stand-out, *more* uses that mode for bold information. The **–u** option suppresses this processing, as do the "ul" and "os" termcap flags.

+*linenumber*
Start up at *linenumber*.

+*pattern*
Start up two lines before the line containing the regular expression *pattern*.

If the program is invoked as *page*, then the screen is cleared before each screenful is printed (but only if a full screenful is being printed), and $k - 1$ rather than $k - 2$ lines are printed in each screenful, where k is the number of lines the terminal can display.

More looks in the file */etc/termcap* to determine terminal characteristics, and to determine the default window size. On a terminal capable of displaying 24 lines, the default window size is 22 lines.

More looks in the environment variable **MORE** to pre-set any flags desired. For example, if you prefer to view files using the **–c** mode of operation, the shell command sequence

MORE = '–c'; export MORE

causes all invocations of *more*, including invocations by programs such as *man*, to use this mode. Normally, the user will place the command sequence which sets up the MORE environment variable in the *.profile* file.

If *more* is reading from a file, rather than a pipe, then a percentage is displayed along with the **––More––** prompt. This gives the fraction of the file (in characters, not lines) that has been read so far.

Other sequences which may be typed when *more* pauses, and their effects, are as follows (*i* is an optional integer argument, defaulting to 1):

i<space>
: display *i* more lines, (or another screenful if no argument is given).

i^D
: display 11 more lines (a "scroll"). If *i* is given, then the scroll size is set to *i* lines.

d
: same as ^D (control-D).

*i***z**
: same as typing a space except that *i*, if present, becomes the new window size.

*i***s**
: skip *i* lines and print a screenful of lines.

*i***f**
: skip *i* screenfuls and print a screenful of lines.

q or **Q**
: Exit from *more*.

=
: Display the current line number.

v
: Start up the editor *vi* at the current line.

h
: Help command; give a description of all the *more* commands.

i/*expr*
: search for the *i*-th occurrence of the regular expression *expr*. If there are less than *i* occurrences of *expr*, and the input is a file (rather than a pipe), then the position in the file remains unchanged. Otherwise, a screenful is displayed, starting two lines before the place where the expression was found. The user's erase and kill characters may be used to edit the regular expression. Erasing back past the first column cancels the search command.

*i***n**
: search for the *i*-th occurrence of the last regular expression entered.

'
: (single quote) Go to the point from which the last search started. If no search has been performed in the current file, this command goes back to the beginning of the file.

!*command*
: invoke a shell to execute *command*. The characters "%" and "!" in *command* are replaced with the current file name and the previous shell command respectively. If there is no current file name, "%" is not expanded. The sequences "\%" and "\!" are replaced by "%" and "!" respectively.

*i***:n**
: skip to the *i*-th next file given in the command line (skips to last file if *i* doesn't make sense).

*i***:p**
: skip to the *i*-th previous file given in the command line. If this command is given in the middle of printing out a file, then *more* goes back to the beginning of the file. If *i* doesn't make sense, *more* skips back to the first file. If *more* is not reading from a file, the bell is rung and nothing else happens.

:f
: display the current file name and line number.

:q or **:Q** exit from *more* (same as **q** or **Q**).

. (dot) repeat the previous command.

The commands take effect immediately, i.e., it is not necessary to type a carriage return. Up to the time when the command character itself is given, the user may hit the line kill character to cancel the numerical argument being formed. In addition, the user may hit the erase character to redisplay the **--More--(*xx*%)**.

At any time when output is being sent to the terminal, the user can hit the quit key (normally control-\). *More* stops sending output, and displays the usual **--More--** prompt. The user may then enter one of the above commands in the normal manner. Unfortunately, some output is lost when this is done, due to the fact that any characters waiting in the terminal's output queue are flushed when the quit signal occurs.

The terminal is set to **noecho** mode by this program so that the output can be continuous. What you type will thus not show on your terminal, except for the / and ! commands.

If the standard output is not a teletype, then *more* acts just like *cat*, except that a header is printed before each file (if there is more than one).

A sample usage of *more* in previewing *nroff* output would be

```
nroff -ms +2 doc.n | more -s
```

FILES

/etc/termcap	terminal data base
/usr/lib/more.help	help file

SEE ALSO

man(1), sh(1), termcap(5).

NAME

mount, umount – mount and unmount file system

SYNOPSIS

/etc/mount [special directory [**–r**]]

/etc/umount special

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Mount announces to the system that a removable file system is present on the device *special*. The *directory* must exist already; it becomes the name of the root of the newly mounted file system. *Directory* must be given as an absolute path name.

These commands maintain a table of mounted devices. If invoked with no arguments, *mount* prints the table.

The optional last argument indicates that the file is to be mounted read-only. Physically write-protected and magnetic tape file systems must be mounted in this way or errors will occur when access times are updated, whether or not any explicit write is attempted.

Umount announces to the system that the removable file system previously mounted on device *special* is to be removed.

HARDWARE DEPENDENCIES

Series 500:

Warning: if virtual memory is brought up on a volume other than the root volume, and if that volume is then mounted, it cannot be unmounted.

FILES

/etc/mnttab mount table

SEE ALSO

mount(2), mnttab(5).

DIAGNOSTICS

Attempts to mount a currently-mounted volume under another name will result in an error [EBUSY].

If an attempt to read and (partially) verify the disc label information fails, the mount will fail.

Umount complains if the special file is not mounted or if it is busy. The file system is busy if it contains an open file or some user's working directory.

BUGS

Some degree of validation is done on the file system, however it is generally unwise to mount garbage file systems.
The third parameter may be anything which has the effect of **–r**.
An error will occur if *mnttab* does not exist.
Names are truncated to MNTLEN bytes (see *mnttab*(5)).

NAME

mt – magnetic tape manipulating program

SYNOPSIS

mt [**–t** tapename] command [count]

HP-UX COMPATABILITY

Level: Magnetic Tape Support – HP-UX/STANDARD

Origin: UCB

DESCRIPTION

Mt is used to give commands to the tape drive. If *tapename* is not specified, /dev/rmt12 is used. If *count* is not specified, 1 is assumed.

Here are the commands:

eof	write *count* end-of-file marks
fsf	space forward *count* files
fsr	space forward *count* records
bsf	space backward *count* files
bsr	space backward *count* records
rew	rewind tape
offl	rewind tape and go offline.

FILES

/dev/mt*	Magnetic tape interface
/dev/rmt*	Raw magnetic tape interface

/dev/rmt12 (or whatever drive is used) must be described as a Berkeley compatibility-mode tape drive without rewind for *mt* to operate as expected.

SEE ALSO

mt(4).

NAME

mvdir – move a directory

SYNOPSIS

/**etc**/**mvdir** dirname name

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

Mvdir moves and/or renames directories within a file system. *Dirname* must be a directory. If *name* exists, then the directory *dirname* is moved. If *name* does not exist, *dirname* is simply renamed.

Name cannot be a subdirectory of *dirname*. *Dirname* may be a subdirectory of *name*, but the shorthand notations **.** and **..** must be used in naming the directories, because *mvdir* does not allow explicit directory names to be used when one directory is a subdirectory of the other.

Only the super-user can use *mvdir*.

SEE ALSO

mkdir(1).

BUGS

The restriction on names is intended to prevent creation of a (cyclic) sub-tree that cannot be reached from the root. The test is strictly by name, thus creating such a sub-tree is still possible. The super-user is cautioned to be very careful in his use of the names **.** and **..** while moving directories.

NAME

ncheck – generate names from i-numbers

SYNOPSIS

/etc/ncheck [**–i** numbers] [**–a**] [**–s**] [file-system]

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

Remarks: *Ncheck* is implemented on the Series 200 only.

DESCRIPTION

Ncheck with no argument generates a path name vs. i-number list of all files on the volumes specified by the file **/etc/checklist.** Names of directory files are followed by /.. The options are as follows:

–i reduces the report to only those files whose i-numbers are specified on the command line in the *numbers* list.

–a allows printing of the names **.** and **..**, which are ordinarily suppressed.

–s reduces the report to special files and files with set-user-ID mode; it is intended to discover concealed violations of security policy.

A file system may be specified.

The report is in no useful order, and probably should be sorted.

SEE ALSO

sort(1), checklist(5), fsck(8).

DIAGNOSTICS

When the file system structure is improper, **??** denotes the "parent" of a parentless file and a path name beginning with **...** denotes a loop.

NAME

newgrp – log in to a new group

SYNOPSIS

newgrp [group]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Newgrp changes the group identification of its caller. The same user remains logged in, and the current directory is unchanged, but calculations of access permissions to files are performed with respect to the new group ID.

Newgrp without an argument changes the group identification to the group in the password file; in effect it changes the group identification back to the caller's original group.

A password is demanded if the group has a password and the user himself does not, or if the group has a password and the user is not listed in **/etc/group** as being a member of that group.

Newgrp is recognized by the shell, and causes the *newgrp* program to execute on top of the shell (instead of as the usual sub-process). If *newgrp* fails, you cannot be returned to your old shell, and you are thus logged out.

FILES

/etc/group
/etc/passwd

SEE ALSO

login(1), group(5).

DIAGNOSTICS

Sorry:	You didn't qualify as a group member.
Unknown group:	The group name was not in /etc/group.
Permission denied:	If a password must be given, it can only come from a teletype port. If the *stdin* is a non-tty file, this message is given.
You have no shell:	*Exec* of the shell failed.

BUGS

There is no convenient way to enter a password into **/etc/group**.
Use of group passwords is not encouraged, because, by their very nature, they encourage poor security practices. Group passwords may disappear in the future.
Shell variables which are not exported are lost.

NAME

news – print news items

SYNOPSIS

news [**–a**] [**–n**] [**–s**] [items]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

News is used to keep the user informed of current events. By convention, these events are described by files in the directory /**usr**/**news**.

When invoked without arguments, *news* prints the contents of all current files in /**usr**/**news**, most recent first, with each preceded by an appropriate header. *News* stores the "currency" time as the modification date of a file named **.news_time** in the user's home directory (the identity of this directory is determined by the environment variable **$HOME**); only files more recent than this currency time are considered "current."

The **–a** option causes *news* to print all items, regardless of currency. In this case, the stored time is not changed.

The **–n** option causes *news* to report the names of the current items without printing their contents, and without changing the stored time.

The **–s** option causes *news* to report how many current items exist, without printing their names or contents, and without changing the stored time. It is useful to include such an invocation of *news* in one's **.profile** file, or in the system's /**etc**/**profile**.

All other arguments are assumed to be specific news items that are to be printed.

If an interrupt is typed during the printing of a news item, printing stops and the next item is started. Another *delete* within one second of the first causes the program to terminate.

FILES

/etc/profile
/usr/news/*
$HOME/.news_time

SEE ALSO

mail(1), profile(5), environ(7).

NAME

nice – run a command at low priority

SYNOPSIS

nice [–increment] command [arguments]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Nice executes *command* with a lower CPU scheduling priority. If the *increment* argument (in the range 1-19) is given, it is used; if not, an increment of 10 is assumed.

The super-user may run commands with priority higher than normal by using a negative increment, e.g., **––10**.

HARDWARE DEPENDENCIES

Series 500:

A note to the super-user: be careful about increasing the priority of your processes. Your keyboard process is running at a nice value of 1, 2, 3, or 4. If you should assign a process a nice value of 0, you will lock out your keyboard, forcing you to reboot the system.

SEE ALSO

nohup(1), nice(2).

DIAGNOSTICS

Nice returns the exit status of the subject command.

BUGS

An *increment* larger than 19 is equivalent to 19.

NAME

nl – line numbering filter

SYNOPSIS

nl [–**h**type] [–**b**type] [–**f**type] [–**v**start#] [–**i**incr] [–**p**] [–**l**num] [–**s**sep] [–**w**width] [–**n**format] [–**d**delim] file

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Nl reads lines from the named *file* or the standard input if no *file* is named and reproduces the lines on the standard output. Lines are numbered on the left in accordance with the command options in effect.

Nl views the text it reads in terms of logical pages. Line numbering is reset at the start of each logical page. A logical page consists of a header, a body, and a footer section. Empty sections are valid. Different line numbering options are independently available for header, body, and footer (e.g., no numbering of header and footer lines while numbering blank lines only in the body).

The start of logical page sections are signaled by input lines containing nothing but the following delimiter character(s):

Line contents	*Start of*
\:\:\:	header
\:\:	body
\:	footer

Unless optioned otherwise, *nl* assumes the text being read is in a single logical page body.

Command options may appear in any order and may be intermingled with an optional file name. Only one file may be named. The options are:

–b*type* Specifies which logical page body lines are to be numbered. Recognized *types* and their meaning are:

a	number all lines;
t	number lines with printable text only;
n	no line numbering;
p*string*	number only lines that contain the regular expression specified in *string*.

The default *type* for logical page body is **t** (text lines numbered).

–h*type* Same as **–b***type* except for header. Default *type* for logical page header is **n** (no lines numbered).

–f*type* Same as **–b***type* except for footer. Default for logical page footer is **n** (no lines numbered).

–p Do not restart numbering at logical page delimiters.

–v*start#* *Start#* is the initial value used to number logical page lines. Default is **1**.

–i*incr* *Incr* is the increment value used to number logical page lines. Default is **1**.

–s*sep* *Sep* is the character(s) used in separating the line number and the corresponding text line. Default *sep* is a tab.

–w*width* *Width* is the number of characters to be used for the line number. Default *width* is **6**.

–n*format* *Format* is the line numbering format. Recognized values are:

ln left justified, leading zeroes suppressed;
rn right justified, leading zeroes supressed;
rz right justified, leading zeroes kept.

Default *format* is **rn** (right justified).

–l*num* *Num* is the number of blank lines to be considered as one. For example, **–l2** results in only the second adjacent blank being numbered (if the appropriate **–ha**, **–ba**, and/or **–fa** option is set). Default is **1**.

–d*xx* The delimiter characters specifying the start of a logical page section may be changed from the default characters (\:) to two user-specified characters. If only one character is entered, the second character remains the default character (:). No space should appear between the **–d** and the delimiter characters. To enter a backslash, use two backslashes.

EXAMPLE

The command:

```
nl –v10 –i10 –d!+ file1
```

will number file1 starting at line number 10 with an increment of ten. The logical page delimiters are ! and +.

SEE ALSO

pr(1).

NAME

nm – print name list (symbol table) of object file

SYNOPSIS

nm [–**gnoprsu**] [*filename* ...]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

Remarks: This manual page describes *nm* as implemented on the Series 200 computers. Refer to other *nm*(1) manual pages for information valid for other implementations.

DESCRIPTION

Nm prints the name list (symbol table) of each object file in the argument list. If an argument is an archive, a listing for each object file in the archive will be produced. If no *file* is given, the symbols in **a.out** are listed.

Each symbol name is preceded by its value printed in a representation appropriate to the architecture of the machine (blanks if undefined) and one of the letters **U** (undefined), **A** (absolute), **T** (text segment symbol), **D** (data segment symbol), **B** (bss segment symbol), **R** (register symbol), **F** (file symbol), or **C** (common symbol). If the symbol is local (non-external) the type letter is in lower case. The output is sorted alphabetically.

Options are:

–**g** Print only global (external) symbols.

–**n** Sort numerically rather than alphabetically.

–**o** Prepend file or archive element name to each output line rather than only once. This option can be used to make piping to *grep*(1) more meaningful.

–**p** Don't sort; print in symbol-table order.

–**r** Sort in reverse order.

–**s** Sort according to the size of the external symbol (computed from the difference between the value of the symbol and the value of the symbol with the next highest value). This difference is the value printed. This flag turns on –**g** and –**n** and turns off –**u** and –**p**.

–**u** Print only undefined symbols.

If the symbol was an align symbol, the letter **L** will be printed after the letter describing its type.

SEE ALSO

ar(1), a.out(5), ar(5).

NAME

nm – print name list (symbol table) of object file

SYNOPSIS

nm [–**gnopru**] [file ...]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

Remarks: This manual page describes *nm* as implemented on the Series 500 computers. Refer to other *nm* manual pages for information valid for other implementations.

DESCRIPTION

Nm prints the name list (symbol table) of each object *file* in the argument list. If an argument is an archive, a listing for each object file in the archive will be produced, preceded by the member name on a separate line. If no *file* is given, the symbols in **a.out** are listed.

Options are:

–**g** Print only global (external) symbols.

–**n** Sort numerically rather than alphabetically.

–**o** Prepend file or archive element name to each output line rather than only once. This option can be used to make piping to *grep*(1) more meaningful.

–**p** Don't sort; print in symbol-table order.

–**r** Sort in reverse order.

–**u** Print only undefined symbols.

The output from *nm* consists of five columns of data. The following is a portion of a typical output:

```
.  X  IDATA      00000108                 A_iob
.  X  IDATA      000002a0                 A_sctab
.  X  ICOMM      00000400 0 00000440      A_sibuf
.  X  ICOMM      00000400 0 00000840      A_sobuf
.  .  UDATA      00000c40                 Aallocs
.  X  FUNC       EDS c04 002a8 00000003   __cleanup
.  X  DDATA      DR 00000098              __ctype
.  X  FUNC       EDS c0c 00000 00000001   __doscan
.  X  SYSTEM     EPP 004 0000e            __exit
.  X  DDATA      DR 00000038              __iob
.  X  DCOMM      00000004 000000b0        __pfile
.  X  DDATA      DR 00000090              __sctab
.  X  PTR        1 00000a 000000b4        __sibuf
.  X  PTR        1 00000c 000000b8        __sobuf
.  .  FILENAME   0000000a                 _exit.o
.  .  FILENAME   0000000f                 _print.o
```

From left to right, the first column specifies whether the symbol is defined (.) or undefined (U). The second column specifies whether the symbol is non-external (.) or external (X). The third column gives the linker symbol type (as defined in *a.out.h* and described below). The fourth column lists the data associated with the specified symbol type. The fifth column gives the name of the system call, file, variable, array, common, etc., described by that entry in the symbol table.

Up to four data elements are reported in the fourth column. If they are not symbolic values (such as 'EDS' or 'DR'), then they are hexadecimal values. The number of data elements reported depends on the symbol type. Each symbol type has one to four parameters associated with it, whose values are given by the data elements in the fourth column. The symbol types and associated parameters are discussed below.

The following symbol types are supported:

ABS not currently generated; reserved for future use.

FUNC or **ENTRY** specifies that the entry refers to a function or procedure call. Four numbers, *ptr_type*, *segment*, *offset*, and *stt_index*, are associated. Their values are given in order, from left to right, by the data elements. *Ptr_type* consists of a single bit that is always cleared. It is symbolically represented by 'EDS'. *Ptr_type* is meaningful to the linker (see *ld*(1)), and specifies the storage format of the call in the symbol table. *Segment* specifies the code segment number (a number in the range 3073 to 4095, that indicates which code segment in the user's program space contains the desired code). *Offset* specifies the number of bytes from the beginning of the code segment where the function or procedure code begins. *Stt_index* is an indirect reference to the beginning of the function or procedure code.

SYSTEM specifies that the entry refers to a procedure call directly into the system kernel. Three numbers, *entry_type*, *segment*, and *stt*, are associated. Their values are given by the data elements. *Entry_type* consists of a single bit that is always set. Its value is symbolically represented by 'EPP'. *Entry_type* is meaningful to the linker, and specifies the storage format of the call in the symbol table. *Segment* specifies the system code segment number (the number of a code segment among those set aside for system use; typically in the range 0 to 64). *Stt* is an indirect pointer to the beginning of the procedure code.

LABEL specifies that the entry is the destination address for a branch instruction. Three numbers, *ptr_type*, *segment*, and *offset*, are associated. Their values are given by the data elements. *Ptr_type* consists of a single bit which is always cleared. Its value is symbolically represented by 'EDS'. *Ptr_type* is meaningful to the linker, and specifies the storage format of the address in the symbol table. *Segment* specifies the user code segment number. *Offset* specifies the number of bytes from the beginning of the code segment where the label begins.

DDATA specifies that the entry is a directly-addressable, initialized data structure (a variable, or the beginning of an array, common, structure, etc.). Two numbers, *base_reg* and *displacement*, are associated. Their values are given by the data elements. *Base_reg* is assigned one of nine possible symbolic values which describe the addressing scheme used to find the data structure. It is meaningful to the linker. The possible symbolic values are P+, P–, DB, DL, Q+, Q–, SB, S–, and DR. *Displacement* specifies the byte offset where the data structure is located. Note that this offset is measured relative to the beginning of the data space of the file for which the *nm* listing is made. The actual byte offset of the data structure in the executable **a.out** file could change.

IDATA or **UDATA** specifies that the entry refers to an indirectly-addressable, uninitialized array, or an indirectly-addressable, initialized common block. One number, *displacement*, is associated. Its value is given by the data element. It is identical to the *displacement* described above under **DDATA**.

DCOMM or **ICOMM** specifies that the entry is treated as a common block. Three numbers, *blocksize*, *needs_length_word, and displacement*, are associated. Their values are given by the data elements. *Blocksize* is the size, in bytes, of the common block.

Needs_length_word is a boolean value which appears in a print-out as either 0 or 1. If its value is 1, the linker places the value of (*blocksize* – 4) in the first four bytes of the common block. This information is necessary when linking FORTRAN programs. *Displacement* is identical to that described under **DDATA** above.

PTR specifies that the entry is a pointer to an indirectly-addressable data structure (variable, array, common block, etc.). Three numbers, *ptr_to_common*, *target*, and *address*, are associated. Their values are given by the data elements. *Ptr_to_common* is an eight-bit boolean expression. Its value is given as 1 (true) or 0 (false). If true, then the entry is a pointer to a common block. If false, the entry is a pointer to some other type of data structure. *Target* is an index into the symbol table to the entry that describes the target of the data structure pointer. *Address* is a pointer to the data structure pointer; that is, an indirect pointer to the data structure.

SEGMENT not currently generated; reserved for future use.

FILENAME specifies that the entry is a file name. One number, *sequence*, is associated. Its value is given by the data element. *Sequence* reflects the order in which the linker encountered each file name.

SEE ALSO

ar(1), a.out(5), ar(5).

DIAGNOSTICS

Nm generates an error message for the following conditions:

invalid option
cannot open *file*
bad magic number
read error

NAME

nohup – run a command immune to hangups, logouts, and quits

SYNOPSIS

nohup command [arguments]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Nohup executes *command* with hangups and quits ignored. If output is not re-directed by the user, it will be sent to **nohup.out**. If **nohup.out** is not writable in the current directory, output is redirected to **$HOME/nohup.out**; otherwise, *nohup* will fail.

If output from *nohup* is redirected to a terminal, or is not redirected at all, the output is sent to **nohup.out**.

SEE ALSO

nice(1), signal(2).

NAME

nroff – format text

SYNOPSIS

nroff [options] [files]

HP-UX COMPATIBILITY

Level: *Nroff* - HP-UX/STANDARD

Origin: System III

DESCRIPTION

Nroff formats text contained in *files* (standard input by default) for printing on typewriter-like devices and line printers. Its capabilities are described in the *NROFF/TROFF User's Manual* cited below.

An argument consisting of a minus (–) is taken to be a file name corresponding to the standard input. The *options*, which may appear in any order, but must appear before the *files*, are:

–o*list* Print only pages whose page numbers appear in the *list* of numbers and ranges, separated by commas. A range *N*–*M* means pages *N* through *M*; an initial –*N* means from the beginning to page *N*; and a final *N*– means from *N* to the end. (See *BUGS* below.)

–n*N* Number first generated page *N*.

–s*N* Stop every *N* pages. *Nroff* will halt *after* every *N* pages (default $N = 1$) to allow paper loading or changing, and will resume upon receipt of a line-feed or new-line (new-lines do not work in pipelines, e.g., with *mm*(1)). When *nroff* halts between pages, an ASCII **BEL** is sent to the terminal.

–r*aN* Set register *a* (which must have a one-character name) to *N*.

–**i** Read standard input after *files* are exhausted.

–**q** Invoke the simultaneous input-output mode of the **.rd** request.

–**z** Print only messages generated by **.tm** (terminal message) requests.

–m*name* Precede the input *files* with the non-compacted (ASCII text) macro file **/usr/lib/tmac/tmac.***name*.

–c*name* Precede the input *files* with the compacted macro files **/usr/lib/macros/cmp.[nt].[dt].***name* and **/usr/lib/macros/ucmp.[nt].***na*

–k*name* Compact the macros used in this invocation of *nroff*, placing the output in files **[dt].***name* in the current directory (see the May 1979 Addendum to the *NROFF/TROFF User's Manual* for details of compacting macro files).

–T*name* Prepare output for specified terminal. Known *name*s are **37** for the (default) TELETYPE® Model 37 terminal, **tn300** for the GE TermiNet 300 (or any terminal without half-line capability), **300s** for the DASI 300s, **300** for the DASI 300, **450** for the DASI 450, **lp** for a (generic) ASCII line printer, **382** for the DTC-382, **4000A** for the Trendata 4000A, **832** for the Anderson Jacobson 832, **X** for a (generic) EBCDIC printer, and **2631** for the Hewlett Packard 2631 line printer.

–**e** Produce equally-spaced words in adjusted lines, using the full resolution of the particular terminal.

–**h** Use output tabs during horizontal spacing to speed output and reduce output character count. Tab settings are assumed to be every 8 nominal character widths.

–u*n* Set the emboldening factor (number of character overstrikes) for the third font position (bold) to *n*, or to zero if *n* is missing.

HARDWARE DEPENDENCIES

Series 500:

The **–c** and **–k** options are not currently supported.

FILES

/usr/lib/suftab suffix hyphenation tables

/tmp/ta$#	temporary file
/usr/lib/tmac/tmac.*	standard macro files and pointers
/usr/lib/macros/*	standard macro files
/usr/lib/term/*	terminal driving tables for *nroff*

SEE ALSO

mm(1).

NROFF/TROFF User's Manual in *HP-UX Concepts and Tutorials.*

BUGS

Nroff is keyed to Eastern Standard Time; as a result, depending on the time of the year and on your local time zone, the date that *nroff* generates may be off by one day from your current date.

When *nroff* is used with the *–olist* option inside a pipeline, it may cause a harmless "broken pipe" diagnostic if the last page of the document is not specified in *list*.

NAME

od, xd – octal and hexadecimal dump

SYNOPSIS

od [**–bcdox**] [file] [[+]offset[**.**][**b**]]
xd [*od* options]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III and HP

DESCRIPTION

Od (*xd*) dumps *file* in one or more formats as selected by the first argument. If the first argument is missing, **–o** (**–x**) is the default. Each line begins with an offset field. For *od*, the offset is in octal, and for *xd* the offset is in hexadecimal. The meanings of the format options are:

–b Interpret bytes in octal (hexadecimal).

–c Interpret bytes in ASCII. Certain non-graphic characters appear as C escapes: null = **0**, backspace = **b**, form-feed = **f**, new-line = **n**, return = **r**, tab = **t**; others appear as 3-digit octal numbers.

–d Interpret 16-bit words in decimal.

–o Interpret 16-bit words in octal.

–x Interpret 16-bit words in hexadecimal.

The *file* argument specifies which file is to be dumped. If no *file* argument is specified, the standard input is used.

The *offset* argument specifies the offset in the file where dumping is to commence. This argument is normally interpreted as octal bytes. If **.** is appended, *offset* is interpreted in decimal. If *offset* begins with **0x**, *offset* is interpreted in hexadecimal. If **b** is appended, *offset* is interpreted in blocks of 512 bytes. If the file argument is omitted, *offset* must be preceded by +.

Dumping continues until end-of-file.

SEE ALSO

adb(1).

NAME

pack, pcat, unpack – compress and expand files

SYNOPSIS

pack [–] [**–f**] name ...

pcat name ...

unpack name ...

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Pack attempts to store the specified files in a compressed form. Wherever possible (and useful), each input file *name* is replaced by a packed file *name*.**z** with the same access modes, access and modified dates, and owner as those of *name*. The **-f** option forces packing of *name*. This is useful for causing an entire directory to be packed even if some of the files do not benefit. If *pack* is successful, *name* is removed. Packed files can be restored to their original form using *unpack* or *pcat*.

Pack uses Huffman (minimum redundancy) codes on a byte-by-byte basis. If the – argument is used, an internal flag is set that causes the number of times each byte is used, its relative frequency, and the code for the byte to be printed on the standard output. Additional occurrences of – in place of *name* will cause the internal flag to be set and reset.

The amount of compression obtained depends on the size of the input file and the character frequency distribution. Because a decoding tree forms the first part of each **.z** file, it is usually not worthwhile to pack files smaller than three blocks, unless the character frequency distribution is very skewed, which may occur with printer plots or pictures.

Typically, text files are reduced to 60-75% of their original size. Load modules, which use a larger character set and have a more uniform distribution of characters, show little compression, the packed versions being about 90% of the original size.

Pack returns a value that is the number of files that it failed to compress.

No packing will occur if:

the file appears to be already packed;
the file name has more than 12 characters;
the file has links;
the file is a directory;
the file cannot be opened;
no disk storage blocks will be saved by packing;
a file called *name*.**z** already exists;
the **.z** file cannot be created;
an I/O error occurred during processing.

The last segment of the file name must contain no more than 12 characters to allow space for the appended **.z** extension. Directories cannot be compressed.

Pcat does for packed files what *cat*(1) does for ordinary files, except that *pcat* cannot be used as a filter. The specified files are unpacked and written to the standard output. Thus to view a packed file named *name*.**z** use:

```
pcat name.z
```

or just:

```
pcat name
```

To make an unpacked copy, say *nnn*, of a packed file named ***name.z*** (without destroying ***name.z***) *use the command:*

pcat name >nnn

Pcat returns the number of files it was unable to unpack. Failure may occur if:

the file name (exclusive of the **.z**) has more than 12 characters;
the file cannot be opened;
the file does not appear to be the output of *pack*.

Unpack expands files created by *pack*. For each file *name* specified in the command, a search is made for a file called ***name.z*** (or just *name*, if *name* ends in **.z**). If this file appears to be a packed file, it is replaced by its expanded version. The new file has the **.z** suffix stripped from its name, and has the same access modes, access and modification dates, and owner as those of the packed file.

Unpack returns a value that is the number of files it was unable to unpack. Failure may occur for the same reasons that it may in *pcat*, as well as for the following:

a file with the "unpacked" name already exists;
the unpacked file cannot be created.

SEE ALSO

cat(1).

NAME

passwd – change login password

SYNOPSIS

passwd [name]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

This command changes (or installs) a password associated with the login *name*. If *name* is omitted, it defaults to *getlogin*(3) name.

The program prompts for the old password (if any) and then for the new one (twice). The caller must supply these. New passwords should be at least four characters long if they use a sufficiently rich alphabet and at least six characters long if monocase. Only the first eight characters of the password are significant.

Only the owner of the name or the super-user may change a password; the owner must prove he knows the old password, if any. The super-user need not supply the old password when he changes another user's password. Only the super-user can create a null password.

The password file is not changed if the new password is the same as the old password, or if the password has not "aged" sufficiently; see *passwd*(5)).

FILES

/etc/passwd

SEE ALSO

login(1), crypt(3C), passwd(5).

DIAGNOSTICS

Permission denied.
name is not in password file, or you are not user *name* or the super-user.

Sorry. the old password does not match.

Sorry: <x weeks since the last change
password aging is in effect, and it is too soon to change yours.

You may not change this password
the super-user has made it impossible to change your password.

Too short
passwords must be at least 4 characters long.

Please use at least one non-numeric character
your new password does not utilize a sufficiently varied selection of characters. You can override this rule by re-entering your new password 2 more times.

Please use a longer password.
your new password is not long enough to be sufficiently secure. You can override this rule by re-entering your new password 2 more times.

They don't match, try again.
the two entries of your new password are not identical.

Temporary file busy; try again later
only one user can change his password at a time.

Cannot create temporary file
see the super-user.

Cannot unlink 'filename'
see the super-user.

cannot link 'file' to 'file'.
see the super-user.

cannot recover 'file'.
see the super-user.

Password unchanged.
the new and old passwords are the same.

NAME

paste – merge lines in one or more files

SYNOPSIS

paste file1 file2 . . .
paste –d list file1 file2 . . .
paste –s [**–d** list] file1 file2 . . .

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

In the first two forms, *paste* concatenates corresponding lines of the given input files *file1*, *file2*, etc. It treats each file as a column or columns of a table and pastes them together horizontally (parallel merging). If you will, it is the counterpart of *cat*(1) which concatenates vertically, i.e., one file after the other. In the last form above, *paste* subsumes the function of an older command with the same name by combining subsequent lines of the input file (serial merging). In all cases, lines are glued together with the *tab* character, or with characters from an optionally specified *list*. Output is to the standard output, so it can be used as the start of a pipe, or as a filter, if – is used in place of a file name.

The meanings of the options are:

–d Without this option, the new-line characters of each but the last file (or last line in case of the **–s** option) are replaced by a *tab* character. This option allows replacing the *tab* character by one or more alternate characters (see below).

list One or more characters immediately following **–d** replace the default *tab* as the line concatenation character. The list is used circularly, i. e. when exhausted, it is reused. In parallel merging (i. e. no **–s** option), the lines from the last file are always terminated with a new-line character, not from the *list*. The list may contain the special escape sequences: **n** (new-line), **t** (tab), \\ (backslash), and **0** (empty string, not a null character). Quoting may be necessary, if characters have special meaning to the shell (e.g. to get one backslash, use " " **–d** " \ \ \ \ ").

–s Merge subsequent lines rather than one from each input file. Use *tab* for concatenation, unless a *list* is specified with **–d** option. Regardless of the *list*, the very last character of the file is forced to be a new-line.

– May be used in place of any file name, to read a line from the standard input. (There is no prompting).

EXAMPLES

ls | paste –d" " –	list directory in one column
ls | paste – – – –	list directory in four columns
paste –s –d"\t\n" file	combine pairs of lines into lines

SEE ALSO

grep(1), cut(1),
pr(1): **pr –t –m**. . . works similarly, but creates extra blanks, tabs and new-lines for a nice page layout.

DIAGNOSTICS

line too long	Output lines are restricted to 511 characters.
too many files	Except for **–s** option, no more than 12 input files may be specified.

NAME

pc – Pascal compiler

SYNOPSIS

pc [options] files

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: HP

Remarks: This manual page describes the Pascal compiler as it is implemented on the Series 200 computer. Refer to other *pc*(1) manual pages for information valid for other implementations.

DESCRIPTION

Pc is the HP standard Pascal compiler. It accepts two types of file arguments:

(1) Arguments whose names end with **.p** are taken to be Pascal source programs. They are compiled, and each object program is left in the current directory in a file whose name is that of the source, with **.a** substituted for **.p**. (The **.a** file will not appear for a single source which is compiled and loaded, nor for any source which fails to compile correctly.)

(2) Arguments whose names end with **.a** are passed on to the linker (*ld*(1)) to be linked into the final program.

The following options are recognized:

–c Suppress linking and produce object (**.a**) files from source files;

–w Suppress warning messages (same as **$WARN OFF$**);

–L Write a program listing to *stdout* or to the file given in *$LIST filename $option* in source during compilation;

–o outfile Name the output file from the linker *outfile* instead of *a.out*;

–e Write lines containing errors to *stderr*;

–v Write expanded compiler and linker runstrings to *stderr*.

Other options are taken to be arguments to *ld*, and are passed along to the linker.

The compiler generates object code in archive file format, putting each module in a separate **.o** format file and archiving them into a **.a** file.

FILES

file.p	input file (Pascal source file)
file.a	archive file of object file(s)
a.out	linked executable output file
/bin/pc	mother program
/usr/lib/pascomp	compiler
/lib/crt0.o	runtime startoff
/lib/libpc.a	Pascal run-time library
/usr/tmp/*	temporary files used by the compiler
/usr/lib/paserrs	compiler errors
/user/lib/escerrs	Pascal escape codes
/usr/lib/syserrs	unix system messages
/usr/lib/ioerrs	Pascal ioresults

SEE ALSO

HP Pascal Language Reference.

DIAGNOSTICS

The diagnostics produced by *pc* are intended to be self-explanatory. If a listing is requested (**–L** option), errors are written to the listing file. If no listing is being generated, errors are written to *stderr*. Errors will be written to both the listing file and *stderr* if the **–L** and **–e** options are both specified. Occasional messages may be produced by the linker.

A list of all compiler errors may be found in /usr/lib/paserrs.

NAME

pc – Pascal compiler

SYNOPSIS

pc [options] files

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: HP

Remarks: This manual page describes the Pascal compiler as implemented on the Series 500 computers. Refer to other *pc*(1) manual pages for information valid for other implementations.

DESCRIPTION

Pc is the HP standard Pascal compiler. It accepts two types of file arguments:

(1) Arguments whose names end with **.p** are taken to be Pascal source programs. They are compiled, and each object program is left in the current directory in a file whose name is that of the source, with **.o** substituted for **.p**. (The **.o** file will not be created for a single source which is compiled and loaded, nor for any source which fails to compile correctly.)

(2) Arguments whose names end with **.o** are passed on to the linker (*ld*(1)) to be linked into the final program.

The following options are recognized:

–c Suppress linking and produce object (**.o**) files from source files;

–w Suppress warning messages (same as **$WARN OFF$**);

–L Write a program listing to *stdout* during compilation;

–o outfile Name the output file from the linker *outfile* instead of *a.out*;

–e Write lines containing errors to *stderr*;

–v Write expanded compiler and linker runstrings to *stderr*;

–W [*bytes*] Display (if *bytes* is omitted) or set a Pascal program's working set size. *Bytes* is the number of bytes in the program's working set;

–H [*bytes*] Display (if *bytes* is omitted) or set a Pascal program's maximum heap size. *Bytes* is the maximum number of bytes in the heap.

Any other options are taken to be arguments to *ld*, and are passed along to the linker.

FILES

file.p	input file (Pascal source file)
file.o	object file
a.out	linked executable output file
/bin/pc	mother program
/usr/lib/pascomp	compiler
/usr/lib/paserrs	error message file
/lib/prt0.o	runtime startup
/lib/libpc.a	Pascal run-time library
/usr/tmp/*	temporary files used by the compiler; names are created by *tmpnam*(3S).

SEE ALSO

Pascal/9000 Language Reference Manual.
Programming in Pascal with Hewlett-Packard Pascal, by Peter Grogono.

DIAGNOSTICS

The diagnostics produced by *pc* are intended to be self-explanatory. If a listing is requested (**–L** option),

errors are written to the listing file. If no listing is being generated, errors are written to *stderr*. Errors will be written to both the listing file and *stderr* if the **–L** and **–e** options are both specified. Occasional messages may be produced by the linker.

A list of all errors may be found in /usr/lib/paserrs.

NAME

pr – print files

SYNOPSIS

pr [options] [files]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Pr prints the named files on the standard output. If *file* is –, or if no files are specified, the standard input is assumed. By default, the listing is separated into pages, each headed by the page number, a date and time, and the name of the file.

By default, columns are of equal width, separated by at least one space; lines which do not fit are truncated. If the **–s** option is used, lines are not truncated and columns are separated by the separation character.

If the standard output is associated with a terminal, error messages are withheld until *pr* has completed printing.

Options may appear singly or be combined in any order. Their meanings are:

+k Begin printing with page *k* (default is 1).

–k Produce *k*-column output (default is 1). The options **–e** and **–i** are assumed for multi-column output.

–a Print multi-column output across the page.

–d Double-space the output.

–e*ck* Expand *input* tabs to character positions $k+1$, $2*k+1$, $3*k+1$, etc. If k is 0 or is omitted, default tab settings at every eighth position are assumed. Tab characters in the input are expanded into the appropriate number of spaces. If *c* (any non-digit character) is given, it is treated as the input tab character (default for *c* is the tab character).

–f Use form-feed character for new pages (default is to use a sequence of line-feeds). Pause before beginning the first page if the standard output is associated with a terminal.

–h Use the next argument as the header to be printed instead of the file name.

–i*ck* In *output*, replace white space wherever possible by inserting tabs to character positions $k+1$, $2*k+1$, $3*k+1$, etc. If k is 0 or is omitted, default tab settings at every eighth position are assumed. If *c* (any non-digit character) is given, it is treated as the output tab character (default for *c* is the tab character).

–l*k* Set the length of a page to *k* lines (default is 66).

–m Merge and print all files simultaneously, one per column (overrides the **–***k*, and **–a** options).

–n*ck* Provide *k*-digit line numbering (default for *k* is 5). The number occupies the first $k+1$ character positions of each column of normal output or each line of **–m** output. If *c* (any non-digit character) is given, it is appended to the line number to separate it from whatever follows (default for *c* is a tab).

–o*k* Offset each line by *k* character positions (default is 0). The number of character positions per line is the sum of the width and offset.

–p Pause before beginning each page if the output is directed to a terminal (*pr* will ring the bell at the terminal and wait for a carriage return).

–r Print no diagnostic reports on failure to open files.

–s*c* Separate columns by the single character *c* instead of by the appropriate number of spaces (default for *c* is a tab).

–t Print neither the five-line identifying header nor the five-line trailer normally supplied for each page. Quit printing after the last line of each file without spacing to the end of the page.

–w*k* Set the width of a line to *k* character positions (default is 72 for equal-width multi-column output, no limit otherwise).

EXAMPLES

Print **file1** and **file2** as a double-spaced, three-column listing headed by "file list":

```
pr –3dh "file list" file1 file2
```

Write **file1** on **file2**, expanding tabs to columns 10, 19, 28, 37, . . . :

```
pr –e9 –t <file1 >file2
```

FILES

/dev/tty* to suspend messages

SEE ALSO

cat(1), lp(1), ul(1).

NAME

prof – display profile data

SYNOPSIS

prof [–**a**] [–**l**] [file]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

Remarks: *Prof*(1) is implemented on the Series 200 only.

DESCRIPTION

Prof interprets the file **mon.out** produced by the *monitor*(3C) subroutine. Under default modes, the symbol table in the named object file (**a.out** default) is read and correlated with the **mon.out** profile file. For each external symbol, the percentage of time spent executing between that symbol and the next is printed (in decreasing order), together with the number of times that routine was called and the number of milliseconds per call.

If the –**a** option is used, all symbols are reported rather than just external symbols. If the –**l** option is used, the output is listed by symbol value rather than decreasing percentage.

In order for the number of calls to a routine to be tallied, the –**p** option of *cc* must have been given when the file containing the routine was compiled. This option also arranges for the **mon.out** file to be produced automatically.

FILES

mon.out for profile
a.out for namelist

SEE ALSO

cc(1), profil(2), monitor(3C).

BUGS

Beware of quantization errors.

NAME

prs – print and summarize an SCCS file

SYNOPSIS

prs [–**d**[dataspec]] [–**r**[SID]] [–**e**] [–**l**] [–**a**] files

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Prs prints, on the standard output, parts or all of an SCCS file (see *sccsfile*(5)) in a user supplied format. If a directory is named, *prs* behaves as though each file in the directory were specified as a named file, except that 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; non-SCCS files and unreadable files are silently ignored.

Arguments to *prs*, which may appear in any order, consist of *keyletter* arguments, and file names.

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

–d[*dataspec*] Used to specify the output data specification. The *dataspec* is a string consisting of SCCS file *data keywords* (see *Data Keywords*) interspersed with optional user supplied text.

–r[SID] Used to specify the *S*CCS *ID*entification (SID) string of a delta for which information is desired. If no SID is specified, the SID of the most recently created delta is assumed. If an SID is specified, it must agree exactly with an SID in the file (i.e. the SID structure used by *get*(1) does not work here).

–e Requests information for all deltas created *earlier* than and including the delta designated via the **–r** keyletter.

–l Requests information for all deltas created *later* than and including the delta designated via the **–r** keyletter.

–a Requests printing of information for both removed, i.e., delta type = *R*, (see *rmdel*(1)) and existing, i.e., delta type = *D*, deltas. If the **–a** keyletter is not specified, information for existing deltas only is provided.

Data Keywords

Data keywords specify which parts of an SCCS file are to be retrieved and output. All parts of an SCCS file (see *sccsfile*(5)) have an associated data keyword. There is no limit on the number of times a data keyword may appear in a *dataspec*.

The information printed by *prs* consists of: (1) the user supplied text; and (2) appropriate values (extracted from the SCCS file) substituted for the recognized data keywords in the order of appearance in the *dataspec*. The format of a data keyword value is either *Simple* (S), in which keyword substitution is direct, or *Multi-line* (M), in which keyword substitution is followed by a carriage return.

Keyword	Data Item	File Selection	Value	Format
:Dt:	Delta information	Delta Table	See below*	S
:DL:	Delta line statistics	"	:Li:/:Ld:/:Lu:	S
:Li:	Lines inserted by Delta	"	nnnnn	S
:Ld:	Lines deleted by Delta	"	nnnnn	S
:Lu:	Lines unchanged by Delta	"	nnnnn	S
:DT:	Delta type	"	D or R	S
:I:	SCCS ID string (SID)	"	:R:.:L:.:B:.:S:	S
:R:	Release number	"	nnnn	S
:L:	Level number	"	nnnn	S
:B:	Branch number	"	nnnn	S
:S:	Sequence number	"	nnnn	S
:D:	Date Delta created	"	:Dy:/:Dm:/:Dd:	S
:Dy:	Year Delta created	"	nn	S
:Dm:	Month Delta created	"	nn	S
:Dd:	Day Delta created	"	nn	S
:T:	Time Delta created	"	:Th:::Tm:::Ts:	S
:Th:	Hour Delta created	"	nn	S
:Tm:	Minutes Delta created	"	nn	S
:Ts:	Seconds Delta created	"	nn	S
:P:	Programmer who created Delta	"	logname	S
:DS:	Delta sequence number	"	nnnn	S
:DP:	Predecessor Delta seq-no.	"	nnnn	S
:DI:	Seq-no. of deltas incl., excl., ignored	"	:Dn:/:Dx:/:Dg:	S
:Dn:	Deltas included (seq#)	"	:DS: :DS:...	S
:Dx:	Deltas excluded (seq#)	"	:DS: :DS:...	S
:Dg:	Deltas ignored (seq#)	"	:DS: :DS:...	S
:MR:	MR numbers for delta	"	text	M
:C:	Comments for delta	"	text	M
:UN:	User names	User Names	text	M
:FL:	Flag list	Flags	text	M
:Y:	Module type flag	"	text	S
:MF:	MR validation flag	"	*yes or no*	S
:MP:	MR validation pgm name	"	text	S
:KF:	Keyword error/warning flag	"	*yes or no*	S
:BF:	Branch flag	"	*yes or no*	S
:J:	Joint edit flag	"	*yes or no*	S
:LK:	Locked releases	"	:R:...	S
:Q:	User defined keyword	"	text	S
:M:	Module name	"	text	S
:FB:	Floor boundary	"	:R:	S
:CB:	Ceiling boundary	"	:R:	S
:Ds:	Default SID	"	:I:	S
:ND:	Null delta flag	"	*yes or no*	S
:FD:	File descriptive text	Comments	text	M
:BD:	Body	Body	text	M
:GB:	Gotten body	"	text	M
:W:	A form of *what*(1) string	N/A	:Z::M:\t:I:	S
:A:	A form of *what*(1) string	N/A	:Z::Y: :M: :I::Z:	S
:Z:	*what*(1) string delimiter	N/A	@(#)	S
:F:	SCCS file name	N/A	text	S
:PN:	SCCS file path name	N/A	text	S

* :Dt: = :DT: :I: :D: :T: :P: :DS: :DP:

User supplied text is any text other than recognized data keywords. Escapes may be used as follows:

tab	\t
new-line	\n
colon	\:
backspace	\b
carriage-return	\r
form feed	\f
backslash	\\
single quote	\'

The default *dataspec* is:

:Dt:\t:DL:\n:MRs:\n:MR:COMMENTS:\n:C:

If no option letters (or only **-a**) are given, *prs* prints the file name, using the default *dataspec*, and the **-e** option; thus, information on all deltas is produced.

EXAMPLES

prs –d"Users and/or user IDs for :F: are:\n:UN:" s.file

may produce on the standard output:

Users and/or user IDs for s.file are:
xyz
131
abc

prs –d"Newest delta for pgm :M:: :I: Created :D: By :P:" –r s.file

may produce on the standard output:

Newest delta for pgm main.c: 3.7 Created 77/12/1 By cas

As a special case (when no **–d** keyletter is given):

prs s.file

may produce on the standard output:

D 1.1 77/12/1 00:00:00 cas 1 000000/00000/00000
MRs:
bl78-12345
bl79-54321
COMMENTS:
this is the comment line for s.file initial delta

for each delta table entry of the "D" type.

FILES

/tmp/pr????? {temp files exist only while psr is active.}

SEE ALSO

admin(1), delta(1), get(1), help(1), sccsfile(5).

SCCS User's Guide in *HP-UX Concepts and Tutorials.*

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

ps – report process status

SYNOPSIS

ps [**–edafl**] [**–s** swapdev] [**–n** namelist] [**–t** tlist] [**–p** plist] [**–u** ulist] [**–g** glist]

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

Ps prints certain information about active processes. With no options, information is printed about processes associated with the current terminal. Otherwise, the information that is displayed is controlled by the following options:

–a Print information about all processes, except process group leaders and processes not associated with a terminal.

–d Print information about all processes, except process group leaders.

–e Print information about all processes.

–f Generate a *full* listing. (Normally, a short listing containing only process ID, terminal ("tty") identifier, cumulative execution time, and the command name is printed.) See below for meaning of columns in a full listing.

–g *glist* Restrict listing to data about processes whose process groups are given in *glist*, where *glist* is a list of process group leaders and is in the same format as *tlist*.

–l Generate a *long* listing. See below.

–n *namelist* The argument will be taken as the name of an alternate *namelist* (/**unix** is the default).

–p *plist* Restrict listing to data about processes whose process ID numbers are given in *plist*, where *plist* is in the same format as *tlist*.

–s *swapdev* Use the file *swapdev* in place of /**dev**/**swap**. This is useful when examining a *corefile*; a *swapdev* of /**dev**/**null** will cause the user block to be zeroed out.

–t *tlist* Restrict listing to data about the processes associated with the terminals given in *tlist*, where *tlist* can be in one of two forms: a list of terminal identifiers separated from one another by a comma, or a list of terminal identifiers enclosed in double quotes and separated from one another by a comma and/or one or more spaces.

–u *ulist* Restrict listing to data about processes whose user ID numbers or login names are given in *ulist*, where *ulist* is in the same format as *tlist*. In the listing, the numerical user ID will be printed unless the **–f** option is used, in which case the login name will be printed.

The column headings and the meaning of the columns in a *ps* listing are given below. The letters **f** and **l** indicate the option (*full* or *long*) that causes the corresponding heading to appear. **All** means that the heading always appears. Note that these two options only determine what information is provided for a process; they do *not* determine which processes will be listed.

F (l) Flags (octal and additive) associated with the process:

01	in core;
02	system process;
04	locked in core (e.g., for physical I/O);
10	currently being swapped out;
20	being traced by another process;
40	another trace flag;
100	text pointer valid;
200	partially swapped out.

S (l) The state of the process:

O	non-existent;
S	sleeping;

W waiting;
R running;
I intermediate;
Z terminated;
T stopped.

UID	(f,l)	The user ID number of the process owner; the login name is printed under the **–f** option.
PID	(all)	The process ID of the process; it is possible to kill a process if you know this datum.
PPID	(f,l)	The process ID of the parent process.
C	(f,l)	Processor utilization for scheduling.
STIME	(f)	Starting time of the process.
PRI	(l)	The priority of the process; higher numbers mean lower priority.
NI	(l)	Nice value; used in priority computation.
ADDR	(l)	The memory address of the process, if resident; otherwise, the disk address.
SZ	(l)	The size in blocks of the core image of the process.
WCHAN	(l)	The event for which the process is waiting or sleeping; if blank, the process is running.
TTY	(all)	The controlling terminal for the process (without the initial "tty", if any).
TIME	(all)	The cumulative execution time for the process (reported in the form "min:sec").
CMD	(all)	The command name; the full command name and its arguments are printed under the **–f** option.

A process that has exited and has a parent, but has not yet been waited for by the parent, is marked <**defunct**> (see "zombie process" in *exit*(2)).

Under the **–f** option, *ps* tries to determine the command name and arguments given when the process was created by examining memory or the swap area. Failing this, the command name, as it would appear without the **–f** option, is printed in square brackets.

HARDWARE DEPENDENCIES

Series 200:

ADDR is the page frame number of the *u_area*, if resident.

Series 500:

The **F** field is always 01.
In the **S** field, **I** means "waiting for input from terminal".
In the **S** field, the **P** (paused) state is added.
In the **S** field, the **T** state is not currently supported.
The **C** field is always zero.
The **ADDR** field reports the partition number.
In the **SZ** field, the block size is 1K bytes.
The **WCHAN** field is always blank.
The **CMD** field is renamed **COMMAND** except when the **–fl** option is specified.

The definition of **STIME** is as follows:

The time when the process was forked, *not* the time when it was modified by *exec*; the date is included only if the elapsed time is greater than 24 hours.

The **s** and **n** options are not currently supported. A diagnostic is printed if they are used.

The file */dev/mem* does not exist.

FILES

/unix	system namelist
/dev/mem	memory
/dev	searched to find swap device and terminal (tty) names

SEE ALSO

kill(1), nice(1), exec(2), exit(2).

BUGS

Things can change while *ps* is running; the picture it gives is only a snapshot in time. Some data printed for defunct processes are irrelevant.

If two special files for terminals are located at the same select code, they are reported in the order in which they appear in /**dev**, not in alphabetical order.

NAME

ptx – permuted index

SYNOPSIS

ptx [options] [input [output]]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Ptx generates the file *output* that can be processed with a text formatter to produce a permuted index of file *input* (standard input and output default). It has three phases: the first does the permutation, generating one line for each keyword in an input line. The keyword is rotated to the front. The permuted file is then sorted. Finally, the sorted lines are rotated so the keyword comes at the middle of each line. *Ptx* output is in the form:

.xx "tail" "before keyword" "keyword and after" "head"

where **.xx** is assumed to be an *nroff* or *troff*(1) macro provided by the user, or provided by the *mptx*(7) macro package. The *before keyword* and *keyword and after* fields incorporate as much of the line as will fit around the keyword when it is printed. *Tail* and *head*, at least one of which is always the empty string, are wrapped-around pieces small enough to fit in the unused space at the opposite end of the line.

The following *options* can be applied:

–f Fold upper- and lower-case letters for sorting.

–t Prepare the output for the phototypesetter.

–w *n* Use the next argument, *n*, as the length of the output line. The default line length is 72 characters for *nroff* and 100 for *troff*.

–g *n* Use the next argument, *n*, as the number of characters that *ptx* will reserve in its calculations for each gap among the four parts of the line as finally printed. The default gap is 3.

–o *only* Use as keywords only the words given in the *only* file.

–i *ignore* Do not use as keywords any words given in the *ignore* file. If the **–i** and **–o** options are missing, use */usr/lib/eign* as the *ignore* file.

–b *break* Use the characters in the *break* file to separate words. Tab, new-line, and space characters are *always* used as break characters.

–r Take any leading non-blank characters of each input line to be a reference identifier (as to a page or chapter), separate from the text of the line. Attach that identifier as a 5th field on each output line.

FILES

/bin/sort
/usr/lib/eign
/usr/lib/tmac/tmac.ptx

SEE ALSO

nroff(1), troff(1), mm(7), mptx(7).

BUGS

Line length counts do not account for overstriking or proportional spacing.

Lines that contain tildes (~) are botched, because *ptx* uses that character internally.

NAME

pwd – working directory name

SYNOPSIS

pwd

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Pwd prints the path name of the working (current) directory.

SEE ALSO

cd(1).

DIAGNOSTICS

"Cannot open .." and "Read error in .." indicate possible file system trouble and should be referred to the super-user.

NAME

r2780 - IBM 2780/3780 terminal emulation

SYNOPSIS

r2780 [options] devicefile

HP-UX COMPATIBILITY

Level: HP-UX/EXTENDED

Origin: HP

Remarks: *R2780* is implemented on the Series 500 only. *R2780* is part of an optional product numbered 97077A (single-user) or 97087A (multi-user).

DESCRIPTION

R2780 emulates an IBM 2780 or 3780 remote job entry station. In addition, support for character set translation and tracing is provided.

Options

–s*speed* Data rate (**1200 2400 3600 4800 7200 9600 19200 e**). Default is **e**.

–3 Emulate 3780 instead of 2780.

–b Suppress character set translation.

–f Enable full duplex.

–p Enable primary station (shorter timeouts).

–m[**amr**] Select type of modem connection: Manual Dial (**m**), AutoAnswer with ring (**a**), Autoanswer without Ring (**r**). Default is **m**.

–t Enable truncation of trailing blanks.

–c Enable compression of imbedded blanks.

–l*retrylimit* Select limit for retries on bad blocks (**1–255**). The default is (**7**).

–o*recsize* Set maximum record size for outgoing records (**1–509** bytes for 3780, **1–397** bytes for 2780). Default is **80**.

–i*recsize* Set maximum record size for incoming records (**1–509** bytes for 3780, **1–397** bytes for 2780). Default is **160**.

–w*timeout* Select timeout for modem connection (**0–255** seconds). **0** disables this timeout. The default is **60**.

–n*timeout* Select no activity timeout (**0–65535** seconds). **0** disables the no activity timeout. The default is **26**.

–r Signal Rate select. Enable alternate (lower) modem speed.

–v Enable MSV2 protocol.

–k[**artcdbmf**] Trace mask. All events (**a**), received data characters (**r**), transmitted data characters (**t**), received control characters (**c**), transmitted control characters (**d**), backplane state machine transitions (**b**), midplane state machine transitions (**m**), frontplane state machine transitions (**f**).

–y*devicefile* Enable tracing and read trace data from specified device file.

–z[*file or shell command*]
Output filter program (if tracefile begins with "|") or file to receive raw trace data.

Commands

R2780 interprets the following commands from standard input:

*****[> or >>*outfile* or |*outfilter*]
configuration display.

|[> or >>*outfile* or |*outfilter*]
interface card status display.

?[> or >>*outfile* or |*outfilter*]
emulator status display.

! enter new local shell.

!*shell command*
execute one local shell command.

–[*config option*]
set one of the following configuration options: **t**, **c**, **b**, **k**, **z**, or **i**.

+[*config option*]
reset one of the following configuration options: **t**, **c**, **b**, **k**, **z**, or **i**.

>[|][>][:][*file* or *filter*]
receive one file, pipe to filter (|), append to existing file (>), suppress echo (:).

<[#][:][*file*]
transmit one file, transparent (#), suppress echo (:).

<@ send EOT.

The interrupt signal (e.g. ^C or BREAK) can be used to abort file transfers. End of file (e.g. ^D from terminal) can be used to terminate *r2780* or to terminate a file transfer from standard input.

SEE ALSO

r2780trace(1c).

NAME

r2780trace – format a trace dump from r2780

SYNOPSIS

r2780trace

HP-UX COMPATIBILITY

Level: HP-UX/EXTENDED

Origin: HP

Remarks: *R2780trace* is implemented on the Series 500 only. *R2780trace* is part of an optional product numbered 97077A (single-user) or 97087A (multi-user).

DESCRIPTION

R2780trace reads from standard input the results of a trace done by *r2780*(1). The format of the input is expected to be the same as the format of the output of *r2780* when the trace feature of that program is on.

If the overhead information at the beginning of the trace (includes date and time of trace) is not present or is in the wrong format, the trace will immediately abort with an error message. If not, the trace input as a whole will be considered valid, and any subsequent invalid data will result in error messages but not termination.

The output of the trace appears in 3 columns. A transmitted byte is printed in column 2 in each of three formats: octal, ascii, and ebcdic. A received byte is printed in column 3 in each of the above three formats. For both of these cases, the value of the trace timer appears in column 1. Control bytes have no trace time associated with them, are interpreted according to tables of messages found in **/usr/lib/tracetable** and are interpreted on the output without regard for alignment with the three columns mentioned above.

Note: any trace data following an end of trace and preceding a new beginning of trace will not be interpreted, but a count of these thrown-away bytes will be printed.

FILES

/usr/lib/tracetable names of state machines, events, and sub-functions.

SEE ALSO

r2780(1).

NAME

ranlib - convert archives to random libraries

SYNOPSIS

ranlib archive ...

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: Version 7

DESCRIPTION

Ranlib converts each named *archive* to a form which can be loaded more rapidly by the link editor (*ld*). It does this by adding a table of contents named __.**SYMDEF** to the beginning of the *archive*. It uses *ar*(1) to reconstruct the *archive*, so that sufficient temporary file space must be available in the file system containing the current directory. A library's table of contents must be updated each time that library is modified, or *ld* refuses to use the table of contents. The table of contents is updated by running the library through *ranlib* again.

SEE ALSO

ar(1), ld(1), ranlib(5).

BUGS

Because generation of a library by *ar* and creation of the table of contents by *ranlib* are separate, phase errors are possible. The link editor *ld* warns when the modification date of a library is more recent than the creation of its table of contents, but this means that you get the warning even if you only copy the library.

On some systems, the date for a table of contents might be newer than the last modified date of the library because of delayed updating of the i-node information. *Ranlib* tries to compensate for this by modifying the archive header for the table of contents: it adds five seconds to the date. Therefore, on systems that perform "fast" updating of the i-node dates (such as the Series 500), it is sometimes possible to execute one or more commands (e.g. *touch*) on the archive without invalidating the table of contents.

NAME

ratfor – rational Fortran dialect

SYNOPSIS

ratfor [options] [files]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Ratfor converts a rational dialect of Fortran into ordinary irrational Fortran. *Ratfor* provides control flow constructs essentially identical to those in C:

statement grouping:
{ statement; statement; statement }

decision-making:
if (condition) statement [**else** statement]
switch (integer value) {
case integer: statement
...
[**default**:] statement
}

loops:
while (condition) statement
for (expression; condition; expression) statement
do limits statement
repeat statement [**until** (condition)]
break
next

and some syntactic sugar to make programs easier to read and write:

free form input:
multiple statements/line; automatic continuation

comments:
this is a comment.

translation of relationals:
>, >=, etc., become **.GT.**, **.GE.**, etc.

return expression to caller from function:
return (expression)

define: **define** *name replacement*

include:
include *file*

The *options* are as follows:

–h causes quoted strings to be turned into **27H** constructs.

–C copies comments to the output and attempts to format it neatly.

–6c Normally, continuation lines are marked with an **&** in column 1; this option makes the continuation character *c* and places it in column 6.

Ratfor may be used with *f77*(1) and *fc*(1).

HARDWARE DEPENDENCIES

Series 200:

Ratfor options conflict with other *f77*(1) options. Therefore, these options are not recognized by *f77*. To use *ratfor* options, *ratfor* must be called directly.

Series 500:

Fc(1) does not recognize *ratfor* **.r** files. Therefore, *ratfor* must be called directly.

The **–h** option should not be used.

The **–6c** option must be used for successful automatic continuation.

SEE ALSO

efl(1), f77(1), fc(1).

NAME

rev – reverse lines of a file

SYNOPSIS

rev [file] ...

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: UCB 4.2

DESCRIPTION

Rev copies the named files to the standard output, reversing the order of characters in every line. If no file is specified, the standard input is copied.

NAME

revision – get HP-UX revision information

SYNOPSIS

/lbin/revision

HP-UX COMPATIBILITY

Level: HP-UX/EXTENDED

Origin: HP

Remarks: *Revision* is implemented on the Series 500 only.

DESCRIPTION

This command prints six lines to standard output. Those six lines consist of the six data items output by *uname*(2), which give information on the kernel.

The following is a sample output from a machine whose loader chip was not programmed with a serial number:

```
System:     HP-UX
Release:    TEST23#
Version:    B
Machine:
Identity:
Nodename:   hpfcla
```

SEE ALSO

uname(2).

NAME

rm, rmdir – remove files or directories

SYNOPSIS

rm [–**fri**] file ...

rmdir dir ...

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Rm removes the entries for one or more files from a directory. If an entry was the last link to the file, the file is destroyed. Removal of a file requires write permission in its directory, but neither read nor write permission on the file itself.

If a file has no write permission and the standard input is a terminal, its permissions are printed and a line is read from the standard input. If that line begins with **y** the file is deleted, otherwise the file remains. No questions are asked if the standard input is not a terminal.

If a designated file is a directory, an error comment is printed (unless the optional argument –**r** has been used – see below).

The options are:

–**f** removes a file with no questions asked, even if the file has no write permission.

–**r** causes *rm* to recursively delete the entire contents of a directory, and then the directory itself. *Rm* can recursively delete up to 17 levels of directories.

–**i** causes *rm* to ask whether or not to delete each file. If –**r** is also specified, *rm* asks whether to examine each directory encountered.

Rmdir removes entries for the named directories, which must be empty and have execute permission for the user trying to remove them.

SEE ALSO

unlink(2).

DIAGNOSTICS

Generally self-explanatory. It is forbidden to remove the file .. merely to avoid the consequences of inadvertently doing something like:

"rm –r .*"

NAME

rmdel – remove a delta from an SCCS file

SYNOPSIS

rmdel –rSID files

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Rmdel removes the delta specified by the *SID* from each named SCCS file. The delta to be removed must be the newest (most recent) delta in its branch in the delta chain of each named SCCS file. In addition, the SID specified must *not* be that of a version being edited for the purpose of making a delta (i. e., if a *p-file* (see *get*(1)) exists for the named SCCS file, the SID specified must *not* appear in any entry of the *p-file*).

If a directory is named, *rmdel* 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; non-SCCS files and unreadable files are silently ignored.

The exact permissions necessary to remove a delta are documented in the *SCCS User's Guide*. Simply stated, they are either (1) if you make a delta you can remove it; or (2) if you own the file and directory you can remove a delta.

FILES

x-file (see *delta*(1))
z-file (see *delta*(1))

SEE ALSO

delta(1), get(1), help(1), prs(1), sccsfile(5).

SCCS User's Guide in *HP-UX Concepts and Tutorials.*

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

rmnl - remove extra new-line characters from file

SYNOPSIS

rmnl

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: UCB

DESCRIPTION

Rmnl is useful for removing excess white space from files for display on a crt terminal. Groups of more than one \n character are compressed to one \n character. This is used by the *man* command. *Rmnl* provides the same functionality as *ssp*(1).

SEE ALSO

man(1), ssp(1).

NAME

rsh – restricted shell (command interpreter)

SYNOPSIS

rsh [flags] [name [arg1 ...]]

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

Rsh is a restricted version of the standard command interpreter *sh*(1). It is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. The actions of *rsh* are identical to those of *sh*, except that the following are disallowed:

cd
setting the value of **$PATH**
command names containing /
> and >>

When invoked with the name –**rsh**, *rsh* reads the user's **.profile** (from **$HOME/.profile**). It acts as the standard *sh* while doing this, except that an interrupt causes an immediate exit, instead of causing a return to command level. The restrictions above are enforced after **.profile** is interpreted.

When a command to be executed is found to be a shell procedure, *rsh* invokes *sh* 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 restricting him to 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, then leaving the user in an appropriate directory (probably *not* the login directory).

Rsh is actually just a link to *sh* and any *flags* arguments are the same as for *sh*(1).

The system administrator often sets up a directory of commands that can be safely invoked by *rsh*. Some systems also provide a restricted editor *red*.

SEE ALSO

sh(1), profile(5).

BUGS

An illegal command (such as *cd*) included in a trap 0 (logout) will cause a memory fault in the shell.

NAME

sact – print current SCCS file editing activity

SYNOPSIS

sact files

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Sact informs the user of any impending deltas to a named SCCS file. This situation occurs when *get*(1) with the **–e** option has been previously executed without a subsequent execution of *delta*(1). If a directory is named on the command line, *sact* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read with each line being taken as the name of an SCCS file to be processed. Non-SCCS files and unreadable files are ignored.

The output for each named file consists of five fields separated by spaces.

Field 1 specifies the SID of a delta that currently exists in the SCCS file to which changes will be made to make the new delta.

Field 2 specifies the SID for the new delta to be created.

Field 3 contains the logname of the user who will make the delta (i.e. executed a **get –e**).

Field 4 contains the date that **get –e** was executed.

Field 5 contains the time that **get –e** was executed.

SEE ALSO

delta(1), get(1), unget(1).

SCCS User's Guide in *HP-UX Concepts and Tutorials.*

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

sccsdiff – compare two versions of SCCS file

SYNOPSIS

sccsdiff **–r**SID1 –rSID2 [–**p**] [–**s**n] files

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Sccsdiff compares two versions of an SCCS file and generates the differences between the two versions. Any number of SCCS files may be specified, but arguments apply to all files.

–r*SID?* SID1 and SID2 specify the deltas of an SCCS file that are to be compared. Versions are passed to *bdiff*(1) in the order given. The SID's accepted and the corresponding version retrieved for the comparison are the same as for *get*(1).

–p pipe output for each file through *pr*(1).

–s*n* *n* is the file segment size that *bdiff* will pass to *diff*(1). This is useful when *diff* fails due to a high system load.

SEE ALSO

bdiff(1), diff(1), get(1), help(1), pr(1).

SCCS User's Guide in *HP-UX Concepts and Tutorials.*

DIAGNOSTICS

"*File*: No differences" if the two versions are the same.
Use *help*(1) for explanations.

NAME

sed – stream text editor

SYNOPSIS

sed [–**n**] [–**e** script] [–**f** sfile] [files]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Sed copies the named *files* (standard input default) to the standard output, edited according to a script of commands. The –**f** option causes the script to be taken from file *sfile*; these options accumulate. If there is just one –**e** option and no –**f** options, the flag –**e** may be omitted. The –**n** option suppresses the default output. A script consists of editing commands, one per line, of the following form:

[address [, address]] function [arguments]

In normal operation, *sed* cyclically copies a line of input into a *pattern space* (unless there is something left after a **D** command), applies in sequence all commands whose *addresses* select that pattern space, and at the end of the script copies the pattern space to the standard output (except under –**n**) and deletes the pattern space.

Some of the commands use a *hold space* to save all or part of the *pattern space* for subsequent retrieval.

An *address* is either a decimal number that counts input lines cumulatively across files, a **$** that addresses the last line of input, or a context address, i.e., a */regular expression/* in the style of *ed*(1) modified thus:

In a context address, the construction \?*regular expression*?, where ? is any character, is identical to */regular expression/*. Note that in the context address **xabc****xdefx**, the second **x** stands for itself, so that the regular expression is **abcxdef**.

The escape sequence **n** matches a new-line *embedded* in the pattern space.

A period **.** matches any character except the *terminal* new-line of the pattern space.

A command line with no addresses selects every pattern space.

A command line with one address selects each pattern space that matches the address.

A command line with two addresses selects the inclusive range from the first pattern space that matches the first address through the next pattern space that matches the second. (If the second address is a number less than or equal to the line number first selected, only one line is selected.) Thereafter the process is repeated, looking again for the first address.

Editing commands can be applied only to non-selected pattern spaces by use of the negation function **!** (below).

In the following list of functions the maximum number of permissible addresses for each function is indicated in parentheses.

The *text* argument consists of one or more lines, all but the last of which end with \ to hide the new-line. Backslashes in text are treated like backslashes in the replacement string of an **s** command, and may be used to protect initial blanks and tabs against the stripping that is done on every script line. The *rfile* or *wfile* argument must terminate the command line and must be preceded by exactly one blank. Each *wfile* is created before processing begins. There can be at most 10 distinct *wfile* arguments.

(1) **a**\
text Append. Place *text* on the output before reading the next input line.

(2) **b** *label* Branch to the **:** command bearing the *label*. If *label* is empty, branch to the end of the script.

(2) **c**\
text Change. Delete the pattern space. With 0 or 1 address or at the end of a 2-address range, place *text* on the output. Start the next cycle.

(2) **d** Delete the pattern space. Start the next cycle.

(2) **D** Delete the initial segment of the pattern space through the first new-line. Start the next

cycle.

(2) **g** Replace the contents of the pattern space by the contents of the hold space.

(2) **G** Append the contents of the hold space to the pattern space.

(2) **h** Replace the contents of the hold space by the contents of the pattern space.

(2) **H** Append the contents of the pattern space to the hold space.

(1) **i**\
text Insert. Place *text* on the standard output.

(2) **l** List the pattern space on the standard output in an unambiguous form. Non-printing characters are spelled in two-digit ASCII and long lines are folded.

(2) **n** Copy the pattern space to the standard output. Replace the pattern space with the next line of input.

(2) **N** Append the next line of input to the pattern space with an embedded new-line. (The current line number changes.)

(2) **p** Print. Copy the pattern space to the standard output.

(2) **P** Copy the initial segment of the pattern space through the first new-line to the standard output.

(1) **q** Quit. Branch to the end of the script. Do not start a new cycle.

(2) **r** *rfile* Read the contents of *rfile*. Place them on the output before reading the next input line.

(2) **s**/*regular expression*/*replacement*/*flags*

Substitute the *replacement* string for instances of the *regular expression* in the pattern space. Any character may be used instead of /. For a fuller description see *ed*(1). *Flags* is zero or more of:

g Global. Substitute for all nonoverlapping instances of the *regular expression* rather than just the first one.

p Print the pattern space if a replacement was made.

w *wfile* Write. Append the pattern space to *wfile* if a replacement was made.

(2) **t** *label* Test. Branch to the **:** command bearing the *label* if any substitutions have been made since the most recent reading of an input line or execution of a **t**. If *label* is empty, branch to the end of the script.

(2) **w** *wfile* Write. Append the pattern space to *wfile*.

(2) **x** Exchange the contents of the pattern and hold spaces.

(2) **y**/*string1*/*string2*/

Transform. Replace all occurrences of characters in *string1* with the corresponding character in *string2*. The lengths of *string1* and *string2* must be equal.

(2)**!** *function*

Don't. Apply the *function* (or group, if *function* is {) only to lines *not* selected by the address(es).

(0) **:** *label* This command does nothing; it bears a *label* for **b** and **t** commands to branch to.

(1) = Place the current line number on the standard output as a line.

(2) { Execute the following commands through a matching } only when the pattern space is selected. The syntax is:

```
{ cmd1
cmd2
cmd3
.
.
.
}
```

(0) An empty command is ignored.

SEE ALSO

awk(1), ed(1), grep(1).

BUGS

There is a limit of 100 commands in the script.

NAME

sh – shell, the standard command programming language

SYNOPSIS

sh [–**ceiknrstuvx**] [args]

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

Sh is a command programming language that executes commands read from a terminal or a file. See *Invocation* below for the meaning of arguments to the shell.

Commands.

A *simple-command* is a sequence of non-blank *words* separated by *blanks* (a *blank* is a tab or a space). 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.

A *list* is a sequence of one or more pipelines separated by **;**, **&**, **&&**, or | |, and optionally terminated by **;** or **&**. Of these four 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 (i.e., the shell does *not* wait for that pipeline to finish). The symbol **&&** (| |) causes the *list* following it to be executed only if the preceding pipeline returns a zero (non-zero) exit status. An arbitrary number of new-lines may appear in a *list*, instead of semicolons, to delimit commands.

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 *name* [**in** *word* ...] **do** *list* **done**

Each time a **for** command is executed, *name* 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* below). Execution ends when there are no more words in the list.

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**

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 sub-shell.

{*list*;}

list is simply executed.

The following 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 { }

Comments.

A word beginning with # causes that word and all the following characters up to a new-line to be ignored.

Command Substitution.

The standard output from a command enclosed in a pair of grave accents (``) may be used as part or all of a word; trailing new-lines are removed.

Parameter Substitution.

The character **$** is used to introduce substitutable *parameters*. Positional parameters may be assigned values by **set**. Variables may be set by writing:

name = *value* [*name* = *value*] ...

Pattern-matching is not performed on *value*.

${*parameter*}

A *parameter* is a sequence of letters, digits, or underscores (a *name*), a digit, or any of the characters *, @, #, ?, –, **$**, and **!**. The value, if any, of the parameter is substituted. The braces are required only when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name. A *name* must begin with a letter or underscore. If *parameter* is a digit then it is a positional parameter. If *parameter* is * or @, then all the positional parameters, starting with **$1**, are substituted (separated by spaces). Parameter **$0** is set from argument zero when the shell is invoked.

${*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 the message "parameter null or not set" is printed.

${*parameter*:+*word*}

If *parameter* is set and is non-null then substitute *word*; otherwise substitute nothing.

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 synchronously executed command, if the command exited normally. Otherwise, the signal number which the command terminated on, either 1-127 if core was not dumped, or 129-255 if core was dumped.
$	The process number of this shell.
!	The process number of the last background command invoked.

The following parameters are used by the shell:

HOME	The default argument (home directory) for the *cd* command.
PATH	The search path for commands (see *Execution* below).
MAIL	If this variable is set to the name of a mail file, then the shell informs the user of the arrival of mail in the specified file.
SHELL	a string specifying which shell to run.
PS1	Primary prompt string, by default "**$** ".
PS2	Secondary prompt string, by default "> ".
IFS	Internal field separators, normally **space**, **tab**, and **new-line**.

The shell gives default values to **PATH**, **PS1**, **PS2**, and **IFS**, while **HOME** and **MAIL** are not set at all by the shell (although **HOME** *is* set by *login*(1)).

Blank Interpretation.

After parameter and command substitution, the results of substitution are scanned for internal 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 **[**. If one of these characters appears then the word is regarded as a *pattern*. The word is replaced with alphabetically sorted file names that match the pattern. If no file name is found that matches the pattern, then the word is left unchanged. The character **.** at the start of a file name or immediately following a /, as well as the character / itself, must be matched explicitly.

*	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. A NOT operator, **!**, can be specified immediately following the left bracket to match any single character not enclosed in the brackets.

Quoting.

The following characters have a special meaning to the shell and cause termination of a word unless quoted:

; & ( ) | < > new-line space tab

A character may be *quoted* (i.e., 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 (''), except a single quote, are quoted. Inside double quote marks (" "), parameter and command substitution occurs and \ quotes the characters \, `, ", and **$**. "**$***" is equivalent to "**$1 $2** ...", whereas "**$@**" is equivalent to "**$1**" "**$2**" ....

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 (i.e., the value of **PS2**) is issued.

Input/Output.

Before a command is executed, its input and output may be redirected using a special notation interpreted by the shell. The following may appear anywhere in a simple-command or may precede or follow a *command* and are *not* passed on to the invoked command; substitution occurs before *word* or *digit* is used:

<**word**	Use file *word* as standard input (file descriptor 0).

>word	Use file *word* as standard output (file descriptor 1). If the file does not exist then it is created; otherwise, it is truncated to zero length.
>>word	Use file *word* as standard output. If the file exists then output is appended to it (by first seeking to the end-of-file); otherwise, the file is created.
<<[–]word	The shell input is read up to a line that is the same as *word*, or to an end-of-file. The resulting 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, (unescaped) **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 >.
<&–	The standard input is closed. Similarly for the standard output using >.

If one of the above is preceded by a digit, then the file descriptor created is that specified by the digit (instead of the default 0 or 1). For example:

```
... 2>&1
```

creates file descriptor 2 that is a duplicate of file descriptor 1. Note that this type of I/O redirection is necessary if you want to *synchronously* collect stdout and stderr output in the same file. Redirecting stdout and stderr separately will cause asynchronous collection of data at the destination (i.e. things written to stdout can subsequently be over-written by things written to stderr, and vice-versa).

If a command is followed by **&** 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 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. Executed commands inherit the same environment. If the user modifies the values of these parameters or creates new ones, none of these affects the environment unless the **export** command is used to bind the shell's parameter to the environment. The environment seen by any executed command is thus composed of any unmodified name-value pairs originally inherited by the shell, plus any modifications or additions, all of which must be noted in **export** commands.

The environment for any *simple-command* may be augmented by prefixing it with one or more assignments to parameters. 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* keyword 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
```

Signals.

The INTERRUPT and QUIT signals for an invoked command are ignored if the command is followed by **&**; otherwise signals have the values inherited by the shell from its parent, with the exception of signal 11 (but see also the **trap** command below).

Execution.

Each time a command is executed, the above substitutions are carried out. Except for the *Special Commands* listed below, a new 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 the current directory, **/bin**, and **/usr/bin**, in that order). 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 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 an **a.out** file, it is assumed to be a file containing shell commands. A sub-shell (i.e., a separate process) is spawned to read it. A parenthesized command is also executed in a sub-shell.

Special Commands.

The following commands are executed in the shell process and, except as specified, no input/output redirection is permitted for such commands:

: No effect; the command does nothing. A zero exit code is returned.

. *file* Read and execute commands from *file* and return. The search path specified by **PATH** is used to find the directory containing *file*. Note that this command does not spawn another shell to execute *file*, and thus differs in behavior and output from executing *file* as a shell script.

break [*n*]
Exit from the enclosing **for** or **while** loop, if any. If *n* is specified then break *n* levels.

continue [*n*]
Resume the next iteration of the enclosing **for** or **while** loop. If *n* is specified then resume at the *n*-th enclosing loop.

cd [*arg*]
Change the current directory to *arg*. The shell parameter **HOME** is the default *arg*.

eval [*arg* . . .]
The arguments are read as input to the shell and the resulting command(s) executed.

exec [*arg* . . .]
The command specified by the arguments is executed in place of this shell without creating a new process. Input/output arguments may appear and, if no other arguments are given, cause the shell input/output to be modified.

exit [*n*]
Causes a 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.)

export [*name* . . .]
The given *name*s are marked for automatic export to the *environment* of subsequently-executed commands. If no arguments are given, then a list of all names that are exported in this shell is printed.

newgrp [*arg* . . .]
Equivalent to **exec newgrp** *arg* . . . .

read [*name* . . .]
One line is read from the standard input and the first word is assigned to the first *name*, the second word to the second *name*, etc., with leftover words assigned to the last *name*. The return code is 0 unless an end-of-file is encountered.

readonly [*name* . . .]
The given *name*s are marked *readonly* and the values of the these *name*s may not be changed by subsequent assignment. If no arguments are given, then a list of all *readonly* names is printed.

set [**–ekntuvx** [*arg* . . .]]

–e If the shell is non-interactive then exit immediately if a command exits with a non-zero exit status.

–k All keyword arguments are placed in the environment for a command, not just those that precede the command name.

–n Read commands but do not execute them.

–t Exit after reading and executing one command.

–u Treat unset variables as an error when substituting.

–v Print shell input lines as they are read.

–x Print commands and their arguments as they are executed.

–– Do not change any of the flags; useful in setting $1 to –.

Using + rather than – causes these flags to be turned off. These flags can also be used upon invocation of the shell. The current set of flags may be found in **$–**. The remaining arguments are positional parameters and are assigned, in order, to **$1**, **$2**, ... If no arguments are given then the values of all names are printed.

shift

The positional parameters from **$2** ... are renamed **$1** ....

test

Evaluate conditional expressions. See *test*(1) for usage and description.

times

Print the accumulated user and system times for processes run from the shell.

trap [*arg*] [*n*] ...

arg is a command to be read and executed when the shell receives signal(s) *n*. (Note that *arg* is scanned once when the trap is set and once when the trap is taken.) 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. An attempt to trap on signal 11 (memory fault) produces an error. If *arg* is absent then all trap(s) *n* 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 *n* is 0 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.

umask [*nnn*]

The user file-creation mask is set to *nnn* (see *umask*(2)). If *nnn* is omitted, the current value of the mask is printed.

wait Wait for all child processes to terminate, and report the termination status. If *n* is not given then all currently active child processes are waited for. The return code from this command is always zero.

Invocation.

If the shell is invoked through *exec*(2) and the first character of argument zero is –, commands are initially read from **/etc/profile** and then from **$HOME/.profile**, if such files exist. Thereafter, commands are read as described below, which is also the case when the shell is invoked as **/bin/sh**. The flags below are interpreted by the shell on invocation only; Note that unless the **–c** or **–s** flag is specified, the first argument is assumed to be the name of a file containing commands, and the remaining arguments are passed as positional parameters to that command file:

–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. Any remaining arguments specify the positional parameters. Shell output 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, then this shell is *interactive*. In this case TERMINATE is ignored (so that **kill 0** does not kill an interactive shell) and INTERRUPT 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 (see *rsh*(1)).

The remaining flags and arguments are described under the **set** command above.

If the **SHELL** parameter appears in the environment, and the first character to the right of the right-most slash is an "r", then the shell becomes restricted. The following lines should be included in the **.profile** files of restricted login names:

```
SHELL = /usr/rsh
export SHELL
```

This causes whoever logs in under the restricted login names to be given a restricted shell (see *rsh*(1)).

FILES

/etc/profile
$HOME/.profile
/tmp/sh*
/dev/null

RETURN VALUE

The error codes returned by the shell are:

0 – success;
1 – a built-in command failure (see *Special Commands*);
2 – syntax error;
3 – signal received that is not trapped.

If the shell is non-interactive, it will terminate and pass one of the above as its exit status. If it is interactive, it will not terminate, but $? will be set to one of the above values.

Whenever a child process of the shell dies due to a signal, the shell returns an exit status of 80 hexadecimal + the number of the signal.

SEE ALSO

cd(1), env(1), login(1), newgrp(1), rsh(1), test(1), umask(1), dup(2), exec(2), fork(2), pipe(2), signal(2), umask(2), wait(2), a.out(5), profile(5), environ(7).

BUGS

The command **readonly** (without arguments) produces the same output as the command **export**.

If << is used to provide standard input to an asynchronous process invoked by **&**, the shell gets mixed up about naming the input document; a garbage file **/tmp/sh*** is created and the shell complains about not being able to find that file by another name.

^ is a synonym for |; the use of ^ is discouraged.

The shell assumes it is talking to terminals that only process the least significant seven bits of a character. If your terminal uses all eight bits, you may see some strange strings.

When the shell encounters >>, it does not open the file in append mode. Instead, it opens the file for writing and seeks to the end.

NAME

size – print section sizes of common object files

SYNOPSIS

size [**–o**] [**–x**] files

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

The *size* command produces section size information for each section in common object files. The size of the text, data and bss (uninitialized data) sections are printed along with the total size of the object file. If an archive file is input to the *size* command the information for all archive members is displayed.

By default, numbers are printed in decimal, unless altered by the options described below.

–o print numbers in octal.

–x print numbers in hexadecimal.

HARDWARE DEPENDENCIES

Series 500:

The *text* size shown is the sum of the sizes of all code segments.

The *data* size shown is the sum of the initialized portions of the ddata and idata segments (which may be one or two data segments).

The *bss* size shown is the sum of the uninitialized portions of the ddata and idata segments.

If *size* is run on any commands shipped with HP-UX, the *text* size does not include any shared library segments referenced by the command.

SEE ALSO

as(1), cc(1), ld(1), a.out(5), ar(5).

DIAGNOSTICS

size: name: cannot open
if *name* cannot be read.

size: name: bad magic
if *name* is not an appropriate common object file.

NAME

sleep – suspend execution for an interval

SYNOPSIS

sleep time

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

Sleep suspends execution for *time* seconds. It is used to execute a command after a certain amount of time, as in:

```
(sleep 105; command)&
```

or to execute a command every so often, as in:

```
while true
do
        command
        sleep 37
done
```

SEE ALSO

alarm(2), sleep(3C).

BUGS

Time must be less than 2^32 seconds.

NAME

sort – sort and/or merge files

SYNOPSIS

sort [**–cmubdfinrTt**x] [+pos1 [–pos2]] . . . [**–o** output] [file ...]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Sort sorts lines of all the named files together and writes the result on the standard output. The name – means the standard input. If no input files are named, the standard input is sorted.

The default sort key is an entire line. Default ordering is lexicographic by bytes in machine collating sequence. The ordering is affected globally by the following options, one or more of which may appear.

b Ignore leading blanks (spaces and tabs) in field comparisons.

d "Dictionary" order: only letters, digits and blanks are significant in comparisons.

f Fold upper case letters onto lower case.

i Ignore characters outside the ASCII range 040-0176 in non-numeric comparisons.

n An initial numeric string, consisting of optional blanks, optional minus sign, and zero or more digits with optional decimal point, is sorted by arithmetic value. Option **n** implies option **b**.

r Reverse the sense of comparisons.

t*x* "Tab character" separating fields is *x*.

The notation +*pos1* –*pos2* restricts a sort key to a field beginning at *pos1* and ending just before *pos2*. *Pos1* and *pos2* each have the form *m.n*, optionally followed by one or more of the flags **bdfinr**, where *m* tells a number of fields to skip from the beginning of the line and *n* tells a number of characters to skip further. If any flags are present they override all the global ordering options for this key. If the **b** option is in effect *n* is counted from the first non-blank in the field; **b** is attached independently to *pos2*. A missing *.n* means .0; a missing –*pos2* means the end of the line. Under the –**t***x* option, fields are strings separated by *x*; otherwise fields are non-empty non-blank strings separated by blanks.

When there are multiple sort keys, later keys are compared only after all earlier keys compare equal. Lines that otherwise compare equal are ordered with all bytes significant.

These option arguments are also understood:

c Check that the input file is sorted according to the ordering rules; give no output unless the file is out of sort.

m Merge only, the input files are already sorted.

u Suppress all but one in each set of equal lines. Ignored bytes and bytes outside keys do not participate in this comparison.

o The next argument is the name of an output file to use instead of the standard output. This file may be the same as one of the inputs.

T The next argument is the name of a directory where *sort* should put its temporary files.

EXAMPLES

Print in alphabetical order all the unique spellings in a list of words (capitalized words differ from uncapitalized):

```
sort –u +0f +0 list
```

Print the password file (*passwd*(5)) sorted by user ID (the third colon-separated field):

```
sort –t: +2n /etc/passwd
```

Print the first instance of each month in an already sorted file of (month-day) entries (the options **–um** with just one input file make the choice of a unique representative from a set of equal lines predictable):

```
sort –um +0 –1 dates
```

FILES

/usr/tmp/stm* default temporary files
/tmp/stm*

SEE ALSO

comm(1), join(1), uniq(1).

DIAGNOSTICS

Comments and exits with non-zero status for various trouble conditions and for disorder discovered under option **–c**.

BUGS

Lines that exceed 512 characters (including the new-line) are silently truncated (i.e., the excess characters are ignored).

Sort will not recognize the line in a file if it is not terminated with a new-line character.

Sort uses signed characters for comparison (unlike *strcmp* – see *string*(3C)), so you may get unexpected results when sorting a file that contains ASCII characters with the most significant bit set (characters 128 – 255).

Sort does not understand "missing" fields. For example, if you have a file with the following contents:

Doe,John	mailman	17550	8
Spencer,Joe	plumber		4
Johns,Ann	secretary	15950	
Malley,Dean	engineer	26750	4

you may get unexpected results if you try to sort on the third or fourth fields (all names and associated data are fictitious).

Sort does not expand tabs when counting characters to locate a field.

NAME

spell, spellin, spellout – find spelling errors

SYNOPSIS

spell [options] [files]

/usr/lib/spell/spellin [list]

/usr/lib/spell/spellout [**–d**] list

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Spell collects words from the named *files* and looks them up in a spelling list. Words that neither occur among nor are derivable (by applying certain inflections, prefixes, and/or suffixes) from words in the spelling list are printed on the standard output. If no *files* are named, words are collected from the standard input.

Spell ignores most *nroff*(1), and *tbl* constructions.

The options are:

–v all words not literally in the spelling list are printed, and plausible derivations from the words in the spelling list are indicated.

–b British spelling is checked. Besides preferring *centre*, *colour*, *speciality*, *travelled*, etc., this option insists upon *-ise* in words like *standardise*, Fowler and the OED to the contrary notwithstanding.

–x every plausible stem is printed with = for each word.

The spelling list is based on many sources, and while more haphazard than an ordinary dictionary, is also more effective with respect to proper names and popular technical words. Coverage of the specialized vocabularies of biology, medicine, and chemistry is light.

Pertinent auxiliary files may be specified by name arguments, indicated below with their default settings. Copies of all output are accumulated in the history file. The stop list filters out misspellings (e.g., thier = thy–y + ier) that would otherwise pass.

Two routines help maintain the hash lists used by *spell* (both expect a list of words, one per line, from the standard input):

spellin adds the words on the standard input to the pre-existing *list* and places a new list on the standard output. If no *list* is specified, the new list is created from scratch.

Spellout looks up each word read from the standard input, and prints on the standard output those that are missing from (or, with the **–d** option, present in) the hash list.

HARDWARE DEPENDENCIES

Series 200/500:

eqn, *typo* and *troff* are not currently supported.

FILES

D_SPELL = /usr/lib/spell/hlist[ab]	hashed spelling lists, American & British
S_SPELL = /usr/lib/spell/hstop	hashed stop list
H_SPELL = /usr/lib/spell/spellhist	history file
/tmp/spell.$$	temporary

/usr/lib/spell/spellprog program

SEE ALSO

deroff(1), nroff(1), sed(1), sort(1), tbl(1), tee(1).

BUGS

The spelling list's coverage is uneven; new installations will probably wish to monitor the output for several months to gather local additions; typically, these are kept in a separate local dictionary that is added to the hashed *list* via *spellin*.

British spelling was done by an American.

NAME

split – split a file into pieces

SYNOPSIS

split [*–n*] [file [name]]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Split reads *file* and writes it in *n*-line pieces (default 1000), as many as necessary, onto a set of output files. The name of the first output file is *name* with **aa** appended, and so on lexicographically. If no output name is given, **x** is default.

If no input file is given, or if – is given instead, then the standard input file is used.

NAME

ssp – remove multiple line-feeds from output

SYNOPSIS

ssp

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: 4.2 BSD

DESCRIPTION

Ssp (single-space) removes multiple line-feeds from the standard input and sends the result to the standard output. It is typically used in pipelines like

```
nroff –ms file1 | ssp
```

NAME

strings – find printable strings in binary file

SYNOPSIS

strings [–] [**–o**] [–number] file ...

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: UCB

DESCRIPTION

Strings looks for ASCII strings in a binary file. If no files are specified, stdin is used. A string is any sequence of 4 or more printing characters ending with a new-line or a null. Unless the – flag is given, *strings* only looks in the initialized data space of object files. If the **–o** flag is given, then each string is preceded by its offset in the file (in octal). If the ***–number*** flag is given, then ***number*** is used as the minimum string length (rather than 4).

Strings is useful for identifying random object files and many other things.

SEE ALSO

od(1).

BUGS

The algorithm for identifying strings is extremely primitive.

NAME

strip – remove symbols and relocation bits

SYNOPSIS

strip name ...

HP-UX COMPATIBILITY

Level: HP-UX/DEVELOPMENT

Origin: System III

Remarks: This manual entry describes *strip*(1) as implemented on the Series 200 computers. Refer to other *strip*(1) pages for information valid for other implementations.

DESCRIPTION

Strip removes the symbol table and relocation bits ordinarily attached to the output of the assembler and link editor. This is useful to save space after a program has been debugged.

The effect of *strip* is the same as use of the **–s** option of *ld*.

If *name* is an archive file, *strip* will remove the local symbols from any *a.out* format files it finds in the archive. Certain libraries, such as those residing in **/lib**, have no need for local symbols. By deleting them, the size of the archive is decreased and link editing performance is increased.

FILES

/tmp/stm* temporary file

SEE ALSO

ld(1), a.out(5).

NAME

strip – remove symbol table and debug table

SYNOPSIS

strip name ...

HP-UX COMPATIBILITY

Level: HP-UX/DEVELOPMENT

Origin: System III

Remarks:

This manual page describes *strip* as implemented on the Series 500 computers. Refer to other *strip* manual pages for information valid for other implementations.

DESCRIPTION

Strip removes the symbol table and debug table from an executable object file. This is useful to save space after a program has been debugged. The effect is the same as using the **–s** option of *ld*.

If *name* is a relocatable file, *strip* removes the debug table. If *name* is an archive file, *strip* removes the debug table from any *a.out* format files it finds in the archive.

FILES

/tmp/s*	temporary files
/tmp/sb*	temporary files

SEE ALSO

ld(1), a.out(5).

NAME

stty – set the options for a terminal port

SYNOPSIS

stty [–**a**] [–**g**] [options]

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

Stty sets certain terminal I/O options for the device that is the current standard input; without arguments, it reports the settings of certain options; with the –**a** option, it reports all of the option settings; with the –**g** option, it reports current settings in a form that can be used as an argument to another *stty* command. Detailed information about the modes listed in the first five groups below may be found in *tty*(4). Options in the last group are implemented using options in the previous groups. Note that many combinations of options make no sense, but no sense checking is performed. The options are selected from the following:

Control Modes

parenb (–**parenb**) enable (disable) parity generation and detection.

parodd (–**parodd**) select odd (even) parity.

cs5 cs6 cs7 cs8 select character size (see *tty*(4)).

0 hang up phone line immediately.

50 75 110 134.5 150 200 300 600 900 1200 1800 2400 3600 4800 7200 9600 19200 38400 exta extb
Set terminal baud rate to the number given, if the hardware will support it.

hupcl (–**hupcl**) hang up (do not hang up) modem connection on last close.

hup (–**hup**) same as **hupcl** (–**hupcl**).

cstopb (–**cstopb**) use two (one) stop bits per character.

cread (–**cread**) enable (disable) the receiver.

crts (–**crts**) enable (disable) request-to-send.

clocal (–**clocal**) assume a line without (with) modem control.

Input Modes

ignbrk (–**ignbrk**) ignore (do not ignore) break on input.

ienqak (–**ienqak**) enable (disable) ENQ-ACK handshaking.

brkint (–**brkint**) signal (do not signal) INTR on break.

ignpar (–**ignpar**) ignore (do not ignore) parity errors.

parmrk (–**parmrk**) mark (do not mark) parity errors (see *tty*(4)).

inpck (–**inpck**) enable (disable) input parity checking.

istrip (–**istrip**) strip (do not strip) input characters to seven bits.

inlcr (–**inlcr**) map (do not map) NL to CR on input.

igncr (–**igncr**) ignore (do not ignore) CR on input.

icrnl (–**icrnl**) map (do not map) CR to NL on input.

iuclc (–**iuclc**) map (do not map) upper-case alphabetics to lower case on input.

ixon (–**ixon**) enable (disable) START/STOP output control. Output is stopped by sending an ASCII DC3 and started by sending an ASCII DC1.

ixany (–**ixany**)	allow any character (only DC1) to restart output.
ixoff (–**ixoff**)	request that the system send (not send) START/STOP characters when the input queue is nearly empty/full.
Output Modes	
opost (–**opost**)	post-process output (do not post-process output; ignore all other output modes).
olcuc (–**olcuc**)	map (do not map) lower-case alphabetics to upper case on output.
onlcr (–**onlcr**)	map (do not map) NL to CR-NL on output.
ocrnl (–**ocrnl**)	map (do not map) CR to NL on output.
onocr (–**onocr**)	do not (do) output CRs at column zero.
onlret (–**onlret**)	on the terminal NL performs (does not perform) the CR function.
ofill (–**ofill**)	use fill characters (use timing) for delays.
ofdel (–**ofdel**)	fill characters are DELs (NULs).
cr0 cr1 cr2 cr3	select style of delay for carriage returns (see *tty*(4)).
nl0 nl1	select style of delay for line-feeds (see *tty*(4)).
tab0 tab1 tab2 tab3	select style of delay for horizontal tabs (see *tty*(4)).
bs0 bs1	select style of delay for backspaces (see *tty*(4)).
ff0 ff1	select style of delay for form-feeds (see *tty*(4)).
vt0 vt1	select style of delay for vertical tabs (see *tty*(4)).
Local Modes	
isig (–**isig**)	enable (disable) the checking of characters against the special control characters INTR and QUIT.
icanon (–**icanon**)	enable (disable) canonical input (ERASE and KILL processing).
xcase (–**xcase**)	canonical (unprocessed) upper/lower-case presentation.
echo (–**echo**)	echo back (do not echo back) every character typed.
echoe (–**echoe**)	echo (do not echo) ERASE character as a backspace-space-backspace string. Note: this mode will erase the ERASEed character on many CRT terminals; however, it does *not* keep track of column position and, as a result, may be confusing on escaped characters, tabs, and backspaces.
echok (–**echok**)	echo (do not echo) NL after KILL character.
lfkc (–**lfkc**)	the same as **echok** (–**echok**); obsolete.
echonl (–**echonl**)	echo (do not echo) NL.
noflsh (–**noflsh**)	disable (enable) flush after INTR or QUIT.
Control Assignments	
control-character c	set *control-character* to *c*, where *control-character* is **erase**, **kill**, **intr**, **quit**, **eof**, **eol**, **min**, or **time** (**min** and **time** are used with –**icanon**; see *tty*(4)). If *c* is preceded by an (escaped from the shell) caret (^), then the value used is the corresponding CTRL character (e.g., ^d is a **CTRL-d**); ^? is interpreted as DEL and ^– is interpreted as undefined.
line *i*	set line discipline to *i* ($0 < i < 127$). (See tty(4).)
Combination Modes	
evenp or **parity**	enable **parenb** and **cs7**.

oddp	enable **parenb**, **cs7**, and **parodd**.
–parity, **–evenp**, or **–oddp**	disable **parenb**, and set **cs8**.
raw (**–raw** or **cooked**)	enable (disable) raw input and output (no ERASE, KILL, INTR, QUIT, EOT, or output post processing).
nl (**–nl**)	unset (set) **icrnl**, **onlcr**. In addition **–nl** unsets **inlcr**, **igncr**, **ocrnl**, and **onlret**.
lcase (**–lcase**)	set (unset) **xcase**, **iuclc**, and **olcuc**.
LCASE (**–LCASE**)	same as **lcase** (**–lcase**).
tabs (**–tabs** or **tab3**)	preserve (expand to spaces) tabs when printing.
ek	reset ERASE and KILL characters back to normal # and @.
sane	resets all modes to some reasonable values.
term	set all modes suitable for the terminal type *term*, where *term* is one of **tty33**, **tty37**, **vt05**, **tn300**, **ti700**, **hp**, or **tek**.

HARDWARE DEPENDENCIES

Series 200/500:

Refer to *tty*(4) for a description of the capabilities that are not supported.

SEE ALSO

tabs(1), ioctl(2), tty(4).

NAME

su – become another user

SYNOPSIS

su [–] [name [arg . . .]]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Su allows one to become another user without logging off. The default user *name* is **root** (i.e., super-user).

To use *su*, the appropriate password must be supplied (unless one is already super-user). If the password is correct, *su* will execute a new shell with the user ID set to that of the specified user. To restore normal user ID privileges, type an **EOF** to the new shell.

Any additional arguments are passed to the shell, permitting execution of the shell procedures with restricted privileges (an *arg* of the form **–c** *string* executes *string* via the shell). After the procedure has executed, the new user is logged out, and the original user becomes the current user. When additional arguments are passed, **/bin/sh** is always used. When no additional arguments are passed, *su* uses the shell specified in the password file.

An initial – flag causes the environment to be changed to the one that would be expected if the user actually logged in again. This is done by invoking the shell with an *arg0* of **–su** causing the **.profile** in the home directory of the new user ID to be executed. Otherwise, the environment is passed along unchanged. The super-user's **$PATH** is unconditionally set to **/bin:/etc:/usr/bin**. Note that the **.profile** can check *arg0* for **–sh** or **–su** to determine how it was invoked.

The – option always resets **$PATH** to **/bin:/etc:/usr/bin** for the super-user, and **/bin:/usr/bin** for all others. However, the files */etc/profile* and *.profile* are normally executed anyway, thus restoring the intended value of **$PATH**.

Su will ensure that the super-user gets a "#" prompt to remind him of his additional responsibilities.

Su logs all attempts to *su* in /usr/adm/sulog, including failures. Successful attempts are flagged with "+", failures with "–".

FILES

/etc/passwd	system's password file
/usr/adm/sulog	log of all *su* attempts
$HOME/.profile	user's profile file

SEE ALSO

env(1), login(1), sh(1), environ(7).

NAME

sum – print checksum and block count of a file

SYNOPSIS

sum [**–r**] file

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Sum calculates and prints a 16-bit checksum for the named file, and also prints the number of (1K byte) blocks in the file. **Stdin** is used if no fie names are given. *Sum* is typically used to look for bad spots, or to validate a file communicated over some transmission line. The option **–r** causes an alternate algorithm to be used in computing the checksum.

SEE ALSO

wc(1).

DIAGNOSTICS

"Read error" is indistinguishable from end of file on most devices; check the block count.

NAME

tabs – set tabs on a terminal

SYNOPSIS

tabs [tabspec] [+**m**n] [–**T**type]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Tabs sets the tab stops on the user's terminal according to the tab specification *tabspec*, after clearing any previous settings. The user must be logged in on a terminal with remotely-settable hardware tabs.

If you are using a non-HP terminal, you should keep in mind that behavior will vary for some tab settings.

Four types of tab specification are accepted for *tabspec*: "canned", repetitive, arbitrary, and file. If no *tabspec* is given, the default value is –**8**, i.e., HP-UX "standard" tabs. The lowest column number is 1. Note that for *tabs*, column 1 always refers to the leftmost column on a terminal, even one whose column markers begin at 0.

–*code* Gives the name of one of a set of "canned" tabs. The legal *code*s and their meanings are as follows:

–**a** 1,10,16,36,72
Assembler, IBM S/370, first format

–**a2** 1,10,16,40,72
Assembler, IBM S/370, second format

–**c** 1,8,12,16,20,55
COBOL, normal format

–**c2** 1,6,10,14,49
COBOL compact format (columns 1-6 omitted). Using this code, the first typed character corresponds to card column 7, one space gets you to column 8, and a tab reaches column 12. Files using this tab setup should include a format specification as follows:
<:t–c2 m6 s66 d:>

–**c3** 1,6,10,14,18,22,26,30,34,38,42,46,50,54,58,62,67
COBOL compact format (columns 1-6 omitted), with more tabs than –**c2.** This is the recommended format for COBOL. The appropriate format specification is:
<:t–c3 m6 s66 d:>

–**f** 1,7,11,15,19,23
FORTRAN

–**p** 1,5,9,13,17,21,25,29,33,37,41,45,49,53,57,61
PL/I

–**s** 1,10,55
SNOBOL

–**u** 1,12,20,44
UNIVAC 1100 Assembler

In addition to these "canned" formats, three other types exist:

–*n* A repetitive specification requests tabs at columns $1+n$, $1+2*n$, etc. Of particular importance is the value –**8**: this represents the HP-UX "standard" tab setting, and is the most likely tab setting to be found at a terminal. It is required for use with the *nroff*(1) –**h**

option for high-speed output. Another special case is the value **–0**, implying no tabs at all.

n1,n2,... The arbitrary format permits the user to type any chosen set of numbers, separated by commas, in ascending order. Up to 40 numbers are allowed. If any number (except the first one) is preceded by a plus sign, it is taken as an increment to be added to the previous value. Thus, the tab lists 1,10,20,30 and 1,10, + 10, + 10 are considered identical.

––file If the name of a file is given, *tabs* reads the first line of the file, searching for a format specification. If it finds one there, it sets the tab stops according to it, otherwise it sets them as **–8**. This type of specification may be used to make sure that a tabbed file is printed with correct tab settings, and would be used with the *pr*(1) command:

tabs –– file; pr file

Any of the following may be used also; if a given flag occurs more than once, the last value given takes effect:

–T*type* *Tabs* usually needs to know the type of terminal in order to set tabs and always needs to know the type to set margins. *Type* is a name listed in *term*(7). If no **–T** flag is supplied, *tabs* searches for the **$TERM** value in the *environment* (see *environ*(7)). If no *type* can be found, *tabs* tries a sequence that will work for many terminals.

+m*n* The margin argument may be used for some terminals. It causes all tabs to be moved over *n* columns by making column *n+1* the left margin. If **+m** is given without a value of *n*, the value assumed is 10. The normal (leftmost) margin on most terminals is obtained by **+m0**. The margin for most terminals is reset only when the **+m** flag is given explicitly.

Tab and margin setting is performed via the standard output.

Format specifications are defined and described in *fspec*(5).

SEE ALSO

nroff(1), tset(1), fspec(5), environ(7), term(7).

DIAGNOSTICS

illegal tabs	when arbitrary tabs are ordered incorrectly.
illegal increment	when a zero or missing increment is found in an arbitrary specification.
unknown tab code	when a "canned" code cannot be found.
can't open	if *––file* option used, and file can't be opened.
file indirection	if *––file* option used and the specification in that file points to yet another file. Indirection of this form is not permitted.

BUGS

There is no consistency among different terminals regarding ways of clearing tabs and setting the left margin.

It is generally impossible to usefully change the left margin without also setting tabs.

Tabs clears only 20 tabs (on terminals requiring a long sequence), but is willing to set 40.

NAME

tail – deliver the last part of a file

SYNOPSIS

tail [[±[number][**lbc**][**f**]] | [**–f**]] [file]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Tail copies the named file to the standard output beginning at a designated place. If no file is named, the standard input is used.

Copying begins at distance *+number* from the beginning, or *–number* from the end of the input (if *number* is null, the value 10 is assumed). *Number* is counted in units of lines, blocks, or characters, according to the appended option **l**, **b**, or **c**. When no units are specified, counting is by lines.

With the **–f** ("follow") option, if the input file is not a pipe, the program will not terminate after the line of the input file has been copied, but will enter an endless loop, wherein it sleeps for a second and then attempts to read and copy further records from the input file. Thus it may be used to monitor the growth of a file that is being written by some other process.

EXAMPLES

Tail accepts at most two arguments: the first consists of specified options, and the second specifies the file of interest. If the *number* and **f** options are both desired, they must be concatenated to create a single option argument, as follows:

```
tail –3lf john
```

This example prints the last three lines in the file *john* to the standard output, and leaves *tail* in "follow" mode.

If only the **f** option is desired, it must be preceded by a –, as follows:

```
tail –f fred
```

This example prints the last ten lines of the file *fred*, followed by any lines that are appended to *fred* between the time *tail* is initiated and killed. Note that this output may build up very quickly for rapidly changing input files, perhaps too fast to read on a CRT.

SEE ALSO

dd(1).

BUGS

Tails relative to the end of the file are stored in a buffer, and thus are limited in length. Thus, be wary of the results when piping output from other commands into *tail*.

Various kinds of anomalous behavior may happen with character special files.

Tail can pick up a maximum of 4K bytes of data from the specified file.

NAME

tar – tape file archiver

SYNOPSIS

tar [key] [files]

HP-UX COMPATIBILITY

Level: HP-UX/DEVELOPMENT

Origin: System III and UCB

DESCRIPTION

Tar saves and restores files on magnetic tape or flexible disc. Its actions are controlled by the *key* argument. The *key* is a string of characters containing at most one function letter and possibly one or more function modifiers. The *key* string may be preceded by a dash (–) (similar to the way options are specified in other HP-UX commands), but it is not necessary. Other arguments to the command are *files* (or directory names) specifying which files are to be dumped or restored. In all cases, appearance of a directory name refers to the files and (recursively) subdirectories of that directory.

The function portion of the *key* is specified by one of the following letters:

c Create a new archive; writing begins at the beginning of the archive.

r The named *files* are added to the end of the archive.

u The named *files* are added to the archive if they are not already there, or have been modified since last written on that archive.

x The named *files* are extracted from the archive. If a named file matches a directory whose contents had been written onto the archive, this directory is (recursively) extracted. If no *files* argument is given, the entire content of the archive is extracted. Note that if several files with the same name are on the archive, the last one overwrites all earlier ones. The file and directory ownership written on the archive is only restored for the super-user.

t Produces a listing of the names of files on the archive. Adding the **v** option will expand this listing to include the file modes and owner numbers.

The following function modifiers may be used in addition to the function letters listed above:

0,...,8 This modifier selects the drive on which the 9 track tape is mounted. The default is **8** (as /dev/rmt8).

v Normally, *tar* does its work silently. The **v** (verbose) option causes it to type the name of each file it treats, preceded by the function letter.

w causes *tar* to print the action to be taken, followed by the name of the file, and then wait for the user's confirmation. If a word beginning with **y** is given, the action is performed. Any other input means "no".

f causes *tar* to use the next argument as the name of the archive instead of **/dev/rmt?**. If the name of the file is –, *tar* writes to the standard output or reads from the standard input, whichever is appropriate. Thus, *tar* can be used as the head or tail of a pipeline. *Tar* can also be used to move hierarchies with the command:

```
cd fromdir; tar cf – . | (cd todir; tar xf –)
```

b causes *tar* to use the next argument as the blocking factor for archive records. This option should only be used when a blocking factor of less than 20 is desired. The default is 20 and the minimum is 1 (512 bytes). The blocking factor is determined automatically when reading 9 track tapes. The blocking factor must be specified when reading flexible discs and cartridge tapes if they were written with a blocking factor different than 20.

l tells *tar* to complain if it cannot resolve all of the links to the files being dumped. If **l** is not specified, no error messages are printed.

m tells *tar* to not restore the modification time written on the archive. The modification time of the file will be the time of extraction.

o This option will suppress the directory information written to the archive. Former versions of tar could not handle this information on input.

p This option causes files to be restored to the original modes written on the archive. *Setuid*

and sticky information will also be restored for the super-user.

If a 9 track tape drive is used as the output device, it must be configured in Berkeley compatability mode; see *mt*(4).

EXAMPLE

```
tar cvf /dev/rfd.0 file1 file2
```

This example creates a new archive on /dev/rfd.0 and copies file1 and file2 onto it, using a blocking factor of 20. The *key* is made up of one function letter (**c**) and two function modifiers (**v**, and **f**).

FILES

/dev/rmt?
/dev/rfd.*
/tmp/tar*

SEE ALSO

ar(1), cpio(1), mt(4).

DIAGNOSTICS

Complains about bad key characters and archive read/write errors.
Complains if enough memory is not available to hold the link tables.

BUGS

There is no way to ask for the *n*-th occurrence of a file.
Tape errors are handled ungracefully.
The **u** option can be slow.
If the archive is on a flexible disc or cartridge tape, and if the blocking factor specified on output was not 20, the same blocking factor must be specified on input. This is because the blocking factor is not explicitly stored on the archive.
The current limit on file-name length is 100 characters.
Some previous versions of tar have claimed to support selective listing of file names using the **t** option with a list. To our knowledge this was an error in the documentation and does not appear in the original source code.
There is no way to restore an absolute path name to a relative position.
The arguments required by the **f** and **b** modifiers must be specified in the same order in which the modifiers are specified in the *key*.
Tar always pads information written to an archive up to the next multiple of the block size. Therefore, if you are creating a small archive and write out one block of information, *tar* reports that one block was written, but the actual size of the archive is (for the default case) 20 blocks.

NAME

tbl – format tables for nroff or troff

SYNOPSIS

tbl [–**TX**] [files]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Tbl is a preprocessor that formats tables for *nroff*(1) or *troff*(1). The input files are copied to the standard output, except for lines between **.TS** and **.TE** command lines, which are assumed to describe tables and are re-formatted by *tbl*. (The **.TS** and **.TE** command lines are not altered by *tbl*).

.TS is followed by global options. The available global options are:

center	center the table (default is left-adjust);
expand	make the table as wide as the current line length;
box	enclose the table in a box;
doublebox	enclose the table in a double box;
allbox	enclose each item of the table in a box;
tab (*x*)	use the character *x* instead of a tab to separate items in a line of input data.

The global options, if any, are terminated with a semi-colon (;).

Next come lines describing the format of each line of the table. Each such format line describes one line of the actual table, except that the last format line (which must end with a period) describes *all* remaining lines of the table. Each column of each line of the table is described by a single key-letter, optionally followed by specifiers that determine the font and point size of the corresponding item, that indicate where vertical bars are to appear between columns, that determine column width, inter-column spacing, etc. The available key-letters are:

c	center item within the column;
r	right-adjust item within the column;
l	left-adjust item within the column;
n	numerically adjust item in the column: units positions of numbers are aligned vertically;
s	span previous item on the left into this column;
a	center longest line in this column and then left-adjust all other lines in this column with respect to that centered line;
^	span down previous entry in this column;
_	replace this entry with a horizontal line;
=	replace this entry with a double horizontal line.

The characters **B** and **I** stand for the bold and italic fonts, respectively. The vertical bar character indicates a vertical line between columns.

The format lines are followed by lines containing the actual data for the table, followed finally by **.TE**. Within such data lines, data items are normally separated by tab characters.

If a data line consists of only _ or =, a single or double line, respectively, is drawn across the table at that point; if a *single item* in a data line consists of only _ or =, then that item is replaced by a single or double line.

The –**TX** option forces *tbl* to use only full vertical line motions, making the output more suitable for devices that cannot generate partial vertical line motions (e.g., line printers).

If no file names are given as arguments, *tbl* reads the standard input, so it may be used as a filter. When it is used with *eqn*(1) or *neqn*(1), *tbl* should come first to minimize the volume of data passed through pipes.

HARDWARE DEPENDENCIES

Series 200/500:

Troff and *eqn* are not currently supported.

SEE ALSO

mm(1), nroff(1), mm(7).

TBL – A Program to Format Tables in *HP-UX Concepts and Tutorials.*

BUGS

See *BUGS* under *nroff*(1).

NAME

tcio – Command Set 80 Cartridge Tape Utility

SYNOPSIS

/lbin/tcio –o [**dervSVC**] [buffersize] filename
/lbin/tcio –i [**drvS**] [buffersize] filename
/lbin/tcio –u [**cmrvV**] [blocknumber] [**save** | **restore**] filename [disc_filename]

HP-UX COMPATIBILITY

Level: HP-UX/NON-STANDARD

Origin: HP

DESCRIPTION

Tcio –o (copy out) reads the standard input and writes the data to the raw Command Set 80 Cartridge Tape Unit specified by *filename*.

Tcio –i (copy in) reads the Command Set 80 Cartridge Tape Unit specified by *filename* in raw mode and writes the data to the standard output.

Tcio –u (utility) performs utility functions on the cartridge tape, such as image backup and restore, release, mark, and/or verify cartridge. **(If the system's I/O processor board's revision number is 3.0 or less, this option will lock up the system.)**

In all cases, *filename* MUST refer to a character special file associated with a Command Set 80 cartridge tape unit.

With the output and input operations, *tcio* utilizes a large buffer to transfer data to/from the cartridge tape, yielding a significant increase in performance, as well as a savings in wear and tear on the media and the mechanism. In addition, *tcio* puts a tape mark in the first block on each tape to prevent the tape from being image restored over a disc; it also utilizes the last block on each tape to flag whether the tape is the last tape in a multi-tape sequence or not.

With the utility operation, *tcio* provides functions that are unique to cartridge tapes.

One of the options **o**, **i**, or **u** must be specified. The meanings of the available modifiers are:

v Verbose mode; prints information and error messages to *stderr*.

d Prints a checksum to *stderr*. The checksum is a 32-bit unsigned addition of the bytes, providing an extra check of the validity of the tape in addition to tape verification. The value is only reported to the user and is not written on the media; thus, it's left up to the user to manually record and check it. The checksum is valid only for the **i** and **o** operations, and if the same number of bytes are read from the tape as were written to it. This option is independent of the verbose modifier.

e Applies only to the output operation, and causes a tape mark to be written on the nearest 1024-byte boundary following the end of the data. When a tape containing an end-of-data tape mark is read back, the read will terminate upon encountering the tape mark. Thus, with the use of this option, the checksums generated by the input and output operations are guaranteed to agree.

S Enables specification of buffer size. This option forces the allocation of a block of memory to be used in reading or writing the tape. The size in bytes of the buffer is 1024 times the value specified for *buffersize*. A *buffersize* less than 32 or greater than 512 will cause the program to terminate. If *buffersize* is not specified, *tcio* will attempt to allocate buffer sizes in powers of 2 from 512 down to 64, taking the largest one possible. The primary uses of this option are to allow buffer sizes smaller than 64 Kbytes, and to allow the user to pick a buffer size that is most suitable for his application.

V This option turns off tape verification. It is suggested that this option not be used, for the sake of the integrity of the data output to tape.

m This option writes a tape mark on a tape at the specified block. If a tape is created by some other means than *tcio*, a tape mark in block 0 of the tape will prevent it from being image restored to a disc. Note that *blocknumber* must be specified.

r Releases the tape from the mechanism, unlocking the door.

c Image copy option. Provides the same capability as the push-button save and restore available in the HP 79XX single controller drive. The **save** and **restore** keywords are the same as the labels on those switches. **Save** implies disc to tape; **restore** implies tape to disc. Currently only single controller disc/tape units can be backed up in this way.

C Check read option. Provides a measure of data security not found in the tape verification or check digit options. Check read requires two I/O buffers of the size indicated by *buffersize*, one for writing and one for reading. The data in the first buffer is written to the tape. Then the tape is backspaced and read into the second buffer. The two buffers are then compared. If a difference occurs, *tcio* reports the error and terminates. This option forces no tape verification. Note that this option promotes wear and tear on both the media and the drive, and should only be used when complete assurance of the data's integrity is required.

HARDWARE DEPENDENCIES

Series 200:

The **C** option is not supported.

The **c** option is not supported (thus, the WARNING below does not apply to the Series 200). The separate stand-alone utility *CS80 Tape Backup* provides this functionality.

Due to I/O software architecture, a *buffersize* greater than 64 would provide no increase in performance, but would merely tie-up system memory. Thus, the default for *buffersize* is 64, and if *buffersize* is specified greater than 64, it is silently truncated back to 64.

Series 500:

In general, tapes which have any tape marks other than in the first or the last block cannot be read successfully.

The **e** option is not supported, and because of the above restriction, tapes which have been written under the **e** option cannot be read successfully.

EXAMPLES

The first example below copies the contents of a directory into an archive; the second restores it:

```
ls | cpio -o | tcio -o /dev/rct
tcio -i /dev/rct | cpio -i
```

SEE ALSO

cpio(1).

WARNING

To be able to use the save/restore facility, the following two conditions must be met:

your system must be in single-user mode;

you must never have used networking on your system. If networking has been used on your system, you must reboot the system before using the save/restore facility.

Tcio can tie up substantial portions of memory, creating a situation where progress on other processes (including those processes feeding *tcio*) is hindered. If this should occur, it is best to kill *tcio* and re-execute using a smaller *buffersize*. This problem is especially acute when using the **C** option, because two buffers are required.

BUGS

If the cartridge drive cannot read the manufacturer's block on the tape, the cartridge is locked in the drive and cannot be extracted without turning off the disc/tape drive. This failure is usually the result of faulty tapes or a dirty drive mechanism.

NAME

tee – pipe fitting

SYNOPSIS

tee [–**i**] [–**a**] [file] ...

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

Tee transcribes the standard input to the standard output and makes copies in the *files*. The –**i** option ignores interrupts; the –**a** option causes the output to be appended to the *files* rather than overwriting them.

NAME

test – condition evaluation command

SYNOPSIS

test *expr*
[expr]

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

Test evaluates the expression *expr* and, if its value is true, returns a zero (true) exit status; otherwise, a non-zero (false) exit status is returned; *test* also returns a non-zero exit status if there are no arguments. The following primitives are used to construct *expr*:

–**r** *file*	true if *file* exists and is readable.
–**w** *file*	true if *file* exists and is writable.
–**x** *file*	true if *file* exists and is executable.
–**f** *file*	true if *file* exists and is a regular file.
–**d** *file*	true if *file* exists and is a directory.
–**c** *file*	true if *file* exists and is a character special file.
–**b** *file*	true if *file* exists and is a block special file.
–**u** *file*	true if *file* exists and its set-user-ID bit is set.
–**g** *file*	true if *file* exists and its set-group-ID bit is set.
–**k** *file*	true if *file* exists and its sticky bit is set.
–**s** *file*	true if *file* exists and has a size greater than zero.
–**t** [*fildes*]	true if the open file whose file descriptor number is *fildes* (1 by default) is associated with a terminal device.
–**z** *s1*	true if the length of string *s1* is zero.
–**n** *s1*	true if the length of the string *s1* is non-zero.
s1 = *s2*	true if strings *s1* and *s2* are identical.
s1 != *s2*	true if strings *s1* and *s2* are *not* identical.
s1	true if *s1* is *not* the null string.
n1 –**eq** *n2*	true if the integers *n1* and *n2* are algebraically equal. Any of the comparisons –**ne**, –**gt**, –**ge**, –**lt**, and –**le** may be used in place of –**eq**.

These primaries may be combined with the following operators:

!	unary negation operator.
–**a**	binary *and* operator.
–**o**	binary *or* operator (–**a** has higher precedence than –**o**).
(expr)	parentheses for grouping.

Notice that all the operators and flags are separate arguments to *test*. Notice also that parentheses are meaningful to the shell and, therefore, must be escaped.

Test is directly interpreted by the shell.

SEE ALSO

eval (see *sh*(1)), expr(1), find(1), sh(1).

WARNING

In the second form of the command (i.e., the one that uses **[ ]**, rather than the word *test*), the square brackets must be ***delimited*** by blanks.

NAME

time – time a command

SYNOPSIS

time command

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

The given command is executed; after it is complete, *time* prints the elapsed time during the command, the time spent in the system, and the time spent in execution of the command. Times are reported in seconds.

The execution time can depend on the performance of the memory in which the program is running. Also, note that I/O-intensive programs will show an elapsed time which is considerably more than the CPU time, since only CPU time is monitored - not I/O time.

The times are printed on standard error.

HARDWARE DEPENDENCIES

Series 500:

For those computers with multiple CPU's, the child CPU times listed may be greater than the actual real elapsed time, since CPU time is counted on a per-CPU basis. Thus, if three CPUs are executing, the times listed are obtained by adding the execution times of each CPU.

SEE ALSO

times command in sh(1), times(2).

NAME

touch – update access/modification/change times of file

SYNOPSIS

touch [–**amc**] [mmddhhmm[yy]] files

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

Touch causes the access, modification, and last change times of each argument to be updated. If no time is specified (see *date*(1)) the current time is used. The –**a** and –**m** options cause touch to update only the access or modification times respectively (default is –**am**). The –**c** option silently prevents *touch* from creating the file if it did not previously exist.

The return code from *touch* is the number of files for which the times could not be successfully modified (including files that did not exist and were not created).

SEE ALSO

date(1), utime(2).

NAME

tr – translate characters

SYNOPSIS

tr [–**cds**] [string1 [string2]]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Tr copies the standard input to the standard output with substitution or deletion of selected characters. Input characters found in *string1* are mapped into the corresponding characters of *string2*. Any combination of the options –**cds** may be used:

–**c** Complements the set of characters in *string1* with respect to the universe of characters whose ASCII codes are 001 through 377 octal.

–**d** Deletes all input characters in *string1*.

–**s** Squeezes all strings of repeated output characters that are in *string2* to single characters.

The following abbreviation conventions may be used to introduce ranges of characters or repeated characters into the strings:

[a–z] Stands for the string of characters whose ASCII codes run from character **a** to character **z**, inclusive.

[a*n] Stands for *n* repetitions of **a**. If the first digit of *n* is **0**, *n* is considered octal; otherwise, *n* is taken to be decimal. A zero or missing *n* is taken to be huge; this facility is useful for padding *string2*.

The escape character \ may be used as in the shell to remove special meaning from any character in a string. In addition, \ followed by 1, 2, or 3 octal digits stands for the character whose ASCII code is given by those digits.

EXAMPLE

The following creates a list of all the words in *file1* one per line in *file2*, where a word is taken to be a maximal string of alphabetics. The strings are quoted to protect the special characters from interpretation by the shell; 012 is the ASCII code for newline.

```
tr –cs "[A–Z][a–z]" "[\012*]" <file1 >file2
```

SEE ALSO

ed(1), sh(1).

BUGS

Won't handle ASCII **NUL** in *string1* or *string2*; always deletes **NUL** from input.

NAME

true, false – provide truth values

SYNOPSIS

true

false

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

True does nothing, successfully. *False* does nothing, unsuccessfully. They are typically used in input to *sh*(1) such as:

```
while true do
        command
done
```

SEE ALSO

sh(1).

DIAGNOSTICS

True has exit status zero, *false* nonzero.

NAME

tset – terminal dependent initialization

SYNOPSIS

tset [options] [**–m** [ident] [test baudrate] :type] ... [type]

reset ...

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: UCB

DESCRIPTION

Tset sets up your terminal when you first log in to an HP-UX system. It does terminal dependent processing, such as setting erase and kill characters, setting or resetting delays, and sending any sequences needed to properly initialize the terminal. It first determines the *type* of terminal involved, and then does the necessary initializations and mode settings. The type of terminal attached to each HP-UX port is specified in the */etc/ttytype* data base. Type names for terminals may be found in the *termcap*(5) data base. If a port is not wired permanently to a specific terminal (not hardwired), it will be given an appropriate generic identifier, such as *dialup*.

In the case where no arguments are specified, *tset* simply reads the terminal type out of the environment variable TERM and re-initializes the terminal. The rest of this manual entry concerns itself with mode and environment initialization, typically done once at login, and options used at initialization time to determine the terminal type and set up terminal modes.

When used in a startup script (*.profile*), it is desirable to give information about the type of terminal you will usually use on ports which are not hardwired. These ports are identified in */etc/ttytype* as *dialup* or *plugboard*, etc. To specify what terminal type you usually use on these ports, the **–m** (map) option flag is followed by the appropriate port type identifier, an optional baud rate specification, and the terminal type. (The effect is to "map" from some conditions to a terminal type, that is, to tell *tset*, "If I'm on this kind of port, then I'll probably be on this kind of terminal".) If more than one mapping is specified, the first applicable mapping prevails. A missing port type identifier matches all identifiers. A *baudrate* is specified as with *stty*(1), and is compared with the speed of the diagnostic output (which should be the control terminal). The baud rate *test* may be any combination of >, @, <, and !. @ means "at" and ! inverts the sense of the test. To avoid problems with metacharacters, it is best to place the entire argument to **–m** within single quotes.

Thus,

```
tset –m 'dialup>300:2622' –m 'dialup:2624' –m 'plugboard:?2623'
```

causes the terminal type to be set to an HP 2622 if the port in use is a dialup at a speed greater than 300 baud, or to an HP 2624 if the port is otherwise a dialup (i.e. at 300 baud or less). If the *type* finally determined by *tset* begins with a question mark, the user is asked if he or she really wants that type. A null response means to use that type; otherwise, another type can be entered. Thus, in the above case, if the user is on a plugboard port, he or she will be asked whether or not he or she is actually using an HP 2623.

If no mapping applies and a final *type* option, not preceded by a **–m**, is given on the command line, then that type is used. Otherwise, the identifier found in the */etc/ttytype* data base will be taken to be the terminal type. The latter should always be the case for hardwired ports.

It is usually desirable to return the terminal type, as finally determined by *tset*, and information about the terminal's capabilities to a shell's environment. This can be done using the **–s** option. Using the Bourne shell (*sh*(1)), the command

```
export TERM; TERM=`tset –s options...
```

or

eval `tset –s *options...*

causes *tset* to place the name of your terminal in the variable TERM in the environment.

Once the terminal type is known, *tset* engages in terminal driver mode setting. This normally involves sending an initialization sequence to the terminal, setting the single character erase (and optionally the full line erase or line-kill) characters, and setting special character delays. Tab and new-line expansion are turned off during transmission of the terminal initialization sequence.

On terminals that can backspace but not overstrike (such as a CRT), and when the erase character is the default erase character ("#" on standard systems), the erase character is changed to BACKSPACE (^H).

The options are:

–e*c* sets the erase character to be the named character *c*; *c* defaults to ^H (BACKSPACE). The character *c* can either be typed directly, or entered using the "hat" notation used here (e.g. the "hat" notation for control-H is ^H).

–k*c* sets the kill character to *c*. The default *c* is ^X. If *c* is not specified, the kill character will remain unchanged unless the original value of the kill character is null. In this case, the kill character is set to an "at" sign (@).

–s outputs *setenv* commands for TERM. This can be used with `tset –s ...`, and is preferred to "setenv TERM `tset – ...`", because **–s** sets the TERMCAP variable also.

–I suppresses *transmitting* terminal initialization strings.

–Q suppresses printing the "Erase set to" and "Kill set to" messages.

–A asks the user for the TERM type.

–S outputs two strings suitable for use in *.profile* files as follows:

set noglob set term=(`tset –S ...`) set TERM $term[1]
set TERMCAP "$term[2]" unset term unset noglob

–u do not update */etc/htmp*.

–h forces a read of */etc/ttytype*. When **–h** is not specified, the termnal type is determined by reading */etc/htmp* or the environment, unless some mapping is specified. This is useful when */etc/htmp* is not correct.

For compatibility with earlier versions of *tset*, the following flags are accepted, but their use is discouraged:

– report terminal type. Whatever type is decided on is reported. If no other flags are stated, the only effect is to write the terminal type on the standard output.

–r report to the user in addition to other flags.

–E*c* sets the erase character to *c* only if the terminal can backspace. *C* defaults to ^H.

EXAMPLES

These examples all assume the Bourne shell (*sh*(1)) and use the – option. Note that a typical use of *tset* in a *.profile* will also use the **–e** and **–k** options, and often the **–n** or **–Q** options as well. These options have not been included here to keep the examples small.

At the moment, you are on an HP 2622. This is suitable for typing by hand but not for a *.profile*, unless you are *always* on a 2622.

export TERM; TERM=`tset – 2622`

You have an HP 2623 at home which you dial up on, but your office terminal is hardwired and known in */etc/ttytype*.

```
export TERM; TERM=`tset - -m dialup:2623`
```

You have a switch which connects everything to everything, making it nearly impossible to key on what port you are coming in on. You use an HP 2622 in your office at 9600 baud, and dial up to switch ports at 1200 baud from home on an HP 2623. Sometimes you use someone else's terminal at work, so you want it to ask you to make sure what terminal type you have at high speeds, but at 1200 baud you are always on a 2623. Note the placement of the question mark, and the quotes to protect the > and ? from interpretation by the shell.

```
export TERM; TERM=`tset - -m 'switch>1200:?2622' -m 'switch<=1200:2623'`
```

All of the above entries will fall back on the terminal type specified in */etc/ttytype* if none of the conditions hold. The following entry is appropriate if you always dial up, always at the same baud rate, on many different kinds of terminals. Your most common terminal is an HP 2622. It always asks you what kind of terminal you are on, defaulting to 2622.

```
export TERM; TERM=`tset - ?2622`
```

If the file */etc/ttytype* is not properly installed and you want to key entirely on the baud rate, the following can be used:

```
export TERM; TERM=`tset - -m '>1200:2624' 2622`
```

FILES

/etc/ttytype	port name to terminal type mapping data base;
/etc/termcap	terminal capability data base.

SEE ALSO

sh(1), stty(1), ttytype(5), termcap(5), environ(7).

NAME

tsort – topological sort

SYNOPSIS

tsort [file]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Tsort produces on the standard output a totally ordered list of items consistent with a partial ordering of items mentioned in the input *file*. If no *file* is specified, the standard input is understood.

The input consists of pairs of items (nonempty strings) separated by blanks. Pairs of different items indicate ordering. Pairs of identical items indicate presence, but not ordering.

SEE ALSO

lorder(1).

DIAGNOSTICS

Odd data: there is an odd number of fields in the input file.

BUGS

Uses a quadratic algorithm; not worth fixing for the typical use of ordering a library archive file.

NAME

tty – get the terminal's name

SYNOPSIS

tty [**–s**]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Tty prints the path name of the user's terminal. The **–s** option inhibits printing, allowing one to test just the exit code.

RETURN VALUE

0 if standard input is a terminal,
1 otherwise.

DIAGNOSTICS

"not a tty" if the standard input is not a terminal and **–s** is not specified.

NAME

ul – do underlining

SYNOPSIS

ul [**–i**] [**–t** terminal] [name ...]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: UCB

DESCRIPTION

Ul reads the named files (or standard input if none are given) and translates occurences of underscores to the sequence which indicates underlining for the terminal in use, as specified by the environment variable **TERM**. The **–t** option overrides the terminal kind specified in the environment. The file */etc/termcap* is read to determine the appropriate sequences for underlining. If the terminal is incapable of underlining, but is capable of a standout mode then that is used instead. If the terminal can overstrike, or handles underlining automatically, *ul* degenerates to *cat*(1). If the terminal cannot underline, underlining is ignored.

The **–i** option causes *ul* to indicate underlining onto by a separate line containing appropriate dashes '–'; this is useful when you want to look at the underlining which is present in an *nroff* output stream on a crt-terminal.

SEE ALSO

man(1), nroff(1).

BUGS

Nroff usually outputs a series of backspaces and underlines intermixed with the text to indicate underlining. No attempt is made to optimize the backward motion.

NAME

umask – set file-creation mode mask

SYNOPSIS

umask [ooo]

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

The user file-creation mode mask is set to *ooo*. The octal three digits refer to read/write/execute permissions for *owner*, *group*, and *others*, respectively (see *chmod*(2) and *umask*(2)). The value of each specified digit is subtracted from the corresponding "digit" specified by the system for the creation of a file (see *creat*(2)). For example, **umask 022** removes *group* and *others* write permission (files normally created with mode **777** become mode **755**; files created with mode **666** become mode **644**).

If *ooo* is omitted, the current value of the mask is printed with four octal digits. The first digit, a zero, specifies that the output is expressed in octal.

Umask is recognized and executed by the shell.

Note that the file creation mask does not affect the set-user-ID, set-group-ID, or "sticky" bits.

SEE ALSO

chmod(1), sh(1), chmod(2), creat(2), umask(2).

NAME

uname – print name of current HP-UX version

SYNOPSIS

uname [**–snrvmia**]

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

Uname prints the current system name of HP-UX on the standard output file. It is mainly useful to determine what system you are using. The options cause selected information returned by *uname*(2) to be printed:

–s print the system name (default).

–n print the nodename (the nodename is a name that the system is known by on a communications network). (e.g. *uucp*).

–r print the operating system release.

–v print the operating system version.

–m print the machine hardware name.

–i print the machine identification number.

–a print all the above information.

SEE ALSO

hostname(1), gethostname(2), sethostname(2), uname(2).

NAME

unget – undo a previous get of an SCCS file

SYNOPSIS

unget [–r*SID*] [–**s**] [–**n**] files

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Unget undoes the effect of a **get –e** done prior to creating the intended new delta. If a directory is named, ***unget*** behaves as though each file in the directory were specified as a named file, except that non-SCCS files and unreadable files are silently ignored. If a name of – is given, the standard input is read with each line being taken as the name of an SCCS file to be processed. Refer to *sact*(1), which describes how to determine what deltas are currently binding for an s-file.

Keyletter arguments apply independently to each named file.

–r*SID* Uniquely identifies which delta is no longer intended. (This would have been specified by *get* as the "new delta"). The use of this keyletter is necessary only if two or more outstanding *get*s for editing on the same SCCS file were done by the same person (login name). A diagnostic results if the specified *SID* is ambiguous, or if it is necessary and omitted on the command line (see *sact*(1)).

–s Suppresses the printout, on the standard output, of the intended delta's *SID*.

–n Causes the retention of the gotten file which would normally be removed from the current directory.

Note: ***unget*** can only be executed by the user who did the corresponding **get –e**. If a system administrator needs to ***unget*** a **get –e** done by another user, he must either use *su*(1) to change into that user, or edit the p-file directly (which can be done either by the s-file owner of the super-user).

FILES

p-file see *delta*(1).

g-file see *delta*(1).

SEE ALSO

delta(1), get(1), sact(1).

SCCS User's Guide in *HP-UX Concepts and Tutorials.*

DIAGNOSTICS

Use ***help***(1) for explanations.

NAME

uniq – report repeated lines in a file

SYNOPSIS

uniq [–**udc** [+n] [–n]] [input [output]]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Uniq reads the input file and compares adjacent lines. In the normal case, the second and succeeding copies of repeated lines are removed; the remainder is written on the output file. *Input* and *output* should always be different. Note that repeated lines must be adjacent in order to be found; see *sort*(1).

The options are:

–u causes only the lines that are not repeated in the original file to be output.

–d specifies that one copy of just the repeated lines is to be written.

–c supersedes **–u** and **–d** and generates an output report in default style but with each line preceded by a count of the number of times it occurred.

–n the first ***n*** fields, together with any blanks before each, are ignored. A field is defined as a string of non-space, non-tab characters separated by tabs and spaces from its neighbors.

+n The first ***n*** characters are ignored. Fields are skipped before characters.

The normal mode output is the union of the **–u** and **–d** mode outputs.

SEE ALSO

comm(1), sort(1).

NAME

units – conversion program

SYNOPSIS

units

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Units converts quantities expressed in various standard scales to their equivalents in other scales. It works interactively in this fashion:

```
You have: inch
You want: cm
        * 2.540000e+00
        / 3.937008e-01
```

A quantity is specified as a multiplicative combination of units optionally preceded by a numeric multiplier. Powers are indicated by suffixed positive integers, and division by the usual sign:

```
You have: 15 lbs force/in2
You want: atm
        * 1.020689e+00
        / 9.797299e-01
```

Units only does multiplicative scale changes; thus it can convert Kelvin to Rankine, but not Celsius to Fahrenheit. Most familiar units, abbreviations, and metric prefixes are recognized, together with a generous leavening of exotica and a few constants of nature including:

pi	ratio of circumference to diameter
c	speed of light
e	charge on an electron
g	acceleration of gravity
force	same as **g**
mole	Avogadro's number
water	pressure head per unit height of water
au	astronomical unit.

Pound is not recognized as a unit of mass; **lb** is. Compound names are run together, (e.g., **lightyear**). British units that differ from their U.S. counterparts are prefixed thus: **brgallon**. For a complete list of units, type:

```
cat /usr/lib/unittab
```

FILES

/usr/lib/unittab

NAME

upm – unpack cpio archives from HP media

SYNOPSIS

upm –E [**cdmtuvx**] pathname chardevice [patterns]

upm –iM [**cdmtuvx**] [patterns] </dev/rmf?

HP-UX COMPATIBILITY

Level: HP-UX/EXTENDED

Origin: HP

Remarks: *Upm* is implemented on the Series 500 only.

DESCRIPTION

Upm is similar to *cpio*(1), and is included to enable you to restore files from 88140L/S tape cartridges or 5.25-inch flexible discs more efficiently.

Upm –E (copy in from tape cartridge) extracts all files specified by *patterns* from the file named by *pathname* (assumed to be the product of a previous *cpio –o*). *Patterns* is a series of zero or more blank-separated character strings given in the name-generating notation of *sh*(1). Note that the metacharacters **?**, *****, and **[...]** match the slash (/) when used in *patterns*. The default *pattern* is '*****', which selects all files. *Chardevice* identifies the character special device file describing the volume containing *pathname*. (Note that, if this volume is not the root, it must be mounted at the time *upm* is used, and *pathname* must include the directory name on which the volume is mounted.)

Upm –iM (copy in) extracts all files selected by zero or more of the specified *patterns* (see above for a description of *patterns*). The files are extracted from the standard input, which is redirected from a raw miniature flexible disc device /dev/rmf?. The resulting standard input is assumed to be the product of a previous *cpio –o*.

Any other options specified must be concatenated with the initial *E* or *iM* options. The options have the following meanings:

- **c** read header information which was previously written in ASCII character form for portability;
- **d** directories are to be created as needed;
- **m** retain previous file modification time. This option is ineffective on directories that are being copied;
- **t** print a table of contents of the input; no files are created;
- **u** copy unconditionally (normally, an older file will not replace a newer file with the same name);
- **v** verbose; causes a list of file names to be printed. When used with the **t** option, the table of contents looks like the output of an *ls –l* command (see *ls*(1));
- **x** restore device special files; *mknod*(2) is used to recreate these files, and thus **–Ex** or **–iMx** can only be used by the super-user. Restoring device files onto a different system can be very dangerous. This is intended for backup use;

When the end of a volume is reached, *upm* will prompt the user for the next flexible disc and continue.

The number of blocks reported by *upm* is always in units of 512-byte blocks, regardless of the block size of the initialized media.

SEE ALSO

cpio(1), tcio(1), mknod(2).

WARNING

The **–B** option must not be used when performing raw I/O using the HP 9130K miniature flexible disc drive.

BUGS

Only the super-user can copy special files.

If /dev/tty is not accessible, *upm* issues a complaint, or refuses to work.

The **–Edr** and **–iMdr** options will not make empty directories.

NAME

uucico – uucp copy in and copy out

SYNOPSIS

/usr/lib/uucp/uucico [**–r1**] [–ssys] [–xnum]

HP-UX COMPATIBILITY

Level: Data Communications – HP-UX/STANDARD

Origin: System III

DESCRIPTION

Uucico scans the */usr/spool* directory for work files. If such files exist, a connection to a remove system is attempted using the line protocol for the remote system specified in the **L.sys** file. *Uucico* then executes all requests for work and logs the results.

The options are as follows:

–r1 Start *uucico* in the MASTER mode; The default is SLAVE mode.

–s*sys* Do work only for the system specified by *sys*. If there is no work for *sys* on the local spool directory, initiate a connection to *sys* to determine if *sys* has work for the local system.

–x*num* Use debugging option. *Num* is an integer in the range 1 – 9. More debugging information is given for larger values of *num*.

Uucico is usually started by a local program (*cron*(8), *uucp*(1C), *uuc*(1C), *uuxqt*(1C), or *uucico*(1C)). It should *only* be directly initiated by a user when debugging.

When started by a local program, *uucico* is considered the MASTER and attempts a connection to a remote system. If *uucico* is started by a remote system, it is considered to be in SLAVE mode.

For the *uucico* connection to a remote system to be successful, there must be an entry in the */etc/passwd* file on the remote system of the form:

uucp::5:5::/usr/spool/uucppublic:/usr/lib/uucp/uucico

FILES

Refer to *Uucp File System* chapter in the *HP-UX Asynchronous Communications Guide.*

SEE ALSO

HP-UX Asynchronous Communications Guide.

NAME

uucp, uulog, uuname – UNIX to UNIX copy; file transfer

SYNOPSIS

uucp [option] ... source-file ... destination-file

uulog [option] ...

uuname

HP-UX COMPATIBILITY

Level: Data Communications - HP-UX/STANDARD

Origin: System III

DESCRIPTION

Uucp copies files named by the *source-file* arguments to the *destination-file* argument. A file name may be a path name on your machine, or may have the form:

system-name!path-name

where *system-name* is taken from a list of system names in the **L.sys** file which *uucp* knows about. Shell metacharacters **?*[]** appearing in *path-name* will be expanded on the appropriate system.

Path names may be one of:

(1) a full path name;

(2) a path name preceded by *~user* where *user* is a login name on the specified system and is replaced by that user's login directory;

(3) a path name preceded by *~/user* where *user* is a login name on the specified system and is replaced by that user's directory under PUBDIR (see FILES);

(4) anything else is prefixed by the current directory.

The local and remote system access to the path name is specified in the USERFILE. If the result is an erroneous path name for the remote system the copy will fail. If the *destination-file* is a directory, the last part of the *source-file* name is used. The accessibility of the file or path name is specified in USERFILE.

Uucp preserves execute permissions across the transmission and gives 0666 read and write permissions (see *chmod*(2)).

The following options are interpreted by *uucp*:

–d Make all necessary directories for the file copy (default).

–f Do not make intermediate directories for the file copy.

–c Use the source file when copying out rather than copying the file to the spool directory (default).

–C Copy the source file to the spool directory immediately and use the copy.

–m Send mail to the requester when the copy is complete.

–n*user* Notify *user* on the remote system that a file was sent.

–e*sys* Send the *uucp* command to system *sys* to be executed there. (Note – this will only be successful if the remote machine allows the *uucp* command to be executed by **/usr/lib/uucp/uuxqt**.)

–g*grade* Request *grade* as a priority for the work sequencing. Grades are specified in the order A – Z, a – z, with **A** specifying that the work should be done first, and **z** specifying that the work should be done last. All other grades specify a sequence somewhere in between. The default is **n**.

–r create the necessary files for the transfer to take place, but do not attempt to call the remote system (i.e. suppress calling *uucico*(1C)). This option is useful for systems which must wait to be called (passive systems).

Uulog maintains a summary log of *uucp* and *uux*(1C) transactions in the file **/usr/spool/uucp/LOGFILE** by gathering information from partial log files named **/usr/spool/uucp/LOG.*.?**. (These files will only be created if the **LOGFILE** is being used by another process when a *uucp*, *uux*, *uucico*, or *uuxqt* process needs to log an entry.) It removes the partial log files only if issued with no parameters.

The options cause *uulog* to print logging information:

–s*sys* Print information about work involving system *sys*.

–u*user* Print information about work done for the specified *user*.

Uuname lists the uucp names of known systems. Duplicate lines are not shown, but blank lines are. The **–l** option returns the local system name.

FILES

/usr/spool/uucp	spool directory
/usr/spool/uucppublic	public directory for receiving and sending (PUBDIR)
/usr/lib/uucp/*	other data and program files

SEE ALSO

mail(1), uux(1C).

HP-UX Asynchronous Communications Guide.

WARNING

The domain of remotely accessible files can be (and for obvious security reasons, usually should be) severely restricted. You will very likely not be able to fetch files by path name; ask a responsible person on the remote system to send them to you. For the same reasons you will probably not be able to send files to arbitrary path names. As distributed, the remotely accessible files are those whose names begin **/usr/spool/uucppublic** (equivalent to ~**uucp** or just ~). Note that, if */etc/passwd* contains a blank line, a null user entry, or an entry for ~**uucp**, then ~ and ~**uucp** will not expand properly. Because of this, the *uuto* script will not send files to the proper directory.

BUGS

All files received by *uucp* will be owned by *uucp*.

The **–m** option will only work sending files or receiving a single file. (Receiving multiple files specified by special shell characters **?*[]** will not activate the **–m** option.)

If *uulog* is issued with no parameters when a *uucp* process is writing to a temporary logfile, some log information (that information written after the **LOG.*** files are unlinked) may be lost.

Uucp, when used to copy files locally, will create the new file with mode 644 instead of 666.

NAME

uustat – uucp status inquiry and job control

SYNOPSIS

uustat [[**–jall** **–v**] | **–jall** | **–m***mch* | **–k***jobn* | **–c***hour*] | [**–u***user* **–s***sys* **–o***hour* **–y***hour* **–v**]

HP-UX COMPATIBILITY

Level: Data Communications - HP-UX/STANDARD

Origin: System III

DESCRIPTION

Uustat will display the status of, or cancel, previously specified *uucp* commands, or provide general status on *uucp* connections to other systems. At most one of the following options may be specified:

–jall Report the status of all *uucp* requests. The **–v** option may appear with **–jall**.

–m*mch* Report the status of accessibility of machine *mch*. If *mch* is specified as **all**, then the status of all machines known to the local *uucp* are provided.

–k*jobn* Kill the *uucp* request whose job number is *jobn*. The killed *uucp* request must belong to the person issuing the *uustat* command unless he is the super-user.

–c*hour* Remove the status entries (i.e. spool files in */usr/spool/uucp*) which are older than *hour* hours. This administrative option can only be initiated by the user **uucp** or the super-user.

If none of the above options are specified, any or all of the following options may appear:

–u*user* Report the status of all *uucp* requests issued by *user*.

–s*sys* Report the status of all *uucp* requests which communicate with remote system *sys*.

–o*hour* Report the status of all *uucp* requests which are older than *hour* hours.

–y*hour* Report the status of all *uucp* requests which are younger than *hour* hours.

–v Report the *uucp* status verbosely. If this option is not specified, a status code is printed with each *uucp* request.

When no options are given, *uustat* outputs the status of all *uucp* requests issued by the current user.

For example, the command

uustat –uhdc –smhtsa –y72 –v

will print the verbose status of all *uucp* requests that were issued by user *hdc* to communicate with system *mhtsa* within the last 72 hours. The meanings of the job request status are:

job-number user remote-system command-time status-time status

where the *status* may be either an octal number or a verbose description. The octal code corresponds to the following description:

OCTAL	STATUS
00001	the copy failed, but the reason cannot be determined
00002	permission to access local file is denied
00004	permission to access remote file is denied
00010	bad *uucp* command is generated
00020	remote system cannot create temporary file
00040	cannot copy to remote directory
00100	cannot copy to local directory
00200	local system cannot create temporary file
00400	cannot execute *uucp*
01000	copy succeeded
02000	copy finished, job deleted
04000	job is queued

The meanings of the machine accessibility status are:

system-name time status

where *time* is the latest status time and *status* is a self-explanatory description of the machine status.

FILES

/usr/spool/uucp	spool directory
/usr/lib/uucp/L_stat	system status file
/usr/lib/uucp/R_stat	request status file

SEE ALSO

uucp(1C).

HP-UX Asynchronous Communications Guide.

NAME

uuto, uupick – public UNIX-to-UNIX file copy

SYNOPSIS

uuto [options] source-files destination
uupick [**–s** system]

HP-UX COMPATIBILITY

Level: Data Communications - HP-UX/STANDARD

Origin: System III

DESCRIPTION

Uuto sends *source-files* to *destination*. *Uuto* uses the *uucp*(1C) facility to send files, while it allows the local system to control the file access. A source-file name is a path name on your machine. Destination has the form:

system!*user*

where *system* is taken from a list of system names that *uucp* knows about (see *uucp*(1C)). *User* is the login name of someone on the specified system.

Two options are available:

–p Copy the source file into the spool directory immediately, and send the copy.
–m Send mail to the requester when the copy is complete.

The files (or sub-trees if directories are specified) are sent to PUBDIR on *system*, where PUBDIR is the *uucp* public directory (/usr/spool/uucppublic). Specifically the files are sent to

PUBDIR/receive/*user*/*mysystem*/files.

The recipient is notified by *mail*(1) of the arrival of files.

Uupick accepts or rejects the files transmitted to the recipient. Specifically, *uupick* searches PUBDIR for files destined for the user. For each entry (file or directory) found, the following message is printed on the standard output:

from *system*: [file *file-name*] [dir *dirname*] **?**

Uupick then reads a line from the standard input to determine the disposition of the file:

<new-line>	Go on to next entry.
d	Delete the entry.
m [*dir*]	Move the entry to named directory *dir* (current directory is default). Note that, if the current working directory is desired for *dir*, you should **not** specify any parameter with **m**. A construction like **m.** is erroneous, and results in loss of data.
a [*dir*]	Same as **m** except move all the files sent from *system*.
p	Print the contents of the file.
q	Stop.
EOT (control-d)	Same as **q**.
!*command*	Escape to the shell to do *command*.
*	Print a command summary.

Uupick invoked with the –s*system* option will only search PUBDIR for files sent from *system*.

FILES

PUBDIR = /usr/spool/uucppublic public directory

SEE ALSO

mail(1), uucp(1C), uustat(1C), uux(1C), uuclean(8).

HP-UX Asynchronous Communications Guide.

NAME

uux – UNIX to UNIX command execution

SYNOPSIS

/usr/bin/uucp/uux [**–**] [**–z**] [**–n**] [**–r**] command-string

HP-UX COMPATIBILITY

Level: Data Communications - HP-UX/STANDARD

Origin: System III

DESCRIPTION

Uux will gather zero or more files from various systems, execute a command on a specified system and then send standard output to a file on a specified system. Note that, for security reasons, many installations will limit the list of commands executable on behalf of an incoming request from *uux*. Many sites will permit little more than the receipt of mail (see *mail*(1)) via *uux*.

The *command-string* is made up of one or more arguments that look like a Shell command line, except that the command and file names may be prefixed by *system-name*!. A null *system-name* is interpreted as the local system.

File names may be one of

(1) a full path name;

(2) a path name preceded by ~*xxx* where *xxx* is a login name on the specified system and is replaced by that user's login directory;

(3) anything else is prefixed by the current directory.

The – option will cause the standard input to the *uux* command to be the standard input to the *command-string*. For example, the command

uux "!diff usg!/usr/dan/f1 pwba!/a4/dan/f1 > !f1.diff"

will get the **f1** files from the "usg" and "pwba" machines, execute a *diff* command and put the results in **f1.diff** in the local directory.

Any special shell characters such as <>; should be quoted either by quoting the entire *command-string*, or quoting the special characters as individual arguments.

Uux will attempt to get all files to the execution system. For files which are output files, the file name must be escaped using parentheses. For example, the command

uux a!uucp b!/usr/file \(c!/usr/file\)

will send a *uucp* command to system "a" to get **/usr/file** from system "b" and send it to system "c".

Uux notifies you of the execution status (success or failure) of all commands except *mail*. The response comes by remote mail from the remote machine.

The options are:

–z reduces the amount of mail notification; the remote system is notified only if the command fails.

–n disables notification completely.

–r create the necessary files for the transfer to take place, but do not attempt to call the remote system (i.e. suppresses calling *uucico*(1C)).

FILES

/usr/bin/uucp/spool	spool directory
/usr/bin/uucp/*	other data and programs

SEE ALSO

uucp(1C), uuclean(8).

HP-UX Asynchronous Communications Guide.

BUGS

Only the first command of a shell pipeline may have a *system-name*!. All other commands are executed on the system of the first command.

The use of the shell metacharacter * will probably not do what you want it to do. The shell tokens << and >> are not implemented.

NAME

uuxqt – uucp command execution

SYNOPSIS

/usr/lib/uucp/uuxqt [**–x**num]

HP-UX COMPATIBILITY

Level: Data Communications – HP-UX/STANDARD

Origin: System III

DESCRIPTION

The *uuxqt* daemon performs local command execution of execution files (**X.***) on the */usr/spool/uucp* directory. *Uux* generates work files with an execution (X) grade which become execution files when transferred to the remote system. The command requested by the execution file is checked against the list of remotely executable commands in the COMMANDS file. The USERFILE is then searched to find the first NULL system field for path access permission.

The option **–x***num* is a parameter specifying debugging information. *Num* is an integer in the range 1 – 9. The amount of debugging information increases as the value of *num* increases.

FILES

Refer to the *Uucp File System* chapter in the *HP-UX Asynchronous Communications Guide.*

SEE ALSO

HP-UX Asynchronous Communications Guide.

NAME

val – validate SCCS file

SYNOPSIS

val –
val [–**s**] [–**r**SID] [–**m**name] [–**y**type] files

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Val determines if the specified *file* is an SCCS file meeting the characteristics specified by the optional argument list. Arguments to *val* may appear in any order. The arguments consist of keyletter arguments, which begin with a –, and named files.

Val has a special argument, –, which causes reading of the standard input until an end-of-file condition is detected. Each line read is independently processed as if it were a command line argument list.

Val generates diagnostic messages on the standard output for each command line and file processed and also returns a single 8-bit code upon exit as described below.

The keyletter arguments are defined as follows. The effects of any keyletter argument apply independently to each named file on the command line.

–s The presence of this argument silences the diagnostic message normally generated on the standard output for any error that is detected while processing each named file on a given command line.

–r*SID* The argument value *SID* (*S*CCS *ID*entification String) is an SCCS delta number. A check is made to determine if the *SID* is ambiguous (e. g., **r**1 is ambiguous because it physically does not exist but implies 1.1, 1.2, etc. which may exist) or invalid (e. g., **r**1.0 and **r**1.1.0 are invalid because neither case can exist as a valid delta number). If the *SID* is valid and not ambiguous, a check is made to determine if it actually exists.

–m*name* The argument value *name* is compared with the SCCS %M% keyword in *file*.

–y*type* The argument value *type* is compared with the SCCS %Y% keyword in *file*.

The 8-bit code returned by *val* is a disjunction of the possible errors, i.e., can be interpreted as a bit string where (moving from left to right) set bits are interpreted as follows:

bit 0 = missing file argument;
bit 1 = unknown or duplicate keyletter argument;
bit 2 = corrupted SCCS file;
bit 3 = can't open file or file not SCCS;
bit 4 = *SID* is invalid or ambiguous;
bit 5 = *SID* does not exist;
bit 6 = %Y%, **–y** mismatch;
bit 7 = %M%, **–m** mismatch;

Note that *val* can process two or more files on a given command line and in turn can process multiple command lines (when reading the standard input). In these cases an aggregate code is returned – a logical **OR** of the codes generated for each command line and file processed.

SEE ALSO

admin(1), delta(1), get(1), prs(1).

SCCS User's Guide in *HP-UX Concepts and Tutorials*.

DIAGNOSTICS

Use ***help***(1) for explanations.

BUGS

Val can process up to 50 files on a single command line. Any number above 50 will produce a fatal error.

NAME

vi, view – visual text editor

SYNOPSIS

vi [**–t** tag] [**–r**] [+command] [**–l**] [**–w**n] [**–R**] [**–x**] name ...

view [vi options]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: UCB

DESCRIPTION

Vi (visual) is a display oriented text editor based on *ex*(1). *Ex* and *vi* run the same code; it is possible to get to the command mode of *ex* from within *vi*, and vice-versa. *View* is a read-only version of *vi*. The following options are recognized:

–t*tag* equivalent to an initial *tag* command. The file containing the *tag* is edited, with the editor positioned at the definition of the *tag*.

–r used to recover from the last editor or system crash. The last saved version of *name* is retrieved, or, if no *name* is specified, a list of saved files is printed.

+*command* The editor begins the editing session by executing the specified *ex*(1) command. If *command* is omitted, it defaults to **$**, positioning the editor at the last line of the current file.

–l automatically sets the *showmatch* and *lisp* options.

–w*n* sets the default window size to *n* lines.

–R sets the *readonly* option at the start of the editing session.

–x causes *vi* to prompt for a *key*, which is used to encrypt and decrypt the contents of the file (which should already be encrypted using the same key).

The Vi Editor provides full details on using *vi*.

HARDWARE DEPENDENCIES

Series 500:

The CTRL–f command does not work if ENQ-ACK handshake is enabled.

SEE ALSO

ex(1).

The Vi Editor in *HP-UX Concepts and Tutorials*.

BUGS

Software tabs using **^T** work only immediately after the *autoindent*.

Left and right shifts on intelligent terminals do not make use of insert and delete character operations in the terminal.

The *wrapmargin* option can be fooled since it looks at output columns when blanks are typed. If a long word passes through the margin and onto the next line without a break, then the line will not be broken.

Insert/delete within a line can be slow if tabs are present on intelligent terminals, since the terminals need help in doing this correctly.

Saving text on deletes in the named buffers is somewhat inefficient.

The *source* command does not work when executed as **:source**; there is no way to use the **:append**, **:change**, and **:insert** commands, since it is not possible to give more than one line of input to a **:** escape. To use these on a **:global** you must **Q** to *ex* command mode, execute them, and then reenter the screen editor with *vi* or *open*.

If the system crashes or *vi* is killed accidentally, the terminal may be left in raw mode. To get out of raw mode, type the following sequence:

CTRL–j
stty sane
CTRL–j

This sequence may alter the baud rate and the erase and kill characters (see *stty*(1)). To restore the erase and kill characters, type:

. /etc/profile
cd
. .profile

On systems which do not support *crypt(1)*, the **−x** option is not available.

NAME

wait – await completion of process

SYNOPSIS

wait

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

Wait waits until all processes started with **&** have completed, and reports on abnormal terminations.

Because the *wait*(2) system call must be executed in the parent process, the shell itself executes *wait*, without creating a new process.

SEE ALSO

sh(1).

BUGS

Not all the processes of a 3- or more-stage pipeline are children of the shell, and thus cannot be waited for.

NAME

wall – write to all users

SYNOPSIS

/etc/wall

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Wall reads its standard input until an end-of-file. It then sends this message to all currently logged in users preceded by "Broadcast Message from ...". It is used to warn all users, typically prior to shutting down the system.

The sender should be super-user to override any protections the users may have invoked.

Wall has timing delays, and will take at least 30 seconds to complete.

FILES

/dev/tty*

SEE ALSO

mesg(1), write(1).

DIAGNOSTICS

"Cannot send to ..." when the open on a user's tty file fails.

NAME

wc – word, line, and character count

SYNOPSIS

wc [–**lwc**] [file ...]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Wc counts lines, words and characters in the named *file*(s), or in the standard input if no *files* appear. It also keeps a total count for all named files. A word is a maximal string of characters delimited by spaces, tabs, or new-lines.

The options are:

–**l** count only lines in the named *file*(s);

–**w** count only words in the named *file*(s);

–**c** count only characters in the named *file*(s).

These options may be used in any combination. The default is –**lwc**.

When *multiple files* are specified on the command line, they will be printed along with the counts.

BUGS

Wc counts the number of new-lines to determine the line count. If an ASCII text file has a final line that is not terminated with a new-line character, the count will be off by one.

If there are very many characters, words, and/or lines in an input file, the output may be hard to read. This is because *wc* reserves a fixed column width for each count.

NAME

what – identify files for SCCS information

SYNOPSIS

what files

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

What searches the given files for all occurrences of the pattern that *get*(1) substitutes for %Z% (this is **@(#)** at this printing) and prints out what follows until the first ", >, new-line, \, or null character. For example, if the C program in file **f.c** contains

```
char ident[] = " @(#)identification information ";
```

and **f.c** is compiled to yield **f.o** and **a.out**, then the command

```
what f.c f.o a.out
```

will print

```
f.c:
        identification information
f.o:
        identification information
a.out:
        identification information
```

What is intended to be used in conjunction with the command *get*(1), which automatically inserts identifying information, but it can also be used where the information is inserted manually.

SEE ALSO

get(1), help(1).

SCCS User's Guide in *HP-UX Concepts and Tutorials.*

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

It's possible that an unintended occurrence of the pattern **@(#)** could be found just by chance, but this causes no harm in nearly all cases.

NAME

whereis – locate source, binary, and/or manual for program

SYNOPSIS

whereis [**–sbm**] [**–u**] [**–SBM** dir ... **–f**] file ...

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: UCB

DESCRIPTION

Whereis locates source/binary and manuals sections for the specified *file*s. The supplied *file*names are first stripped of leading pathname components and any (single) trailing extension of the form ".ext", e.g. ".c". Prefixes of "s." resulting from use of SCCS are also dealt with. *Whereis* then attempts to locate the desired *file* in a list of standard places.

The options are:

–s find only the source version of *file*;

–b find only the binary version of *file*;

–m find only the manual section(s) applicable to *file*.

The **–s**, **–b**, and **–m** options can be specified alone or in any combination of two.

–u search for unusual entries. A file is said to be unusual if it does not have one entry of each requested type. Thus

```
whereis –m –u *
```

asks for those files in the current directory which have no documentation.

–S *dir* search for source versions of files in the given *dir*ectory;

–B *dir* search for binary versions of files in the given *dir*ectory;

–M *dir* search for manual sections for files in the given *dir*ectory.

–f terminates the last directory list (given by **–S**, **–B**, and/or **–M**) and signals the start of file names.

EXAMPLE

The following finds all the files in /usr/bin which are not documented in /usr/man/man1 with source in /usr/src/cmd:

```
cd /usr/bin
whereis –u –M /usr/man/man1 –S /usr/src/cmd –f *
```

FILES

/usr/src/*
/usr/doc/*
/usr/man/*
/lib
/etc
/usr/lib
/usr/bin
/usr/ucb
/usr/old
/usr/new
/usr/local

BUGS

Directories given with the **–S**, **–B**, and **–M** options must be absolute pathnames.

NAME

who – list which users are on the system

SYNOPSIS

who [who-file] | [**am I**]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Who, without an argument, lists the login name, terminal name, and login time for each current HP-UX user. *Who* examines */etc/utmp* to obtain its information.

If a single argument is given, *who* assumes it is the name of a file which is to be used instead of */etc/utmp*. Typically the given file is */usr/adm/wtmp*, which contains a record of all the logins since it was created. If */usr/adm/wtmp* is given, *who* lists logins, logouts, and crashes since the creation of the **wtmp** file. Each login is listed with user name, terminal name (with **/dev/** suppressed), and date and time. When an argument is given, logouts produce a similar line without a user name. Reboots produce a line with **x** in the place of the device name, and a fossil time indicative of when the system went down.

If two arguments are given (it doesn't matter what they are – "**am i**" is suggested), *who* tells who you are logged in as.

FILES

/etc/utmp

SEE ALSO

getuid(2), utmp(5).

NAME

whodo – which users are doing what

SYNOPSIS

/etc/whodo

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Whodo produces merged, reformatted, and dated output from the *who*(1) and *ps*(1) commands.

SEE ALSO

ps(1), who(1).

NAME

write – interactively write (talk) to another user

SYNOPSIS

write user [tty]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Write copies lines from your terminal to that of another user. When first called, it sends the message:

Message from *your-logname your-tty* . . .

The recipient of the message should *write* back at this point. Communication continues until an end of file is read from the terminal or an interrupt is sent. At that point, *write* writes **EOF** on the other terminal and exits.

If you want to write to a user who is logged in more than once, the *tty* argument may be used to indicate the appropriate terminal.

Permission to write may be denied or granted by use of the *mesg*(1) command. At the outset, writing is allowed. Certain commands, in particular *nroff*(1) and *pr*(1), disallow messages in order to prevent messy output.

If the character **!** is found at the beginning of a line, *write* calls the shell to execute the rest of the line as a command.

The following protocol is suggested for using *write*: when you first write to another user, wait for him or her to write back before starting to send. Each party should end each message with a distinctive signal (**(o)** for "over" is conventional), indicating that the other may reply; **(oo)** for "over and out" is suggested when conversation is to be terminated.

FILES

/etc/utmp	to find user
/bin/sh	to execute **!**

SEE ALSO

mail(1), mesg(1), who(1).

NAME

xargs – construct argument list(s) and execute command

SYNOPSIS

xargs [flags] [command [initial-arguments]]

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Xargs combines the fixed *initial-arguments* with arguments read from standard input to execute the specified *command* one or more times. The number of arguments read for each *command* invocation and the manner in which they are combined are determined by the flags specified.

Command, which may be a shell script, is searched for, using your **$PATH**. If *command* is omitted, **/bin/echo** is used.

Arguments read in from standard input are defined to be contiguous strings of characters delimited by one or more blanks, tabs, or new-lines; empty lines are always discarded. Blanks and tabs may be embedded as part of an argument if escaped or quoted. Characters enclosed in quotes (single or double) are taken literally, and the delimiting quotes are removed. Outside of quoted strings a backslash (\) escapes the next character.

Each argument list is constructed starting with the *initial-arguments*, followed by some number of arguments read from standard input (Exception: see **–i** flag). Flags **–i**, **–l**, and **–n** determine how arguments are selected for each command invocation. When none of these flags are coded, the *initial-arguments* are followed by arguments read continuously from standard input until an internal buffer is full, and then *command* is executed with the accumulated args. This process is repeated until there are no more args. When there are flag conflicts (e.g., **–l** vs. **–n**), the last flag has precedence. *Flag* values are:

–l*number* *Command* is executed for each non-empty *number* lines of arguments from standard input. The last invocation of *command* will be with fewer lines of arguments if fewer than *number* remain. A line is considered to end with the first new-line *unless* the last character of the line is a blank or a tab; a trailing blank/tab signals continuation through the next non-empty line. If *number* is omitted, 1 is assumed. Option **–x** is forced.

–i*replstr* Insert mode: *command* is executed for each line from standard input, taking the entire line as a single arg, inserting it in *initial-arguments* for each occurrence of *replstr*. A maximum of 5 arguments in *initial-arguments* may each contain one or more instances of *replstr*. Blanks and tabs at the beginning of each line are thrown away. Constructed arguments may not grow larger than 255 characters, and option **–x** is also forced. {} is assumed for *replstr* if not specified.

–n*number* Execute *command* using as many standard input arguments as possible, up to *number* arguments maximum. Fewer arguments are used if their total size is greater than *size* characters, and for the last invocation if there are fewer than *number* arguments remaining. If option **–x** is also coded, each *number* arguments must fit in the *size* limitation, else *xargs* terminates execution.

–t Trace mode: The *command* and each constructed argument list are echoed to file descriptor 2 just prior to their execution.

–p Prompt mode: You are asked whether to execute *command* each invocation. Trace mode (**–t**) is turned on to print the command instance to be executed, followed by a **?...** prompt. A reply of **y** (optionally followed by anything) executes the command; anything else, including just a carriage return, skips that particular invocation of *command*.

–x Causes *xargs* to terminate if any argument list would be greater than *size* characters; **–x** is forced by the options **–i** and **–l**. When neither of the options **–i**, **–l**, or **–n** are coded, the total length of all arguments must be within the *size* limit.

–s*size* The maximum total size of each argument list is set to *size* characters; *size* must be a positive integer less than or equal to 470. If **–s** is not coded, 470 is taken as the default. Note that the character count for *size* includes one extra character for each argument and the count of characters in the command name.

–e*eofstr* *Eofstr* is taken as the logical end-of-file string. Underbar (_) is assumed for the logical **EOF** string if **–e** is not coded. The value **–e** with no *eofstr* coded turns off the logical **EOF** string capability (underbar is taken literally). *Xargs* reads standard input until either end-of-file or the logical **EOF** string is encountered.

Xargs terminates if it receives a return code of **–1** from, or if it cannot execute, *command*. When *command* is a shell script, it should explicitly *exit* (see *sh*(1)) with an appropriate value to avoid accidentally returning with **–1**.

EXAMPLES

The following moves all files from directory $1 to directory $2, and echos each *mv* command just before doing it:

```
ls $1 | xargs –i –t mv $1/{} $2/{}
```

The following combines the output of the parenthesized commands onto one line, which is then echoed to the end of file *log*:

```
(logname; date; echo $0 $*) | xargs >>log
```

This example asks you which files in the current directory are to be archived and archives them into *arch* (1.) one at a time, or (2.) many at a time.

```
1. ls | xargs –p –l ar r arch
2. ls | xargs –p –l | xargs ar r arch
```

The following executes *diff*(1) with successive pairs of arguments originally typed as shell arguments:

```
echo $* | xargs –n2 diff
```

SEE ALSO

sh(1).

DIAGNOSTICS

Self-explanatory.

NAME

yacc – yet another compiler-compiler

SYNOPSIS

yacc [**–dv**] grammar

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Yacc converts a context-free grammar into a set of tables for a simple automaton which executes an LR(1) parsing algorithm. The grammar may be ambiguous; specified precedence rules are used to break ambiguities.

The output file, **y.tab.c**, must be compiled by the C compiler to produce a program *yyparse*. This program must be loaded with the lexical analyzer program, *yylex*, as well as *main* and *yyerror*, an error handling routine. These routines must be supplied by the user; *lex*(1) is useful for creating lexical analyzers usable by *yacc*.

The options are:

–d produces the file **y.tab.h**, containing the **#define** statements that associate the *yacc*-assigned "token codes" with the user-declared "token names". This allows source files other than **y.tab.c** to access the token codes.

–v produces the file **y.output**, which contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.

FILES

y.output	
y.tab.c	
y.tab.h	definitions (#define's) for token names
yacc.tmp, yacc.acts	temporary files
/usr/lib/yaccpar	parser prototype for C programs

SEE ALSO

lex(1).

YACC – Yet Another Compiler Compiler in *HP-UX Concepts and Tutorials*.

DIAGNOSTICS

The number of reduce-reduce and shift-reduce conflicts is reported on the standard output; a more detailed report is found in the **y.output** file. Similarly, if some rules are not reachable from the start symbol, this is also reported.

BUGS

Because file names are fixed, at most one *yacc* process can be active in a given directory at a time.

Manual Comment Card

If you have any comments or questions regarding this manual, write them on this comment card and place it in the mail. Include page numbers with your comments wherever possible. Enter the last date from the Printing History page on the line above your name. Also include a return address so that we can respond as soon as possible.

HP-UX Reference
for the HP 9000 Series 200/500

09000-90007 March 1985

Last Date: _____
(See the Printing History in the front of the manual)

Name: _____

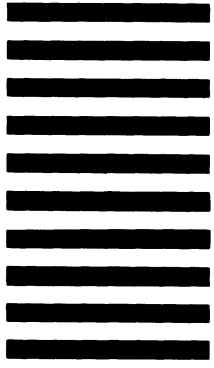
Company: _____

Address: _____

_____

Phone No: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Fort Collins Systems Division
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525

Manual Comment Card

If you have any comments or questions regarding this manual, write them on this comment card and place it in the mail. Include page numbers with your comments wherever possible. Enter the last date from the Printing History page on the line above your name. Also include a return address so that we can respond as soon as possible.

HP-UX Reference
for the HP 9000 Series 200/500

09000-90007 March 1985

Last Date: _____
(See the Printing History in the front of the manual)

Name: _____

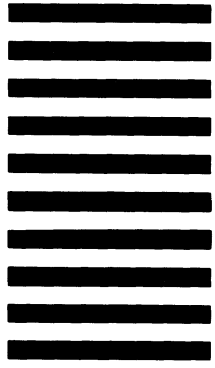
Company: _____

Address: _____

_____

Phone No: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Fort Collins Systems Division
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525

Manual Comment Card

If you have any comments or questions regarding this manual, write them on this comment card and place it in the mail. Include page numbers with your comments wherever possible. Enter the last date from the Printing History page on the line above your name. Also include a return address so that we can respond as soon as possible.

HP-UX Reference
for the HP 9000 Series 200/500

09000-90007 March 1985

Last Date: _____
(See the Printing History in the front of the manual)

Name: _____

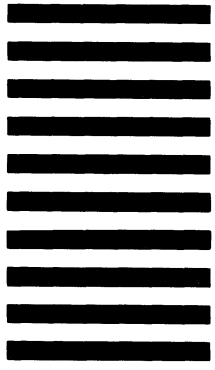
Company: _____

Address: _____

_____

Phone No: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Fort Collins Systems Division
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525

Part No. 09000-90007
E 0385
Microfiche No. 09000-99007

Printed in U.S.A.
Edition 1 with update
March 1985