HP-UX Reference

Vol. 1: Sections 1 and 9

HP 9000 Series 500 Computers
HP-UX Release 5.2

HP Part Number 09000-90010

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

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MANUAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

WARRANTY

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

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.

April 1987...Edition 1

INTRODUCTION

HP-UX is Hewlett-Packard Company's implementation of a standard operating system that is compatible with the AT&T UNIX* System V Release 2 operating system, but which also includes important features from Berkeley Software Distribution 4.2 as well as many bug fixes, enhanced capabilities, and other features developed by HP. This combination makes HP-UX a very powerful and useful operating system capable of supporting a wide range of applications ranging from simple text processing to sophisticated engineering graphics and design. It can also be readily used to control instruments and other peripheral devices through the HP-IB (IEEE-488) and HP-HIL interfaces as well as through GPIO interfacing. Real-time capabilities further expand HP-UX's flexibility as a powerful tool for solving a myriad of problems in design, manufacturing, business, and other areas of interest. Extensive internationalization makes HP-UX readily adaptable to a myriad of spoken languages, and interfacing to Local Area Network and many other networking and resource-sharing facilities provides flexible interaction with other computers and operating systems. Optional software products extend HP-UX capabilities into a broad range of specialized needs.

This manual is not intended for use as a learning tool for beginners. It is a reference guide that is most useful to experienced users of UNIX or UNIX-like systems. If you are not already familiar with UNIX and HP-UX, refer to the tutorial manuals and other learning documents supplied with your system. System implementation and maintenance details are explained in the *HP-UX System Administrator Manual.*

Manual Organization

This manual is divided into several sections contained in two volumes. Volume 1 contains Sections 1 and 9 as well as a permuted index. The remaining sections (1M and 2 thorugh 8) are in volume 2.

Section 1 (*Commands and Application Programs*) describes programs that are usually invoked directly by users or from command language procedures, as opposed to system calls (section 2) or subroutines (section 3) that are called by user and application programs. Commands usually reside in the directory **/bin** (for **bin**ary programs). Some programs 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 a shell (*sh*(1) or *csh*(1)). Other Section 1 commands are located in **/lib** and **/usr/lib**. Refer to *hier*(5) and tutorial manuals supplied with your system for more information about file system structure.

Section 1M (*System Maintenance Procedures*) describes commands used for system maintenance including boot processes, crash recovery, system integrity testing, and other needs. This section contains topics that pertain primarily to system administrator and super-user tasks.

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

Section 3 (*Subroutines*) describes available subroutines that reside (in binary form) in various system libraries stored in directories **/lib** and **/usr/lib**. Refer to *intro*(3) for descriptions of these libraries and the files where they are stored.

* UNIX is a registered trademark of AT&T Bell Laboratories, Inc.

Section 4 (*File Formats*) documents the structure of various types of files. For example, the link editor output-file format is described in *a.out*(4). Files that are used only by a single command (such as intermediate files used by assemblers) are not described. C-language **struct** declarations corresponding to the formats in Section 4 can be found in directories **/usr/include** and **/usr/include/sys**.

Section 5 (*Miscellaneous Facilities*) contains a variety of information such as descriptions of character sets, macro packages, and other topics.

Section 6 (*Games*) is absent because no games are currently supported on HP-UX.

Section 7 (*Device Special Files*) discusses the characteristics of special (device) files that provide the link between HP-UX and system I/O devices. The names for each topic usually refer to the type of I/O device rather than to the names of individual special files.

Section 9 (*Glossary*) is located in Volume 1 after Section 1. It defines selected terms used in this manual.

Permuted Index

An alphabetical listing based on rotation of the NAME line on the first page each manual page entry. The center column is in alphabetical order, and the name on the page heading related to the subject is in the right-hand column.

Each section (except 9) contains a number of independent entries frequently referred to as **manpages** or **manual pages**. Each manpage entry consists of one or more pages, with the entry or page name printed in the upper corners of each page. Manpage entries are arranged alphabetically within each section of the reference, except for the introductory entry at the beginning of each section. Textual references to manpage entries are of the form *pagename*(NS) where N is the section number and S is a subsection identifier letter (Section 3 only). For example, *io_burst*(3I) refers to an entry in the subroutine I/O library (Section 3 library subsection I) by the name of *ioburst*.

Page numbering is arranged so that each entry starts on its own page 1. Some manpage entries describe several commands or routines on a single manpage. In such cases, the manpage is not duplicated for each topic, but appears only once, usually arranged under the first keyword appearing in the NAME section of the manpage. Occasionally, another name is used for the page. In such instances, the name describes the keywords in more general terms such as the entry for *acct* or *acctsh* in Section 1M or *string* in Section 3.

SYSTEM STANDARDIZATION

This reference is based on extensive system-design control documents that have been used to ensure software compatiblity across HP-UX computer model lines. HP-UX is compatible with AT&T UNIX System V Interface Definition (SVID), but also includes important additional features from Berkeley Software Distribution 4.2 plus HP enhancements for international language support, real-time, graphics, and instrumentation capabilities. HP-UX also contains numerous bug fixes and has been extensively tested in real-use environments.

As of this printing, HP-UX has been implemented on HP 9000 Series 200, 300, 500, and 800 computers. This document is valid for first release of HP-UX on Series 800 as well as Release 5.2 on Series 300 and Series 500. Releases prior to 5.2 on Series 300, 5.1 and earlier on Series 500, and 5.0/5.1 on Series 200 use the *HP-UX Reference* part number 09000-90008.

The Integral PC also supports HP-UX, and the Series 200/300 AXE (Applications Execution Environment) supports a subset of the standard HP-UX operating system. A list of commands and features supported by various HP-UX systems is listed later in this introduction.

PAGE HEADERS AND FOOTERS

Since HP-UX is still being actively developed and expanded to meet the emerging demands of evolving technologies, some capabilities have not been fully integrated into all product implementations. Also, some systems have unique needs that may not be appropriate on other series (such as *osmark*(1M) or *oscp*(1M) which are on Series 500 only). The headers and footers on each manpage are designed to clearly identify:

- Whether or not the page has been standardized across model lines, and
- Which series in the HP 9000 computer family support the features documented on that particular page.

If the page has been standardized, the date in the lower right corner is preceded by a standard version number. In addition, the word **HP-UX** is printed in the center of the top (header) line.

If a given manpage defines a feature that is not implemented on all series, a second header line identifies which series numbers support the feature. The second line, when present, contains **Series** xxx **Only** or **Series** xxx **Implementation**, where *xxx* represents the series numbers that support the page. The word "implementation" in the header indicates that the feature is implemented on each series indicated but that the implementation on one series differs from another in various aspects of its operation. Such cases typically arise when hardware distinctions force software uniqueness (such as in assemblers).

If a manpage is missing the version number at the bottom and the HP-UX label at the top, the feature has not been standardized, and should not be used in software that must be readily portable without modification across various HP-UX systems.

MANPAGE FORMATS

All manpage entries follow an established topic format, but not all topics are included in each entry.

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

SYNOPSIS Summarizes the use of the entry or program entity being described. A few conventions are used:

Boldface strings are literals, and are to be typed exactly as they appear in the manual.

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.

Ellipses (...) are used to show that the previous argument can 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 names that begin with -, +, or =.

DESCRIPTION
Discusses the function and behavior of each entry.

HARDWARE DEPENDENCIES
Points out variations in HP-UX operation that are related to the use of specific hardware or combinations of hardware.

EXAMPLES Provides examples of typical usage, where appropriate.

FILES Lists file names that are built into the program or command.

RETURN VALUE
Discusses various values returned upon completion of program calls.

SEE ALSO Provides pointers to related topics.

DIAGNOSTICS
Discusses diagnostic indications that may be produced. Self-explanatory messages are not listed.

WARNINGS Points out potential pitfalls.

BUGS Discusses known bugs and observed deficiencies. Occasionally, suggested fixes are provided.

AUTHOR Indicates the origin of the software documented by the manpage.

Some manual pages also include examples for illustration purposes to indicate typical use.

The table of contents included at the beginning of each volume contains a complete listing of all manpages in the order they appear in each section. Additional alphabetical entries identify all keywords on manpages that document multiple keywords, providing an easy path for locating commands and features whose names do not match the title heading on the corresponding manpage.

HOW TO GET STARTED

This discussion provides a very brief overview of how to use HP-UX: how to log in and log out, how to communicate through your machine, and how to run a program (if you are a beginning user, refer to other tutorial manuals 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 your system administrator. Keep pressing the "break" or "del" until the **login:** message appears.

When a connection has been established, the system displays **login:** on your terminal. Type your user name then press the RETURN key. 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 in your login name in lowercase if possible. If you type uppercase letters, HP-UX assumes that your terminal cannot generate lowercase letters, and treats subsequent uppercase input as lowercase. When you have logged in successfully, the shell displays a **$** prompt unless programmed for a different prompt (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 (see *csh*(1) for information about *ignoreeof* if you are using C-shell). The shell will terminate and the **login:** message will appear again.

How to Communicate Through Your Terminal

HP-UX gathers keyboard input characters and saves them in a buffer. The accumulated characters are not passed to the shell or other program until a **RETURN** is typed.

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 severely overloaded or operating abnormally. When the read-ahead limit is exceeded, the system throws away **all** the saved characters.

Erase, Kill, and Output Interrupt/Resume Characters

On an input line from the terminal, the character @ "kills" all 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 passes as output to later programs.

Interrupt and Quit Characters

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.

End-of-Line and Tab Characters

Besides adapting to the speed of the terminal, HP-UX tries to be intelligent as to whether you have a terminal with a new-line (line-feed) key, or whether it must be simulated with a carriage-return and line-feed pair. In the latter case, all incoming carriage-return characters are changed to line-feed characters (the standard line delimiter), and a carriage-return/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 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 a shell is monitoring input from your terminal. The shell accepts typed lines from the terminal, splits them into command names and arguments, then executes the command which is nothing more than an executable program. Usually, the shell looks first in your current directory (discussed 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 you 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 *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 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.

Inter-User Communication

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) can be used to transfer text directly to that user's terminal display (if permission to do so has been granted by the user). Otherwise, *mail*(1) or *mailx*(1) sends a message to that user's mailbox. The user is then informed by HP-UX that mail has arrived (if currently logged in) or mail is present (when he or she next logs in). Refer to the *mail*, *mailx*, and *write*, manpages in Section 1 for explanations of how each of these commands is used.

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

HP-UX FILE SYSTEMS

HP-UX supports several file systems, depending on which series you are using. The three systems implemented are:

BFS — Bell File System. This file system format is implemented on the Integral PC and on Series 200 prior to HP-UX Release 5.0. BFS files can be accessed on newer systems by using the BIF (Bell Interchange Format) utilities such as *bifcp*(1), *bifls*(1), *biffind*(1), etc.

HFS — High-performance File System. This file system format is implemented on all Series 300 and 800 systems, and on Series 200 beginning at Release 5.0.

SDF — Structured Directory Format. This file system format is implemented on all Series 500 releases. Use the SDF utilities such as *sdfcp*(1), *sdfinit*(1), *sdfrm*(1), *sdffind*(1), etc. to access SDF files from other systems.

File system formats are transparent to most users, and are of little importance in most applications. Most of the time, formats only prevent direct reading of disks of a particular format on a machine that supports a different format. Thus, SDF cannot be read on an HFS system without using SDF utilities. However, an SDF-based system can readily transfer files to an HFS-based system over UUCP, LAN, or other data communication facilities.

When transportable data is needed, a tape cartridge or flexible disk can be used. Flexible disks can be readily formatted and read or written in LIF (Logical Interchange Format) using the *lifinit*, *lifcp*, *lifls*, *lifrename*, and *lifrm* commands in Section 1, and are easily used on other non-HP-UX systems that support the HP LIF format.

IMPLEMENTED STANDARD COMMANDS

The following commands and facilities from the *HP-UX Standard* have been implemented on the series indicated. For commands and other features implemented on the Series 300 AXE (Applications EXecution Environment), refer to the *Application Execution Environment User's Guide*.

Section 1 Commands: All Systems

acctcom(1)	diff3(1)	id(1)	nroff(1)	tail(1)
adjust(1)	diffh(1)	insertmsg(1)	od(1)	tar(1)
admin(1)	diffmk(1)	inv(1)	osdd(1)	tbl(1)
ar(1)	dircmp(1)	ipcrm(1)	pack(1)	tee(1)
as(1)	dirname(1)	ipcs(1)	page(1)	test(1)
asa(1)	disable(1)	join(1)	passwd(1)	time(1)
at(1)	du(1)	kill(1)	paste(1)	touch(1)
awk(1)	dumpmsg(1)	l(1)	pathalias(1)	tput(1)
banner(1)	echo(1)	last(1)	pc(1)	tr(1)
basename(1)	ed(1)	lastb(1)	pcat(1)	true(1)
batch(1)	edit(1)	ld(1)	pdp11(1)	tset(1)
bc(1)	egrep(1)	leave(1)	pg(1)	tsort(1)
bdiff(1)	enable(1)	lex(1)	pr(1)	tty(1)
bfs(1)	env(1)	lifcp(1)	prealloc(1)	u3b(1)
bs(1)	ex(1)	lifinit(1)	primes(1)	u3b5(1)
cal(1)	expand(1)	lifls(1)	prmail(1)	ul(1)
calendar(1)	expr(1)	lifrename(1)	prs(1)	umask(1)
cancel(1)	f77(1)	lifrm(1)	ps(1)	uname(1)
cat(1)	factor(1)	line(1)	ptx(1)	unexpand(1)
cb(1)	false(1)	lint(1)	pwd(1)	unget(1)
cc(1)	fc(1)	ll(1)	ratfor(1)	uniq(1)
cd(1)	fgrep(1)	ln(1)	red(1)	units(1)
cdc(1)	file(1)	lock(1)	rev(1)	unpack(1)
cflow(1)	find(1)	login(1)	rm(1)	uucp(1)
chgrp(1)	findmsg(1)	logname(1)	rmail(1)	uulog(1)
chmod(1)	findstr(1)	lorder(1)	rmdel(1)	uuname(1)
chown(1)	fixman(1)	lp(1)	rmdir (1)	uupick(1)
chsh(1)	fold(1)	lpstat(1)	rmnl(1)	uustat(1)
clear(1)	from(1)	ls(1)	rsh(1)	uuto(1)
cmp(1)	gencat(1)	lsf(1)	rtprio(1)	uux(1)
col(1)	get(1)	lsr(1)	sact(1)	val(1)
comb(1)	getopt(1)	lsx(1)	sccsdiff(1)	vax(1)
comm(1)	grep(1)	m4(1)	sdb(1)	vc(1)
cp(1)	hashcheck(1)	mail(1)	sdiff(1)	vi(1)
cpio(1)	hashmake(1)	mailx(1)	sed(1)	vis(1)
cpp(1)	head(1)	make(1)	sh(1)	wait(1)
crontab(1)	help(1)	makekey(1)	size(1)	wc(1)
csh(1)	hostname(1)	man(1)	sleep(1)	what(1)
ctags(1)	hp(1)	mesg(1)	sort(1)	whereis(1)
cu(1)	hp9000s200(1)	mkdir(1)	spell(1)	which(1)
cut(1)	hp9000s300(1)	mkstr(1)	spellin(1)	who(1)
cxref(1)	hp9000s500(1)	mm(1)	split(1)	whoami(1)
date(1)	hp9000s800(1)	more(1)	ssp(1)	write(1)
dc(1)	hyphen(1)	mt(1)	strings(1)	xargs(1)
dd(1)	id(1)	mv(1)	stty(1)	xd(1)

delta(1)	insertmsg(1)	neqn(1)	su(1)	yacc(1)
deroff(1)	inv(1)	newform(1)	sum(1)	
diff(1)	ipcrm(1)	newgrp(1)	tabs(1)	

Distribution of *crypt*(1) which runs on all series is restricted.

Section 1 Commands: Series 200, 300, and 500 Only

bifchgrp(1)	bifcp(1)	bifmkdir(1)	cdb(1)	pdb(1)
bifchmod(1)	biffind(1)	bifrm(1)	ct(1)	send(1)
bifchown(1)	bifls(1)	bifrmdir(1)	fdb(1)	

Section 1 Commands: Series 200, 300, and 800 Only

adb(1)	groups(1)	prof(1)	slp(1)	tcio(1)
getprivgrp(1)	mediainit(1)			

Section 1 Commands: Series 300 and 800 Only

sdfchmod(1)	sdfcp(1)	sdfln(1)	sdfmv(1)	iostat(1)
sdfchgrp(1)	sdffind(1)	sdfls(1)	sdfrm(1)	
sdfchown(1)	sdfll(1)	sdfmkdir(1)	sdfrmdir(1)	

Section 1 Commands: Series 800 Only

csplit(1)	nm(1)	strip(1)	vmstat(1)	
_exit(2)	fchmod(2)	ioctl(2)	rtprio(2)	sigspace(2)

Section 2 System Calls: All Systems

access(2)	fchown(2)	kill(2)	sbrk(2)	sigvector(2)
acct(2)	fcntl(2)	link(2)	select(2)	stat(2)
alarm(2)	fork(2)	lockf(2)	semctl(2)	stime(2)
brk(2)	fstat(2)	lseek(2)	semget(2)	stty(2)
chdir(2)	fsync(2)	mkdir(2)	semop(2)	sync(2)
chmod(2)	ftruncate(2)	mknod(2)	setgid(2)	time(2)
chown(2)	getegid(2)	mount(2)	sethostname(2)	times(2)
chroot(2)	geteuid(2)	msgctl(2)	setitimer(2)	truncate(2)
close(2)	getgid(2)	msgget(2)	setpgrp(2)	ulimit(2)
creat(2)	gethostname(2)	msgop(2)	setpgrp2(2)	umask(2)
dup(2)	getitimer(2)	nice(2)	settimeofday(2)	umount(2)
dup2(2)	getpgrp(2)	open(2)	setuid(2)	uname(2)
errno(2)	getpgrp2(2)	pause(2)	shmctl(2)	unlink(2)
execl(2)	getpid(2)	pipe(2)	shmget(2)	ustat(2)
execle(2)	getppid(2)	plock(2)	shmop(2)	utime(2)
execlp(2)	gettimeofday(2)	prealloc(2)	sigblock(2)	vfork(2)
execv(2)	getuid(2)	read(2)	signal(2)	wait(2)
execve(2)	gtty(2)	readv(2)	sigpause(2)	write(2)
execvp(2)	intro(2)	rmdir(2)	sigsetmask(2)	writev(2)
exit(2)				

Section 2 System Calls: Series 200, 300, and 800 Only

ftime(2)	getprivgrp(2)	ptrace(2)	setgroups(2)	swapon(2)
getgroups(2)	profil(2)	reboot(2)	setprivgrp(2)	

Section 2 System Calls: Series 500 Only

ems(2)	memchmd(2)	memlck(2)	memvary(2)	vson(2)
memadvise(2)	memfree(2)	memulck(2)	vsadv(2)	vsoff(2)
memallc(2)				

Section 2 System Calls: Series 800 Only

setresgid(2)	setresuid(2)

Section 3 Subroutines: All Systems

_tolower(3C)	gamma(3M)	l3tol(3C)	setkey(3C)
_toupper(3C)	gcvt(3C)	l64a(3C)	setpwent(3C)
a64l(3C)	getc(3S)	langinfo(3C)	setutent(3C)
abort(3C)	getchar(3S)	langtoid(3C)	setvbuf(3S)
abs(3C)	getcwd(3C)	lcong48(3C)	sgetl(3X)
acos(3M)	getenv(3C)	ldexp(3C)	signgam(3M)
asctime(3C)	getfsent(3X)	lfind(3C)	sin(3M)
asin(3M)	getfsfile(3X)	localtime(3C)	sinh(3M)
assert(3X)	getfsspec(3X)	log(3M)	sleep(3C)
atan(3M)	getfstype(3X)	log10(3M)	sprintf(3S)
atan2(3M)	getgrent(3C)	logname(3X)	sprintmsg(3C)
atof(3C)	getgrgid(3C)	longjmp(3C)	sputl(3X)
atoi(3C)	getgrnam(3C)	lrand48(3C)	sqrt(3M)
atol(3C)	getlogin(3C)	lsearch(3C)	srand(3C)
bsearch(3C)	getmsg(3C)	ltol3(3C)	srand48(3C)
calloc(3C)	getopt(3C)	mallinfo(3X)	sscanf(3S)
calloc(3X)	getpass(3C)	malloc(3C)	ssignal(3C)
catread(3C)	getpw(3C)	malloc(3X)	stdio(3S)
ceil(3M)	getpwent(3C)	mallopt(3X)	strcat(3C)
clearerr(3S)	getpwnam(3C)	matherr(3M)	strchr(3C)
clock(3C)	getpwuid(3C)	memccpy(3C)	strcmp(3C)
closedir(3C)	gets(3S)	memchr(3C)	strcmp16(3C)
cos(3M)	getutent(3C)	memcmp(3C)	strcmp8(3C)
cosh(3M)	getutid(3C)	memcpy(3C)	strcpy(3C)
crypt(3C)	getutline(3C)	memset(3C)	strcspn(3C)
ctermid(3S)	getw(3S)	mktemp(3C)	strlen(3C)
ctime(3C)	gmtime(3C)	modf(3C)	strncat(3C)
currlangid(3C)	gpio_get_status(3I)	mrand48(3C)	strncmp(3C)
curses(3X)	gpio_set_ctl(3I)	nl_asctime(3C)	strncmp16(3C)
cuserid(3S)	gsignal(3C)	nl_atof(3C)	strncmp8(3C)
daylight(3C)	hcreate(3C)	nl_ctime(3C)	strncpy(3C)
dial(3C)	hdestroy(3C)	nl_gcvt(3C)	strpbrk(3C)
drand48(3C)	hpib_abort(3I)	nl_isalnum(3C)	strrchr(3C)
ecvt(3C)	hpib_bus_status(3I)	nl_isalpha(3C)	strspn(3C)
edata(3C)	hpib_card_ppoll_resp(3I)	nl_isgraph(3C)	strtod(3C)

encrypt(3C)
end(3C)
endfsent(3X)
endgrent(3C)
endpwent(3C)
endutent(3C)
erand48(3C)
erf(3M)
erfc(3M)
errno(3C)
etext(3C)
exp(3M)
fabs(3M)
fclose(3S)
fcvt(3C)
fdopen(3S)
feof(3S)
ferror(3S)
fflush(3S)
fgetc(3S)
fgetgrent(3C)
fgetpwent(3C)
fgets(3S)
fileno(3S)
floor(3M)
fmod(3M)
fopen(3S)
fprintf(3S)
fprintmsg(3C)
fputc(3S)
fputs(3S)
fread(3S)
free(3C)
free(3X)
freopen(3S)
frexp(3C)
fscanf(3S)
fseek(3S)
ftell(3S)
ftok(3C)
ftw(3C)
fwrite(3S)
hpib_eoi_ctl(3I)
hpib_io(3I)
hpib_pass_ctl(3I)
hpib_ppoll(3I)
hpib_ppoll_resp_ctl(3I)
hpib_ren_ctl(3I)
hpib_rqst_srvce(3I)
hpib_send_cmnd(3I)
hpib_spoll(3I)
hpib_status_wait(3I)
hpib_wait_on_ppoll(3I)
hsearch(3C)
hypot(3M)
idtolang(3C)
initgroups(3C)
io_eol_ctl(3I)
io_get_term_reason(3I)
io_interrupt_ctl(3I)
io_lock(3I)
io_on_interrupt(3I)
io_reset(3I)
io_speed_ctl(3I)
io_timeout_ctl(3I)
io_unlock(3I)
io_width_ctl(3I)
isalnum(3C)
isalpha(3C)
isascii(3C)
isatty(3C)
iscntrl(3C)
isdigit(3C)
isgraph(3C)
islower(3C)
isprint(3C)
ispunct(3C)
isspace(3C)
isupper(3C)
isxdigit(3C)
j0(3M)
j1(3M)
jn(3M)
jrand48(3C)
nl_islower(3C)
nl_isprint(3C)
nl_ispunct(3C)
nl_isupper(3C)
nl_strtod(3C)
nl_tolower(3C)
nl_tools_16(3C)
nl_toupper(3C)
nrand48(3C)
opendir(3C)
optarg(3C)
opterr(3C)
optind(3C)
pclose(3S)
perror(3C)
popen(3S)
pow(3M)
printf(3S)
printmsg(3C)
putc(3S)
putchar(3S)
putenv(3C)
putpwent(3C)
puts(3S)
pututline(3C)
putw(3S)
qsort(3C)
rand(3C)
readdir(3C)
realloc(3C)
realloc(3X)
regcmp(3X)
regex(3X)
rewind(3S)
rewinddir(3C)
scanf(3S)
seed48(3C)
seekdir(3C)
setbuf(3S)
setfsent(3X)
setgrent(3C)
setjmp(3C)
strtok(3C)
strtol(3C)
swab(3C)
sys_errlist(3C)
sys_nerr(3C)
system(3S)
tan(3M)
tanh(3M)
tdelete(3C)
telldir(3C)
tempnam(3S)
tfind(3C)
tgetent(3X)
tgetflag(3X)
tgetnum(3X)
tgetstr(3X)
tgoto(3X)
timezone(3C)
tmpfile(3S)
tmpnam(3S)
toascii(3C)
tolower(3C)
toupper(3C)
tputs(3X)
tsearch(3C)
ttyname(3C)
ttyslot(3C)
twalk(3C)
tzname(3C)
tzset(3C)
undial(3C)
ungetc(3S)
utmpname(3C)
vfprintf(3S)
vprintf(3S)
vsprintf(3S)
y0(3M)
y1(3M)
yn(3M)

Distribution of *crypt*(3C) and *encrypt*(3C) which run on all series is restricted.

Section 3 Subroutines: Series 200/300 and 800 Only

monitor(3C)
nlist(3C)

Section 3 Subroutines: Series 800 Only

blmode(3C)
datalock(3C)

Section 4 Files: All Systems

a.out(4)	cpio(4)	inittab(4)	model(4)	ttytype(4)
acct(4)	dialups(4)	issue(4)	passwd(4)	tztab(4)
ar(4)	d_passwd(4)	lif(4)	profile(4)	utmp(4)
checklist(4)	fspec(4)	magic(4)	sccsfile(4)	wtmp(4)
col_seq_8(4)	gettydefs(4)	mknod(4)	term(4)	btmp(4)
core(4)	group(4)	mnttab(4)	terminfo(4)	

Section 4 Files: Series 200, 300, and 500 Only

bif(4)

Section 4 Files: Series 200, 300, and 800 Only

dir(4)	fs(4)	inode(4)	nlist(4)	privgrp(4)
disktab(4)				

Section 4 Files: Series 300 and 800 Only

sdf(4)

Section 5 Miscellaneous: All Systems

advance(5)	GETC(5)	kana8(5)	PEEKC(5)	term(5)
ascii(5)	hier(5)	langid(5)	RETURN(5)	types(5)
compile(5)	hpnls(5)	man(5)	roman8(5)	UNGETC(5)
environ(5)	INIT(5)	math(5)	stat(5)	values(5)
ERROR(5)	ioctl(5)	mm(5)	step(5)	varargs(5)
fcntl(5)				

Section 5 Miscellaneous: Series 800 Only

prof(5)

Section 7 Device Files: All Systems

console(7)	lp(7)	mt(7)	pty(7)	termio(7)
ct(7)	modem(7)	null(7)	stty(7)	tty(7)
disk(7)				

Section 7 Device Files: Series 200/300 and 800 Only

mem(7)	kmem(7)

IMPLEMENTED NON-STANDARD AND HARDWARE-DEPENDENT COMMANDS

The following non-standard commands and facilities and hardware-dependent features have been implemented on the series indicated:

Series 300 Only

as(1)
astrn(1)
atrans(1)
dos2ux(1)
doschmod(1)
doscp(1)
dosdf(1)
dosls(1)
dosmkdir(1)
dosrm(1)
pam(1)

mvdevs(1M) (temporary command, Release 5.2 only)
reconfig(1M)

crt0(3): crt0.o, mcrt0.o, frt0.o, mfrt0.o
cvtnum(3C)

dosif(4)

Series 200/300 Only

chatr(1)
lsdev(1)
nm(1)

backup(1M)
config(1M)

io_burst(3I)

a.out(4)
core(4)
master(4)

graphics(7): CRT graphics
iomap(7)

Series 500 Only

aterm(1)
backup(1)
chatr(1)
err(1)
ftio(1)
linkinfo(1)
lsdev(1)
nm(1)
query(1)
revision(1)
tcio(1)
upm(1)

autobkup(1M)
backup(1M)
chsys(1M)
osck(1M)
oscp(1M)
osmark(1M)
osmgr(1M)
rootmark(1M)
stopsys(1M)
uconfig(1M)
vstat(1M)

errinfo(2)
trapno(2)

intrapoff(3M)
intrapon(3M)

a.out(4)
core(4)
dir(4)
errfile(4)
fs(4)
inode(4)

Series 300/500 Only

compress(1)
man(1) (compressed manual pages version)
shl(1)
uncompress(1)
stty(1)
zcat(1)
tty(1)

catman(1M) (compressed manual pages version)
mkrs(1M)

Series 200, 300, and 500 Only

arcv(1)
basic(1)
ccat(1)
compact(1)
kermit(1)
strip(1)
umodem(1)
uncompact(1)
vt(1)

revck(1M)
sysrm(1M)
update(1M)
vtdaemon(1M)

ranlib(4)

Series 800 Only

as(1)	lssf(1)	xdb(1)	hpux(1M)	devices(4)
hpiutil(1)	mksf(1)	boot(1M)	isl(1M)	afi(7)
insf(1)	psqlc(1)	decode(1M)	pdc(1M)	diag0(7)
iquery(1)	sqlutil(1)	delog(1M)	sysdiag(1M)	hpib(7)
isql(1)	uxgen(1)	disksecn(1M)	a.out(4)	

Series 800 HPIMAGE(3X) Non-Standard Commands

hpibegin(3X)	hpifindset(3X)	hpiundo(3X)	chpierror(3X)	chpiopen(3X)
hpiclose(3X)	hpiget(3X)	hpiupdate(3X)	chpifind(3X)	chpiput(3X)
hpicontrol(3X)	hpiinfo(3X)	chpibegin(3X)	chpifindset(3X)	chpiundo(3X)
hpidelete(3X)	hpilock(3X)	chpiclose(3X)	chpiget(3X)	chpiupdate(3X)
hpiend(3X)	hpimemo(3X)	chpicontrol(3X)	chpiinfo(3X)	
hpierror(3X)	hpiopen(3X)	chpidelete(3X)	chpilock(3X)	
hpifind(3X)	hpiput(3X)	chpiend(3X)	chpimemo(3X)	

TABLE OF CONTENTS

VOLUME 1

1. Commands

Table of Contents

9. Glossary

VOLUME 2

1M. System Maintenance Utilities

Table of Contents

2. System Calls

3. Subroutines

Table of Contents

Table of Contents

4. File Formats

5. Miscellaneous Facilities

6. Games

No games are currently supported.

7. Special Files

9. Glossary

The glossary is located in Volume 1 after Section 1.

NAME

intro – introduction to command utilities and application programs

DESCRIPTION

This section describes commands accessible by users, as opposed to system calls in Section (2), or subroutines in Section (3), which are accessible by user programs.

Command Syntax

Unless otherwise noted, commands described in this section accept options and other arguments according to the following syntax:

name [*option*(*s*)] [*cmdarg*(*s*)]

where:

name	The name of an executable file.
option	– *noargletter*(*s*) or, – *argletter*<>*optarg* where <> is optional white space.
noargletter	A single letter representing an option without an argument.
argletter	A single letter representing an option requiring an argument.
optarg	Argument (character string) satisfying preceding *argletter*.
cmdarg	Path name (or other command argument) *not* beginning with – or, – by itself indicating the standard input.

DIAGNOSTICS

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

WARNINGS

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

SEE ALSO

getopt(1), exit(2), wait(2), getopt(3C), hier(5).
The introduction to this manual.

NAME

acctcom – search and print process accounting file(s)

SYNOPSIS

acctcom [[options][file]] . . .

DESCRIPTION

Acctcom reads *file*, the standard input, or **/usr/adm/pacct**, in the form described by *acct*(4) and writes selected records to the standard output. Each record represents the execution of one process. The output shows the **COMMAND NAME, USER, TTYNAME, START TIME, END TIME, REAL (SEC), CPU (SEC), MEAN SIZE(K)**, and optionally, **F** (the *fork*/*exec* flag: **1** for *fork* without *exec*), **STAT** (the system exit status), **HOG FACTOR, KCORE MIN, CPU FACTOR, CHARS TRNSFD**, and **BLOCKS READ** (total blocks read and written).

The command name is prepended with a # if it was executed with *super-user* privileges. If a process is not associated with a known terminal, a **?** is printed in the **TTYNAME** field.

If no *files* are specified, and if the standard input is associated with a terminal or **/dev/null** (as is the case when using **&** in the shell), **/usr/adm/pacct** is read; otherwise, the standard input is read.

If any *file* arguments are given, they are read in their respective order. Each file is normally read forward, i.e., in chronological order by process completion time. The file **/usr/adm/pacct** is usually the current file to be examined; a busy system may need several such files of which all but the current file are found in **/usr/adm/pacct***?*. The *options* are:

–a	Show some average statistics about the processes selected. The statistics will be printed after the output records.
–b	Read backwards, showing latest commands first. This *option* has no effect when the standard input is read.
–f	Print the *fork*/*exec* flag and system exit status columns in the output.
–h	Instead of mean memory size, show the fraction of total available CPU time consumed by the process during its execution. This "hog factor" is computed as: (total CPU time)/(elapsed time).
–i	Print columns containing the I/O counts in the output.
–k	Instead of memory size, show total kcore-minutes.
–m	Show mean core size (the default).
–r	Show CPU factor (user time/(system-time + user-time).
–t	Show separate system and user CPU times.
–v	Exclude column headings from the output.
–l *line*	Show only processes belonging to terminal **/dev/***line*.
–u *user*	Show only processes belonging to *user* that may be specified by: a user ID, a login name that is then converted to a user ID, a # which designates only those processes executed with *super-user* privileges, or **?** which designates only those processes associated with unknown user IDs.
–g *group*	Show only processes belonging to *group*. The *group* may be designated by either the group ID or group name.
–s *time*	Select processes existing at or after *time*, given in the format *hr* [:*min* [:*sec*]].
–e *time*	Select processes existing at or before *time*.
–S *time*	Select processes starting at or after *time*.
–E *time*	Select processes ending at or before *time*. Using the same *time* for both **–S** and **–E** shows the processes that existed at *time*.
–n *pattern*	Show only commands matching *pattern* that may be a regular expression as in *ed*(1) except that + means one or more occurrences.
–q	Do not print any output records, just print the average statistics as with the **–a** option.

–o *ofile* Copy selected process records in the input data format to *ofile*; supress standard output printing.

–H *factor* Show only processes that exceed *factor*, where factor is the "hog factor" as explained in option **–h** above.

–O *time* Show only those processes with operating system CPU time that exceeds *time*.

–C *sec* Show only processes with total CPU time, system plus user, exceeding *sec* seconds.

–I *chars* Show only processes transferring more characters than the cut-off number given by *chars*.

Listing options together has the effect of a logical *and*.

FILES

/etc/group
/usr/adm/pacct
/etc/passwd

SEE ALSO

acct(1M), acctcms(1M), acctcon(1M), acctmerg(1M), acctprc(1M), acctsh(1M), fwtmp(1M), ps(1), runacct(1M), su(1), acct(2), acct(4), utmp(4).

BUGS

Acctcom only reports on processes that have terminated; use *ps*(1) for active processes. If *time* exceeds the present time, then *time* is interpreted as occurring on the previous day.

NAME

adb – debugger

SYNOPSIS

adb [**–w**] [–Idir] [objfil [corfil]]

DESCRIPTION

Adb is a general purpose debugging program sensitive to the underlying architecture of the processor upon which it is running. 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*. The **–I** option specifies a directory where files to be read with $< or $<< (see below) will be sought; the default is */usr/lib/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** below.

Expressions

.	The value of *dot*.
+	The value of *dot* incremented by the current increment.
^	The value of *dot* decremented by the current increment.
"	The last *address* typed.
integer	A number. The prefixes 0o and 0O (zero oh) force interpretation in octal radix; the prefixes 0t, 0T, 0d and 0D force interpretation in decimal radix; the prefixes 0x and 0X force interpretation in hexadecimal radix. Thus 0o20 = 0t16 = 0x10 = sixteen. If no prefix appears, then the *default radix* is used; see the $d command. The radix is initialized to the base normally used in the assembly language for the processor involved. Note that a hexadecimal number whose most significant digit would otherwise be an alphabetic character must have a 0x (or 0X) prefix (or a leading zero if the default radix is hexadecimal).
integer.fraction	A 32-bit floating point number.
'cccc'	The ASCII value of up to 4 characters. A \ 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 named by single letters or digits, see **Variables** below. If *name* is a register name then the value of the register is obtained from the system header in *corfil* (before the subprocess is initiated) or from the subprocess' user area. The register names are implementation dependent; see the $r command.

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 prefixed to *symbol* if needed.

_ *symbol* If the compiler prefixes a _ to an external symbol, 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*.

The monadic operators are:

**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.

The 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 can take format specifiers. (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. If a subprocess has been initiated, *address* references a location in the subprocess' address space instead of *objfil*.

/f Locations starting at *address* in *corfil* are printed according to the format *f* and *dot* is incremented as for ?. If a subprocess has been initiated, *address* references a location in the subprocess' address space instead of *corfil*.

=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 an 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. First, the sign bit is discarded, then character values 000 to 040 are printed as @ followed by the corresponding character in the range 0100 to 0140. The character @ is printed as @@.
s *n*	Print the addressed characters until a zero character is reached.
S *n*	Print a string using the @ escape convention. The value *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 machine instructions. The value of *n* is the number of bytes occupied by the instruction.
a 0	Print the value of *dot* in symbolic form.
p *n*	Print the addressed value in symbolic form. The value of *n* is a machine dependent constant.
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. New-line can also be used to repeat a **:s** or **:c** command; any arguments to the previous command, however, are lost.

[?/]**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 **!**.

$*modifier* The available **$** commands are:

- **$<***f* Read commands from the file *f*. If this command is executed in a file, further commands in the file are not seen. If a *count* is given, and is zero, the command will be ignored. The value of the count will be placed in variable *9* before the first command in *f* is executed.
- **$<<***f* Similar to **$<** except it can be used in a file of commands without causing the file to be closed. Variable *9* is saved during the execution of this command, and restored when it completes. Only five **$<<** files can be open at once.
- **$>***f* Send output to the file *f*, which is created if it does not exist.
- **$r** Print the general registers and the instruction addressed by the process counter. *Dot* is set to the process counter contents.
- **$f** Print the floating point registers in an appropriate machine-dependent manner.
- **$b** Print all breakpoints and their associated counts and commands.
- **$c** C stack backtrace. If *address* is given then it is taken as the address of the current frame (instead of the normal stack frame pointer). If *count* is given then only the first *count* frames are printed.
- **$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*. The default is system dependent.
- **$o** The default for all integers input is octal.
- **$d** Set the default radix to *address* and report the new value. Note that *address* is interpreted in the (old) current radix. Thus 10$d never changes the default radix. To make decimal the default radix, use 0t10$d.
- **$x** The default for all integers input is hexadecimal.
- **$q** Exit from *adb*.
- **$v** Print all non-zero variables in the current radix.
- **$m** Print the address map.
- **$**new-line print the process id and register values.

:*modifier* The available **:** commands, which manage subprocesses, 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*** will 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. The value *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. If there is no current subprocess then *objfil* is run as a subprocess as for **:r**. In this case no signal can be sent; the remainder of the line is treated as arguments to the subprocess.

: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.

9 The count on the last **$<** command.

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

b The base address of the data segment.

d The data segment size.

e The entry point.

m The "magic" number as defined in *magic.h*.

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, see **$m**. Each mapping is represented by two triples (*b1, e1, f1*) and (*b2, e2, f2*).

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. In some cases (e.g., for programs with separated I and D space) the two segments for a file may overlap. If a **?** or **/** is followed by an ***** then 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.

RETURNS

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

HARDWARE DEPENDENCIES

Series 200, 300

The **-I** option is not currently supported.

The **I** format prints machine instructions, like **i**, except that immediate constants are printed in decimal.

The command **$n** is provided to set the number of significant digits for floating point dumps to *address*.

Variable 9 is not updated for the **$<** command, and the **$<<** command is not supported.

Series 800

The **$f** command will print floating point registers as 32-bit single precision and **$F** will print these registers as 64-bit doubles.

$R will print all registers available to *adb* users.

The **:x** and **:S** commands are not currently supported.

A **–k** option is provided that allows virtual-to-physical translations, which is useful for kernel debugging. In this case, *core* should be an HP-UX crash dump or */dev/mem*.

When *adb* is invoked with this option, it sets up the context of the currently running process using space registers four through seven. A user specified address is dereferenced by combining it with the appropriate space register depending on which quadrant the 32 bit address lies. The **$p** command is provided to change the current context. The address argument is the address of the corresponding process structure.

When the current radix is not (decimal) ten, the **-k** option allows *adb* to support the notion of long pointers or addresses in the form <space>.<offset>. Once a space is specified, all subsequent addresses are dereferenced using that space until another long address is entered. If a space equal to (hexadecimal) 0xffffffff is used, the previous context is back in effect and *adb* reverts to using space registers four through seven to dereference 32 bit addresses.

AUTHOR

Adb was developed by AT&T and HP.

FILES

a.out
core
/dev/mem

/dev/swap

SEE ALSO

ptrace(2), a.out(4), core(4).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

adjust – simple text formatter

SYNOPSIS

adjust [**–bcjr**] [**–m** *column*] [**–t** *tabsize*] [*files...*]

DESCRIPTION

This command is a simple text formatter for filling, centering, left and right justifying, or right-justifying text paragraphs, designed for interactive use. It reads the concatenation of input files (or standard input if none are given) and produces on standard output a formatted version of its input, with each paragraph formatted separately. If "–" is given as an input filename, *adjust* reads standard input at that point. (You can use "--" as an argument to separate "-" from options.)

Adjust reads text from input lines as a series of words separated by blanks, tabs, or newlines. Text lines are grouped into paragraphs separated by blank lines. By default, text is carried over to the output subject only to simple filling (see below), with a right margin of 72, and leading blanks converted to tabs where possible.

Options are:

–b Do not convert leading blanks to tabs on output. Thus there are no tabs in the output.

–c Center text on each line. Lines are pre- and post-processed, but no filling is done.

–j Justify text. After filling, insert blanks in each line, except the last line of each paragraph, as needed to right-justify it, while keeping the justified left margin.

–r After filling text, adjust the indentation of each line for a smooth right margin (ragged left margin).

–m*column*

Set the right fill margin to the given column number, instead of 72. Text is filled, and optionally right justified, so that no output line extends beyond this column (if possible). If **–m0** is given, the current right margin of the first line of each paragraph is used for that and all subsequent lines in the paragraph.

By default, text is centered on column 40. With **–c**, the **–m** option sets the middle column of the centering "window", but **–m0** auto-sets the right side as before (which then determines the center of the "window").

–t*tabsize*

Set the tab size to other than the default (eight columns).

Only one of the **–c**, **–j**, and **–r** options is allowed at once.

Details

Before doing anything else to a line of input text, *adjust* first handles backspaces, rubbing out preceding characters in the usual way. Next it ignores all non-printable characters except tab. Then it expands all tabs to blanks.

For simple text filling, the first word of the first line of each paragraph is indented the same amount as in the input line. Each word is then carried to the output followed by one blank. "Words" ending in <terminal>[<quote>][<close>] are followed by two blanks, where <terminal> is any of ".:?!", <quote> is a single or double quote, and <close> is any of ")]}", e.g.:

end. of? sentence.' sorts!" of.) words?"]

The program starts a new output line whenever adding a word (other than the first one) to the current line would pass the right margin.

Adjust understands indented first lines of paragraphs (like this one) when filling. The second and subsequent lines of each paragraph are indented the same amount as the second line of the paragraph in the input, if there is a second line, else the same as the first line.

* *Adjust* has a rudimentary understanding of tagged paragraphs (like this one) when filling. If the second line of a paragraph is indented more than the first, and the first line has a word beginning at the same indentation as the second line, the column positions of the tag words (prior to that one) are "frozen" (not altered).

Tag words are passed through without change of column position, even if they extend beyond the right margin. The rest of the line is filled or right-justified from the position of the first non-tag word.

When **–j** is given, *adjust* uses an intelligent algorithm to insert blanks in output lines where they are most needed, until the lines extend to the right margin. First, all one-blank word separators are examined. One blank is added first to those separators with the most total letters in the words on both sides. If all one-blank separators are increased to two blanks, and more blanks must be inserted, it repeats the algorithm, this time with two-blank separators, and so on.

Output line indentation is held to one less than the right margin. If a single word is larger than the line size (right margin minus indentation), that word appears on a line by itself, properly indented, and extends beyond the right margin. However, if **–r** is used, such words are still right-justified, if possible.

DIAGNOSTICS

Adjust complains to standard error and later returns a non-zero value if any input file cannot be opened (it skips the file). It does the same (but quits immediately) if the argument of **–m** or **–t** is out of range, or if the program is improperly invoked.

Input lines longer than BUFSIZ are silently split (before tab expansion) or truncated (afterwards). Lines too wide to center begin in column 1 (no leading blanks).

WARNINGS

This program is designed to be simple and fast. It does not recognize backslash to escape white space or anything else. It does not recognize tagged paragraphs where the tag is on a line by itself. It knows that lines end in newline or null, and how to deal with tabs and backspaces, but it does not do anything special with other characters like form feed (they are just ignored). For complex operations, the standard text processors are likely to be more appropriate.

This program could be implemented instead as a set of independent programs, fill, center, and justify (with **–r** option). However, this would be much less efficient in actual use, especially given the program's special knowledge of tagged paragraphs and last lines of paragraphs.

These options were considered but not added, because the creeping featurism had to stop somewhere, before this program became another *nroff*(1):

–h Hyphenate. Allows the program to break and join words at existing hyphens (only). Words are broken across lines, at single hyphens surrounded by non-whitespace characters. Likewise, a word ending in a single hyphen, followed by whitespace, followed by a non-hyphen character, is joined to the next word without whitespace if needed.

–n Nofill. Only allowed with **–j** or **–r**, it inhibits filling, i.e. words are not moved from one line to another. Thus existing text can be left/right or right justified without being otherwise modified. (Note that **–n** is always in effect for **–c**, centering.)

EXAMPLES

This command is useful for filtering text while in *vi*(1). For example,

```
!}adjust
```

reformats the rest of the current paragraph (from the current line down), evening the lines.

You can give the **vi** command:

```
:map ^X {!}adjust -j^V^M
```

(where "^" denotes control characters) to set up a useful "finger macro". Then typing ^X will reformat the entire current paragraph.

Note that **adjust -m1** is a simple way to break text into separate words, without white space, except for tagged paragraphs tags.

AUTHOR

Adjust was developed by HP.

SEE ALSO

nroff(1).

INTERNATIONAL SUPPORT

8-bit data and filenames.

NAME

admin – create and administer SCCS files

SYNOPSIS

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

DESCRIPTION

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

If a directory is named, *admin* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the 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*str* 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. If a value is supplied, the keywords must exactly match the given string, however the string must contain a keyword, and no embedded newlines.

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

l*list* A *list* of releases to which deltas 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* *Module* 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 %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 *logins*, or numerical group IDs, as desired may be on the list simultaneously. If the list of users is empty, then anyone may add deltas. If *login* or group ID is preceded by a ! they are to be denied permission to make deltas.

–e*login* A *login* name, or 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*(4)), 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(4).

DIAGNOSTICS

Use *help*(1) for explanations.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

ar – archive and library maintainer for portable archives

SYNOPSIS

ar key [posname] afile [name] ...

DESCRIPTION

Ar maintains groups of files combined into a single archive file. Its main use is to create and update library files as used by the link editor. It can be used, though, for any similar purpose. The magic string and the file headers used by *ar* consist of printable ASCII characters. If an archive is composed of printable files, the entire archive is printable.

Individual files are inserted without conversion into the archive file. When *ar* creates an archive, it creates headers in a format that is portable across all machines. The portable archive format and structure is described in detail in *ar*(4). The archive symbol table (described in *ar*(4)) is used by the link editor (*ld*(1)) to effect multiple passes over libraries of object files in an efficient manner. An archive symbol table is only created and maintained by *ar* when there is at least one object file in the archive. The archive symbol table is in a specially named file which is always the first file in the archive. This file is never mentioned or accessible to the user. Whenever the *ar*(1) command is used to create or update the contents of such an archive, the symbol table is rebuilt. The **s** option described below will force the symbol table to be rebuilt.

Key must be present, and is an optional –, followed by one character from the set **drqtpmx**, optionally concatenated with one or more of **vuaibcls**. *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 files 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 dates of modification later than the archive files are replaced. If an optional positioning character from the set **abi** is used, the *posname* argument must be present and specifies that new 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 *names*, *ar* will create an empty archive file whose only contents is the archive header (see *ar*(4)).

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:

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

v Verbose. Give a verbose file-by-file description of the making of a new archive file from the old archive and the constituent files. When used with **t**, it gives a long listing of all information about the files. When used with 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. Place temporary files in the local current working directory, rather than in the directory specified by the environment variable **TMPDIR** or in the default directory **/tmp**. Only the **d**, **m**, **r** and **s** 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**, **s**
p:	**v**, **s**
m:	**v**, **l**, and **a** | **b** | **i**
x:	**v**, **s**

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 and f3 follows f2, the commands:

ar ma f1 newlib.a f2 f3
ar ma f2 newlib.a f3
ar r 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 f2 and f3 after f1 in newlib.a", thus creating the order:

f1 f3 f2 f4

Note that the relative order of f2 and f3 has not changed. The second command says "move f2 after f3 in newlib.a", creating the order:

f1 f2 f3 f4

The third command then replaces the files f2 and f3. Since the files f2 and f3 both already existed in the archive, this sequence of commands could not be simply replaced by:

ar ra f1 newlib.a f2 f3

because the previous position and relative 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: f3´ f2´ f1 f4

FILES

/tmp/ar* temporaries

SEE ALSO

ld(1), lorder(1), strip(1), tmpnam(3S), a.out(4), ar(4).

VARIABLES

TMPDIR Where temporary files are kept.

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 can 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.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

arcv – convert archives to new format

SYNOPSIS

arcv file ...

DESCRIPTION

Arcv converts archive files (see *ar*(1), and *ar*(5)) from a pre–HP-UX 5.0 format to the HP-UX 5.0 portable archive format. The conversion is done in place, and the command refuses to alter a file not in old archive format.

Old archives are marked with a magic number of 0177545 at the start; new archives have a first line "!<arch>".

FILES

/tmp/arc*

SEE ALSO

ar(1), ar(5).

NAME

as – assembler

SYNOPSIS

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

REMARKS

This is a generic page for a machine-dependent assembler. A specific page will be provided for each assembler. Not all HP–UX systems provide an assembler.

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. 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(4).
The assembler reference for each machine.

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 displayed on *stderr* together with the line number and the file name in which it occurred.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

Series 300 Only starting at HP-UX Release 5.15

NAME

as – assembler for MC68000, MC68010, and MC68020

SYNOPSIS

as [–L] [–N**s***n*] [–**m**] [–**d**] [–**w**] [–**o** objfile] [file]
as10 [–L] [–N**s***n*] [–**m**] [–**d**] [–**w**] [–**o** objfile] [file]
as20 [–L] [–N**s***n*] [–**m**] [–**d**] [–**w**] [–**o** objfile] [file]

Remarks:

This version of *as* is implemented on Series 300 only.

DESCRIPTION

As assembles the named *file* (which usually has a **.s** suffix as in *my_prog.s*). If *file* is not specified or if – is given, standard input is used instead. *As* is linked to *as10* or *as20* at installation time to match the system processor model. This link determines which of the two assemblers is the default for *as*. If *as10* is invoked (separately or through *as*), MC68010 object code is produced (compatible with both the MC68010 and MC68020). If, on the other hand, *as20* is invoked, MC68020 object code is produced (some MC68020 processor instructions are not supported by the MC68010).

All undefined symbols in the assembly are treated as global.

Options are as follows:

–L Generates entries in the linker symbol table for local as well as global symbols. Normally, only global and undefined symbols are entered into the table. This option is useful when using *adb*(1) to debug assembly language programs.

–N**s***n*
Changes the size of the user symbol table to accommodate up to *n* elements. Default is 4000 entries.

–**m** Processes the input file using the *m4*(1) macro preprocessor before assembling it.

–**d** When used with *as20* assembler, *as20* generates short-displacement forms for MC68010-compatible syntaxes, including forward references. This option is ignored by *as10*.

–**o** Causes output object code to be placed in file *objfile*. If –**o** is not specified and the source file is read from *stdin*, the object file is written to *a.out*. If –**o** is not specified and the source file is not *stdin*, the object file is written to a file whose name is created by removing the **.s** suffix (if present) from the basename of filename *file*, then adding a **.o** suffix to the base filename. The object **.o** file is placed in the current directory.

–**w** Suppresses warning messages (errors are not suppressed).

FILES

/usr/tmp/*	temporary files (can be changed by using TMPDIR. See *tmpnam*(3S)).
file.o	object file

SEE ALSO

adb(1), astrn(1), atrans(1), ld(1), m4(1), nm(1), a.out(5).
HP-UX Assembler Reference and ADB Tutorial for Series 200/300 Computers.

DIAGNOSTICS

If the name chosen for the output file is of the form ***.c** or ***.s**, the assembler issues an appropriate complaint and quits. When syntactic or semantic errors occur, a single-line diagnostic is produced, including the line number and file name in which it occurred.

RESTRICTIONS/CAVEATS

Expressions cannot have more than one forward-referenced symbol, except for the special form <symbol>-<symbol>.

WARNINGS

If the **-m** option is used, keywords for *m4* cannot be used as symbols in the input file because *m4* cannot determine which are assembler symbols and which are real *m4* macros.

BUGS

The displacement value for the **movp** instruction must be a first-pass absolute 16-bit value.

NOTES

Wherever possible, the assembler should be accessed through a compilation system interface program, such as *cc*(1).

Both assemblers support the complete MC68000 instruction set. However, if you are writing code for an MC68000 processor, you must limit instructions and program structures to those supported by the microprocessor. Using instructions supported by MC68010 or MC68020 processors on an MC68000 will cause an illegal instruction trap during program execution, but may not produce an error during program assembly and loading.

NAME

as – assembler (Precision Architecture)

SYNOPSIS

as [[option] ... [file] ...] ...

DESCRIPTION

As assembles the named *file*, or the standard input if no file name is specified. The optional argument –l may be used to obtain an assembly listing with offsets.

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.

The output of *as* is not optimized. *As* creates relocatable object files which must be processed by *ld* to be made executable.

Cc assembles *.s* files together with *pcc_prefix.s* and subsequently invokes *ld*.

Options

As recognizes the following options.

–e	An unlimited number of errors will be tolerated before the assembly process is abandoned. Normally, only a hundred errors are allowed.
–f	Procedures by default will be callers of other procedures. The normal default is that procedures do not call other procedures.
–l	Listing to standard output is made of the program after assembly. This listing shows offsets of instructions and actual values for fields.
–o *outfile*	Produce an output object file by the name *outfile* instead of using the default *.o* suffix.
–s	The output file will have suffix *.ss* and be of a format suitable for conversion to the ROM burning programs.
–u	Unwind descriptors will not be created. In order to avoid the need for .CALLINFO, it must also be the case that .ENTER and .LEAVE have not been used.
–v *xrfile*	Provides the name of a file to which cross reference data is written.

DIAGNOSTICS

When syntactic or semantic errors occur, a single-line diagnostic is displayed on *stderr* together with the line number and the file name in which it occurred.

WARNINGS

As does not do macro processing.

Trailing operands (except for a pc–relative branch displacement) may be omitted and default to zero. Trailing commas may also be omitted. Leading commas are ignored.

FILES

/lib/pcc_prefix.s	space and register definitions
/usr/include/hard_reg.h	hardware register equates
/usr/include/soft_reg.h	follows calling convention
/usr/include/std_space.h	space and subspace declarations
/lib/as_msgs.cat	error message catalog
file.o	object file

SEE ALSO

cc(1), ld(1), adb(1), nm(1).

Precision Architecture Assembler Technical Reference Manual.

NAME

asa – interpret ASA carriage control characters

SYNOPSIS

asa [files]

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

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

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

```
asa file
```

SEE ALSO

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

INTERNATIONAL-SUPPORT

8- and 16-bit data, 8-bit filenames.

Series 300 Only

NAME

astrn – translate assembly language

SYNOPSIS

astrn [filename]

Remarks:

Astrn is implemented on the Series 200/300 only.

DESCRIPTION

Astrn translates an assembly language *source file* from previous HP-UX **Series 200/300** assembly language syntax to new **Series 300** HP-UX assembly language syntax. If no *filename* is given, input is assumed to come from **stdin.**

If an opcode is not recognized, a warning message is given and the entire line is passed through unchanged. For any syntax error detected such that translation cannot continue, *astrn* reports an error and translation terminates.

Lines longer than 132 characters are truncated to 132 characters.

For a line beginning with '*' (indicating a comment), the '*' is translated to a '#' but is preceded by a blank to allow preprocessing with *cpp(1)*.

Absolute displacements off the program counter cannot be guaranteed to translate correctly. Any line referencing the program counter will be flagged by a warning message.

Certain capabilities supported on the old assembler are not accepted by the new assembler. These include:

The *alias* and *include* pseudo-ops are not supported. An error message is given and translation terminates.

The new assembler restricts expressions involving forward references for which *astrn* makes no check. Such references may involve only a single symbol, a symbol plus or minus an absolute expression, or the subtraction of two symbols.

The characters '$', '@', '?', and '\177' are no longer accepted as valid identifier characters. These are translated to 'S', 'A', 'Q', and 'D' respectively and a warning is issued.

Span-dependent branches **j***cc* are translated to **b***cc.w*.

An identifier equated to a register name will be translated but the assembler will report an error.

Local labels are translated to a concatenation of the nearest previous ordinary label and the local label itself. This includes changing the '$' to a 'S'.

SEE ALSO

as(1), atrans(1).

NAME

at, batch – execute commands at a later time

SYNOPSIS

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

batch

DESCRIPTION

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

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

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

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

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

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

Thus legitimate commands include:

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

At and *batch* write the job number and schedule time to standard error.

Batch submits a batch job. It is almost equivalent to "at now", but not quite. For one, it goes into a different queue. For another, "at now" will respond with the error message too late.

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

EXAMPLES

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

be useful to redirect standard output.

This sequence can be used at a terminal:

```
batch
nroff filename >outfile
<control-D> (hold down 'control' and depress 'D')
```

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

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

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

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

FILES

/usr/lib/cron	main cron directory
/usr/lib/cron/at.allow	list of allowed users
/usr/lib/cron/at.deny	list of denied users
/usr/spool/cron/atjobs	spool area
/usr/lib/cron/queuedefs	scheduling information

SEE ALSO

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

DIAGNOSTICS

Complains about various syntax errors and times out of range.

INTERNATIONAL SUPPORT

at: 8- and 16-bit data, 8-bit filenames.

Series 500 Only

NAME

aterm – general purpose asynchronous terminal emulation

SYNOPSIS

aterm configfile

Native Language Support:
8–bit data.

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 HP-UX system;
uucp(1C) for file transfers with other HP-UX systems.

HP-UX Network Communications Guide.

BUGS

Does not work with 6–channel multiplexer.

Series 300 Only

NAME

atrans – translate assembly language

SYNOPSIS

atrans [**–n**] [filename]

Remarks:

Atrans is implemented on the Series 200/300 only. This page describes Series 300 HP-UX starting at Release 5.15.

DESCRIPTION

Atrans translates an assembly language *source file* from **Series 200/300** Pascal workstation assembly language syntax to **Series 300** HP-UX assembly language syntax. If no *filename* is given, input is assumed to come from **stdin.**

If an opcode is not recognized, the entire line is passed through unchanged. For any syntax error detected such that a line cannot be translated, *atrans* issues an error message.

Lines longer than 132 characters are truncated to 132 characters.

Absolute displacements off the program counter cannot be guaranteed to translate correctly. Any line referencing the program counter will be flagged by a warning message.

The HP-UX assembler restricts expressions involving forward references for which *atrans* makes no check. Such references may involve only a single symbol, a symbol plus or minus an absolute expression, or the subtraction of two symbols.

The characters '$' and '@' are not accepted as valid identifier characters on the HP-UX assembler. These are translated to 'S' and 'A' respectively and a warning is issued.

Lines containing the following list of **Series 200/300** Pascal workstation pseudo-ops have no parallel in **Series 300** 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*, *include*, or *src* pseudo-ops are translated as comment lines, and a warning is printed stating these are not supported by the *Series 300* HP-UX assembler.

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

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 300** HP-UX assembler. These pseudo-ops are: *com, lmode, org, rorg, rmode, smode, start.*

When specifying certain addressing modes, the Pascal workstation assembler may allow operands to appear out of order, whereas the HP-UX assembler does not. *Atrans* does not rearrange these into proper order.

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

SEE ALSO

as(1), astrn(1).

NAME

awk – text pattern scanning and processing language

SYNOPSIS

awk [**–Fc**] [prog] [parameters] [files]

DESCRIPTION

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

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

Files are read in order; if there are no files, the standard input is read. The 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 ("); single quotes (**'fP) are not recognized.**

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 }
```

Print file, filling in page numbers starting at 5:

```
/Page/ { $2 = n++; }
       { print }
```

command line: awk -f program n=5 input

SEE ALSO

grep(1), lex(1), sed(1), malloc(3X).

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.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

banner – make posters in large letters

SYNOPSIS

banner strings

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.

SEE ALSO

echo(1).

NAME

basename, dirname extract portions of path names

SYNOPSIS

basename string [suffix]
dirname string

DESCRIPTION

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

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

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`
```

RETURNS

Both commands return 0 for success. Both commands return 1 when given no arguments.

SEE ALSO

expr(1), sh(1).

INTERNATIONAL SUPPORT

8-bit filenames.

NAME

basic – Technical BASIC interpreter

SYNOPSIS

basic[-t]

Remarks:

This command requires installation of optional Technical BASIC software (not included with the standard HP-UX operating system) before it can be used.

DESCRIPTION

This command invokes the HP-UX Technical BASIC interpreter which can be used to execute BASIC commands or run BASIC programs.

The BASIC **SHELL** command is used when you need to temporarily exit the BASIC environment and spawn a new Bourne shell, from which you can execute any number of HP-UX commands. To terminate the shell and return to BASIC, type CTRL-D.

The following option is recognized:

-t causes the BASIC interpreter to operate in non-line-oriented mode. (Using this option only makes sense when running BASIC on a line-oriented terminal, since all other consoles and terminals will automatically run in non-line-oriented mode. See "line-oriented terminal" in the Glossary of the *HP-UX Technical BASIC Reference Manual* if you are not sure whether or not your terminal is line-oriented.)

There are two situations in which this mode is useful:

1. When you are using BASIC program lines or commands that exceed the width of a single line on the terminal screen (usually 80 characters). Refer to the "Line Length" discussion in the Introduction section of the *HP-UX Technical BASIC Reference Manual* for further information about entering lines longer than the width of the screen.
2. When data-overrun errors occur during relatively high system use that are caused by a serial interface that has only a single-character buffer. Such interfaces lose occasional characters because they have insufficient buffer character space. This type of error does not occur when using interfaces that are equipped with multiple-character buffers such as the Series 200/300 Datacomm Interface or Series 500 ASI card.

FILES

/usr/bin/basic	the BASIC interpreter.
/usr/bin/makebin_c	a shell script used for creating C binaries (binaries are routines that are written in another language but which can be called from BASIC).
/usr/lib/bcrt0.o	used when creating C binaries.
/usr/bin/makebin_p	a shell script used for creating Pascal binaries.
/usr/lib/bprt0.o	a file that is used when creating Pascal binaries.
/usr/bin/makebin_f	a shell script used for creating FORTRAN binaries.
/usr/lib/bfrt0.o	a file that is used when creating FORTRAN binaries.
/usr/lib/libb.a	(Series 200/300 only) a library that is used instead of **libc.a** when creating binaries.
*/usr/lib/examples/basic/get_started/**	demonstration programs that are discussed in the *HP-UX Technical BASIC Getting Started* manual.

SEE ALSO

HP-UX Technical BASIC Getting Started Manual
HP-UX Technical BASIC Programming Guide
HP-UX Technical BASIC I/O Programing Guide
HP-UX Technical BASIC Reference Manual
HP-UX Technical BASIC Implementation Specifics
HP-UX Technical BASIC Quick Reference

BUGS

Depending on system load, some characters may be missing from the start-up message (*Basic ready 1.0*) or termination message (*Exiting Basic*). This should not happen on single-user systems or on multi-user systems where only one person is currently using the system.

NAME

bc arbitrary-precision arithmetic language

SYNOPSIS

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

DESCRIPTION

Bc is an interactive processor for a language that resembles C but provides unlimited precision arithmetic. It takes input from any files given, then reads the standard input. The 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 pre-defined. As a side-effect, the scale factor is set.

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

Other operands
arbitrarily long numbers with optional sign and decimal point.
(E)
sqrt (E)
length (E) number of significant decimal digits
scale (E) number of digits right of decimal point
L (E , ... , E)
Strings of ASCII characters enclosed in quotes (").

Arithmetic operators (yield an E as a result)
+ - * / % ^ (% is remainder (not mod, see below); ^ is power)
++ -- (prefix and postfix; apply to names)
= =+ =- =* =/ =% =^

Relational operators (yield an R when used as **E** *op* **E**).
== <= >= != < >

Statements
E
{ S ; ... ; S }
if (R) S
while (R) S
for (E ; R ; E) S
null statement
break
quit

Function definitions

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

Functions 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.

The % operator yields the remainder at the current scale, not the integer modulus. Thus, at scale 1, **7 % 3** is .1 (one tenth), not 1. This is because (at scale 1) **7 / 3** is 2.3 with .1 as the remainder.

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

SEE ALSO

bs(1), dc(1).

BUGS

There are 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**]

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.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

bfs – big file scanner

SYNOPSIS

bfs [–] name

DESCRIPTION

Bfs is (almost) like *ed*(1) except that it is read-only and processes much larger files. Files can be up to 1024K bytes (the maximum possible size) and 32K lines, with up to 512 characters, including new-line, per line. *Bfs* is usually more efficient than *ed* for scanning a file, since the file is not copied to a buffer. It is most useful for identifying sections of a large file where *csplit*(1) can be used to divide it into more manageable pieces for editing.

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

All address expressions described under *ed* are supported. In addition, regular expressions may be surrounded with two symbols besides / and ?: > indicates downward search without wrap-around, and < indicates upward search without wrap-around. Since *bfs* uses a different regular expression-matching routine from *ed*, the regular expressions accepted are slightly wider in scope (see *regcmp*(3X)). There is a slight difference in mark names: only the letters **a** through **z** may be used, and all 26 marks are remembered.

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

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

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

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

[. . .]**xb**/*regular expression*/*label*

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

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

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

xb/^/ label

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

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

xt *number*

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

xv[*digit*] [*spaces*] [*value*]

The variable name is the specified *digit* following the **xv**. **xv5100** or **xv5 100** both assign the value **100** to the variable **5**. **Xv61,100p** assigns the value **1,100p** to the variable **6**. To reference a variable, put a **%** in front of the variable name. For example, using the above assignments for variables **5** and **6**:

```
'
:
1,%5p
1,%5
%6
```

will all print the first 100 lines.

g/%5/p

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

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

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

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

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

xv7\'date

stores the value **!date** into variable **7**.

xbz *label*
xbn *label* These two commands will test the last saved *return code* from the execution of an HP-UX system command (**!***command*) for a zero or nonzero value, respectively, and cause a branch to the specified label. The two examples below both search for the next five lines containing the string **size**.

First example:

xv55

```
: l
/size/
xv5!expr %5 - 1
!if [ %5 != 0 ] ; then exit 2 ; fi
xbn l
```

Second Example:

```
xv45
: l
/size/
xv4!expr %4 - 1
!if [ %4 = 0 ] ; then exit 2 ; fi
xbz l
```

xc [*switch*]

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

DIAGNOSTICS

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

SEE ALSO

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

INTERNATIONAL SUPPORT

8-bit data and filenames.

NAME

bifchmod – change mode of a BIF file

SYNOPSIS

bifchmod mode device:file ...

DESCRIPTION

Bifchmod is intended to mimic *chmod*(1).

A BIF file name is recognized by the embedded colon (:) delimiter (see *bif*(4) for BIF file naming conventions).

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**, which is 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**; **t** only works with **u**.

EXAMPLES

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

bifchmod o–w file

bifchmod +x file

The next 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):

bifchmod 4555 file

bifchmod 644 file

The following two examples perform the same function, namely to give read, write, and execute permission to the owner and read and execute permissions to everybody else for the BIF file **/etc/script** on **/dev/rdsk/1s0**:

bifchmod a=rx,u+w /dev/rdsk/1s0:/etc/script

bifchmod 755 /dev/rdsk/1s0:/etc/script

AUTHOR

Bifchmod was developed by HP.

SEE ALSO

bif(4), chmod(1), chmod(2).

NAME

bifchown, bifchgrp – change file owner or group

SYNOPSIS

bifchown owner device:file ...
bifchgrp group device:file ...

DESCRIPTION

Bifchown and *bifchgrp* are intended to mimic *chown*(1) and *chgrp*(1).

A BIF file name is recognized by the embedded colon (:) delimiter (see ***bif***(4) for BIF file naming conventions).

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

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

EXAMPLES

The examples that follow assume that a BIF directory structure exists on the HP-UX device file **/dev/rdsk/1s0**.

The first example sets the owner of the BIF file **/users/abc/phone.num** to **adm**:

bifchown adm /dev/rdsk/1s0:/users/abc/phone.num

The second example sets the group ID of the BIF file **/tmp/b.date** to the decimal number **105**:

bifchgrp 105 /dev/rdsk/1s0:/tmp/b.date

AUTHOR

Bifchown was developed by HP.

FILES

/etc/passwd
/etc/group

SEE ALSO

bif(4), chown(1), group(4), passwd(4).

NAME

bifcp – copy to or from BIF files

SYNOPSIS

bifcp file1 file2
bifcp file1 [file2...] directory

DESCRIPTION

Bifcp is intended to mimic *cp*(1).

A BIF file name is recognized by the embedded colon (:) delimiter (see *bif*(4) for BIF file naming conventions).

Bifcp copies a BIF or HP-UX file to a BIF or HP-UX file, or list of files (HP-UX or BIF) to a directory. The last name on the argument list is the destination file or directory.

The file name '-' (dash) is interpreted to mean standard input or standard output, depending on its position in the argument list.

RETURNS

Bifcp returns exit code **0** if the file is copied successfully. Otherwise it prints a diagnostic and returns non-zero.

EXAMPLES

Copy the HP-UX file **abc** to the BIF file **x/y/z** within HP-UX device **/dev/rdsk/1s0**:

```
bifcp abc /dev/rdsk/1s0:x/y/z
```

Copy BIF file **/backup/log** within **/dev/rdsk/1s0** to HP-UX file **logcopy** within the current directory:

```
bifcp /dev/rdsk/1s0:/backup/log logcopy
```

Copy BIF file **archive** within HP-UX device **/dev/dsk/2s5** to standard output:

```
bifcp /dev/dsk/2s5:archive -
```

The following example copies the BIF files **/a**, **/b**, and **/c** to the HP-UX directory **/users/dave**:

```
sdfcp /dev/rdsk/2s3:/a /dev/rdsk/2s3:/b /dev/rdsk/2s3:/c /users/dave
```

The last example shows how you can implement a *cat*(1) program for concatenating BIF files using *bifcp* in a shell script:

```
if [ $# -lt 1 ]
then
        echo "Usage: bifcat file ..."
        exit 1
fi
for i in $*
do
        bifcp $i -
done
```

WARNINGS

Note that the media should **NOT** be mounted before using *bifcp*.

The '-' (stdio) notation does not work in some situations.

AUTHOR

Bifcp was developed by HP.

SEE ALSO

bif(4), cp(1).

NAME

biffind – find files in a BIF system

SYNOPSIS

biffind path-name-list expression

DESCRIPTION

Biffind is intended to mimic *find*(1).

A BIF file name is recognized by the embedded colon (:) delimiter (see *bif*(4) for BIF file naming conventions).

Biffind 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.

–name *pattern* True if *pattern* matches the current file name. Pattern may consist of ASCII characters as well as the meta characters:

'*' match all characters

'?' match any character

[...] match a range of characters.

–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.

–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 that satisfy the other primaries in the *expression*.

–inum *n* True if the file has inode number *n*.

EXAMPLES

To print the names of all files on the BIF volume **/dev/rdsk/2s0**:

biffind /dev/rdsk/2s0: -print

The following command finds all files in **/dev/dsk/1s3:/usr/lib** that are directories:

biffind /dev/dsk/1s3:/usr/lib -type d -print

Finally,

biffind /dev/rdsk/2s2:/users -type d -exec bifls -l {} ;

gives a long listing of every directory under **/users** on the device **/dev/rdsk/2s2**.

AUTHOR

Biffind was developed by HP.

FILES

/etc/passwd
/etc/group

SEE ALSO

bif(4), find(1).

NAME

bifls – list contents of BIF directories

SYNOPSIS

bifls [**-AadFilp**] [device:names...]
bifll [**-AadFilp**] [device:names...]

DESCRIPTION

Bifls is intended to mimic *ls*(1).

A BIF file name is recognized by the embedded colon (:) delimiter (see *bif*(4) for BIF file naming conventions).

For each directory named, *bifls* lists the contents of that directory; for each file named, *bifls* repeats its name and any other information requested.

If you are the super-user, *bifls* defaults to listing all files except . (current directory) and .. (parent directory). If invoked by the name *bifll*, the -l option is implied.

There are several options to *bifls*:

-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**.

-d If argument is a directory, list only its name; often used with **-l** to get the status of a directory.

-F List with indicator of file type: / means a directory, * means executable.

-i List the inode of a file or files.

-l List in long format, giving mode, number of links, owner, group, size in bytes, and time of last modification for each file.

-p Do not use **/etc/passwd** and **/etc/group** to interpret user and group ownership, but rather print out the numeric form.

EXAMPLES

The examples that follow assume that an BIF directory structure exists on the HP-UX device file **/dev/rdsk/1s0**.

The first example will list all the files in the root directory of the BIF directory structure:

bifls -a /dev/rdsk/1s0:

The second example gives (in long format) all the information about the BIF directory **/users/root** itself (but not the files in the directory):

bifls -ld /dev/rdsk/1s0:/users/root

WARNINGS

Remember, to obtain a listing of the BIF files on **/dev/rdsk/1s0**, you must not say **bifls /dev/rdsk/1s0** but you must include the colon, as in **bifls /dev/rdsk/1s0:**. If the colon is omitted, you get a listing of the HP-UX file **/dev/rdsk/1s0**, not its BIF contents.

AUTHOR

Bif was developed by HP.

FILES

/etc/passwd to get user ids.
/etc/group to get group ids.

SEE ALSO

bif(4), ls(1).

NAME

bifmkdir – make a BIF directory

SYNOPSIS

bifmkdir device:dirname ...

DESCRIPTION

Bifmkdir is intended to mimic *mkdir*(1).

A BIF file name is recognized by the embedded colon (:) delimiter (see *bif*(4) for BIF file naming conventions).

Bifmkdir creates specified directories in mode 777. The standard entries, . for the directory itself, and .. for its parent, are made automatically.

RETURNS

Bifmkdir returns exit code **0** if all directories were successfully made; otherwise, it prints a diagnostic and returns non-zero.

EXAMPLES

Create an empty subdirectory named sysmods under the directory **/usr/lib** on HP-UX device **/dev/dsk/2s0**:

bifmkdir /dev/dsk/2s0:/usr/lib/sysmods

AUTHOR

Bif was developed by HP.

SEE ALSO

bif(4), mkdir(1).

NAME

bifrm, bifrmdir – remove BIF files or directories

SYNOPSIS

bifrm [**–fri**] device:file ...

bifrmdir device:dir ...

DESCRIPTION

Bifrm and *bifrmdir* are intended to mimic *rm*(1) and *rmdir*(1).

A BIF file name is recognized by the embedded colon (:) delimiter (see *bif*(4) for BIF file naming conventions).

Bifrm 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.

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 *bifrm* to recursively delete the entire contents of a directory, and then the directory itself. *Bifrm* can recursively delete up to 17 levels of directories.

–i causes *bifrm* to ask whether or not to delete each file. If **–r** is also specified, *bifrm* asks whether to examine each directory encountered.

Bifrmdir removes entries for the named directories, which must be empty.

EXAMPLES

The following examples assume that an BIF directory structure exists on the HP-UX device file **/dev/rdsk/1s0**.

The first example recursively combs through the BIF directory **/tmp** and asks if each BIF file should be removed (forced, with no file mode checks):

bifrm –irf /dev/rdsk/1s0:/tmp

The second example removes the BIF directory **/users/doug**:

bifrmdir /dev/rdsk/1s0:/users/doug

AUTHOR

Bifrm was developed by HP.

SEE ALSO

bif(4), rm(1), rmdir(1).

NAME

bs - a compiler/interpreter for modest-sized programs

SYNOPSIS

bs [file [args]]

DESCRIPTION

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

If the command line *file* argument is provided, the file is used for input before the console is read. By default, statements read from *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 innermost *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 edit. (Otherwise no file name 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 bypassing 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 the constants **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* ...] sequence.

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 (uppercase or lowercase) optionally followed by letters and digits. Only the first six characters of a name are significant. Except for names declared in *fun* statements, all names are global to the program. Names can take on numeric (double float) values, string values, or can be associated with input/output (see the built-in function *open*( ) 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 that function is printed, and a request 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 file.

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 array element. The result is the right operand. Assignment binds right to left, all other operators bind left to right.

_ _ (under-score) 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. Since it is not always possible to know whether **a** is a number or a string when the format call is coded, coercing **a** to the type required by **f** by either adding zero (for **e** or **f** format) or concatenating (_) the null string (for **s** format) should be considered.
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, sta, width)
: returns the sub-string of s defined by the *starting* 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 no 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*(1). The *requests* are as follows:

Call	*Function*
plot(0, term)	causes further *plot* output to be piped into *tplot*(1) with an argument of **−T***term*. *Term* may be up to 40 characters in length.
plot(1)	"erases" the plotter.
plot(2, string)	labels the current point with *string*.
plot(3, x1, y1, x2, y2)	draws the line between (*x1*,*y1*) and (*x2*,*y2*).
plot(4, x, y, r)	draws a circle with center (*x*,*y*) and radius *r*.
plot(5, x1, y1, x2, y2, x3, y3)	draws an arc (counterclockwise) with center (*x1*,*y1*) and endpoints (*x2*,*y2*) and (*x3*,*y3*).
plot(6)	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*(1). 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.

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    1600
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 "infile" to "newfile":
open("read", "infile", "r")
open("write", "newfile", "w")
...
while "(write = read)
...
# close "read" and "write":
close("read")
close("write")

#    Pipe between commands:
open("ls", "ls * ", "p")
open("pr", "lpr -2 -l 43list", "w")
while "(pr = ls) ...
...
# be sure to close (wait for) these:
close("ls")
close("pr")
```

SEE ALSO

sh(1), sh(1), tplot(1), access(2), printf(3S), stdio(3S), plot(4).
See section 3M for a further description of the mathematical functions (*pow* on *exp*(3M) is used for exponentiation). *bs* uses the Standard Input/Output package.

VARIABLES

EDITOR the editor to use for the edit command

WARNINGS

The graphics mode is nearly useless without *tplot*(1).

BUGS

bs is not extremely tolerant of some errors. Mistyping a *fun* declaration is painful, as a new declaration cannot be be made without doing a *clear*. Starting using the *edit* command is the best solution in this case.

NAME

cal – print calendar

SYNOPSIS

cal [[month] year]

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 [-]

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 "Aug. 24," "august 24," "8/24," etc., are recognized, but not "24 August" or "24/8". On weekends "tomorrow" extends through Monday.

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

FILES

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

SEE ALSO

cron(1M), mail(1).

BUGS

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

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

cat - concatenate, copy, and print files

SYNOPSIS

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

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, enabling you to combine standard input with other files.

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. (The 4.2BSD **cat -s** feature is provided by *ssp*(1).)
- **−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, **−t** 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), pg(1), pr(1), rmnl(1), ssp(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.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

cb - C program beautifier, formatter

SYNOPSIS

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

DESCRIPTION

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

SEE ALSO

cc(1).

The C Programming Language by B. W. Kernighan and D. M. Ritchie.

BUGS

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

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

cc – C compiler

SYNOPSIS

cc [options] files

DESCRIPTION

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

(1) 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**. However, if a single C program is compiled and linked all in one step, the **.o** file is deleted.

(2) Similarly, arguments whose names end with **.s** are taken to be assembly source programs and are assembled, producing a **.o** file.

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

Arguments can be passed to the compiler through the **CCOPTS** environment variable as well as on the command line. The compiler picks up the value of **CCOPTS** and places its contents before any arguments on the command line. For example (in *sh*(1) notation),

```
CCOPTS=-v
export CCOPTS
cc -g prog.c
```

is equivalent to

```
cc -v -g prog.c
```

Options

The following options are recognized by *cc*.

–c Suppress the link edit phase of the compilation, and force an object (**.o**) file to be produced for each **.c** file even if only one program is compiled. Object files produced from C programs must be linked before being executed.

–C Prevent the preprocessor from stripping C-style comments. See *cpp*(1) for details.

–D*name=def*
–D*name* Define *name* to the preprocessor, as if by '#define'. See *cpp*(1) for details.

–E Run only *cpp*(1) on the named C or assembly programs, and send the result to the standard output.

–g Cause the compiler to generate additional information needed by the symbolic debugger.

–I*dir* Change the algorithm used by the preprocessor for finding include files to also search in directory *dir*. See *cpp*(1) for details.

–l*x* Cause the linker to search the library **lib***x***.a**. See *ld*(1) for details.

–n Cause the output file from the linker to be marked as *shareable*. For details and system defaults, see *ld*(1).

–N Cause the output file from the linker to be marked as *unshareable*. For details and system defaults, see *ld*(1).

–o *outfile* Name the output file from the linker *outfile*. The default name is **a.out**.

–O Invoke the optimizer.

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

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

–q Cause the output file from the linker to be marked as *demand loadable*. For details and system defaults, see *ld*(1).

–Q Cause the output file from the linker to be marked as *not demand loadable*. For details and system defaults, see *ld*(1).

–s Cause the output of the linker to be stripped of symbol table information. The use of this option will prevent the use of a symbolic debugger on the resulting program. See *ld*(1) for more details.

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

–t *c,name* Substitute or insert subprocess *c* with *name* where *c* is one or more of a set of identifiers indicating the subprocess(es). This option works in two modes: 1) if *c* is a single identifier, *name* represents the full path name of the new subprocess; 2) if *c* is a set of identifiers, *name* represents a prefix to which the standard suffixes are concatenated to construct the full path names of the new subprocesses.

c can take one or more of the values:

p preprocessor (standard suffix is *cpp*)
c compiler body (standard suffix is *ccom*)
0 same as *c*
a assembler (standard suffix is *as*)
2 optimizer (standard suffix is *c2*)
l linker (standard suffix is *ld*)

–U*name* Remove any initial definition of "name" in the preprocessor. See *cpp*(1) for details.

–v Enable verbose mode, which produces a step-by-step description of the compilation process on *stderr*. Also echoes **CCOPTS** if it is set.

–w Suppress warning messages.

–W *c,arg1*[,*arg2*...]
Hand off the argument[s] *argi* to pass *c* where *c* can assume one of the values listed under the **-t** option as well as *d* (driver program). The **–W** option specification allows additional, implementation-specific options to be recognized by the compiler driver. For example, on the Series 300,

```
–W d,–x
```

causes the driver to call various subprocesses needed to generate MC68020 code. Furthermore, a shorthand notation for this mechanism can be used by placing "+" in front of the option name as in

```
+x
```

which is equivalent to the previous option example. Some commonly used subprocess options can also be abbreviated in a similar fashion. Note that for simplicity, this shorthand must be applied to each option individually. Options that can be abbreviated using "+" are implementation-dependent, and are listed under HARDWARE DEPENDENCIES.

-z Do not bind anything to address zero. This option will allow runtime detection of null pointers. See the note on *pointers* below.

-Z Allow dereferencing of null pointers. See the note on *pointers* below.

Any other options encountered will generate a warning to *stderr*.

Other arguments are taken to be 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 those below.

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

pointers
Accessing the object of a NULL (zero) pointer is technically illegal, (see Kernighan and Ritchie) but many systems have permitted it in the past. The following is provided to maximize portability of code. If the hardware is able to return zero for reads of location zero (when accessing at least 8- and 16-bit quantities), it must do so unless the **-z** flag is present. The **-z** flag requests that SIGSEGV be generated if an access to location zero is attempted. Writes of location zero may be detected as errors even if reads are not. If the hardware cannot assure that location zero acts as if it was initialized to zero or is locked at zero, the hardware should act as if the **-z** flag is always set.

identifiers
Identifiers are significant up to 255 characters.

types
Certain programs require that a type be a specific number of bits wide. It can be assumed that an *int* can hold at least as much information as a *short*, and that a *long* can hold at least as much information as an *int*. Additionally, either an *int* or a *long* can hold a pointer.

HARDWARE DEPENDENCIES

Series 200, 300:

The following options are not supported: **-w**, **-z**.

The default is to allow null pointer dereferencing, hence using **-Z** has no effect.

The default is to generate code for the processor on the machine where the compilation is taking place. For example, on a Series 300 with a MC98020 processor, the compiler will generate MC98020 code.

The compiler driver supports the following cross-compilation options, which may also be passed to it from *cc* using the **-W d** option:

+x *or* **-W d,-x**
causes the compiler to generate inline code for the MC68020 and MC68881.

+X *or* **-W d,-X**
causes the compiler to generate "generic" MC68010 code. The code will also run on MC68020 processors, but it will not take advantage of new architectural

capabilities.

The compiler subprocess *ccom* supports the following options, which may be passed to it from *cc* using the **-W c** option. Some of these can be passed directly to the driver using the "+" notation.

+b *or* **-W c,-b**
causes the MC68010 compiler to generate code for floating point operations that will use the 98635 floating point card if it is installed in the computer at run-time (if not installed, operations will be done in software). This option cannot be used when code is being generated explicitly for the MC68020, either by default on a MC68020 based system or via the **+x** option.

+f *or* **-W c,-f**
causes the MC68010 compiler to generate code for floating point operations that must use the 98635 floating point card. This code does not run unless the floating point card is installed. This option cannot be used when code is being generated explicitly for the MC68020, either by default on a MC68020 based system or via the **+x** option.

+M *or* **-W c,-M**
causes the MC68020 compiler NOT to generate inline code for the MC68881 math coprocessor. Library routines will be referenced for *matherr* capability. This option is meaningless on MC68010 based systems or in conjunction with **+X**.

+N<*secondary*><*n*> *or* **-W c,-N**<*secondary*><*n*>
This option adjusts the size of internal compiler tables. The *ccom* compiler uses fixed size arrays for certain internal tables. *Secondary* is one of the letters from the set **{abdepstw}**, 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:

- **a** maximum size of the asciz table. Default is 10000 table entries.
- **b** maximum size of the bc table. This table saves break and continue labels within loops and switch statements. Default is 100 table entries.
- **d** maximum size of the dimtab table. This table maintains information about the definitions of all structures, unions, and arrays. Default is 1000 table entries.
- **e** maximum number of nodes per statement. Default is 350 table entries.
- **p** maximum size of the parameter stack. Default is 150 table entries.
- **s** maximum size of the symbol table. Default is 1000 table entries.
- **t** maximum size of the tasciz table. Default is 20000 table entries.
- **w** maximum size of the switch table stack. Default is 250 table entries.

-W c,-Y S
This option causes source code lines to be printed on the assembly (.s) file as assembly comments, thus showing the correspondence between C source and the resulting assembly code.

The following option is supported:

−Y Enable support of 16-bit characters inside string literals and comments. Note that 8-bit parsing is always supported. See *hpnls*(5) for more details on International Support.

Series 500:

The following options are not supported: **−p**, **−w**.

The default is not to allow null pointer dereferencing, hence using **−z** has no effect.

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

The compiler subprocess *ccom* supports the following options, which may be passed to it from *cc* using the **-W c** option. Some of these can be passed directly to the driver using the "+" notation.

+N<*secondary*><*n*> *or* **−W c,−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 {**bpwgi**}, 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:

b maximum size of the bc table. This table saves break and continue labels within loops and switch statements. Default is 100 table entries.

p maximum size of the parameter stack. Default is 150 table entries.

g maximum size of the argument stack. Default is 100 table entries.

w maximum size of the switch table. Default is 250 table entries.

i maximum size of the instruction table for generated code. Default is 300 table entries.

The following option is supported:

−Y Enable support of 16-bit characters inside string literals and comments. Note that 8-bit parsing is always supported. See *hpnls*(5) for more details on International Support.

Series 800:

The default is to allow null pointer dereferencing, hence using **−Z** has no effect.

The **-g** option is incompatible with optimization.

The compiler subprocess *ccom* supports the following options, which may be passed to it from *cc* using the **-W** option. Note: The "+opt1 +opt2" notation may be used instead of the "-Wc,-opt1,-opt2" notation.

+a *or* **−W c,−a**

When processing files which have been written in assembly language, do not assemble with the prefix file which sets up the space and subspace structure required by the linker. Files assembled with this option may not be linked unless they contain the equivalent information.

+O*opt* *or* **−W c,-O***opt*

Invoke optimizations selected by *opt*. If *opt* is '1', then only level 1 optimizations are handled. If *opt* is '2', then all optimizations are performed. The option **+O2** is the same as **−O**.

EXAMPLE

The following will compile the C program *prog.c*, creating a *prog.o* file, and will then invoke the

link editor *ld*(1) to link *prog.o* and *procedure.o* with all the C startup routines in */lib/crt0.o* and library routines from the C library *libc.a*; the resulting executable program is output in *prog*:

```
cc prog.c procedure.o -o prog
```

FILES

file.c	input file
file.o	object file
a.out	linked output
/tmp/ctm*	temporary
/usr/tmp/ctm*	temporary
/lib/cpp	preprocessor
/lib/ccom	compiler, *cc*
/lib/c2	optional optimizer (for Series 200, Series 300 and Series 500 only)
/bin/as	assembler, *as*(1)
/bin/ld	link editor, *ld*(1)
/lib/crt0.o	runtime startoff
/lib/mcrt0.o	startoff for profiling
/lib/libc.a	standard C library, see section 3 of this manual
/usr/include	standard directory for **#include** files
Series 200, 300:	
/lib/ccom10	compiler, MC68010 version (linked to **/lib/ccom** on MC68010 systems).
/lib/ccom20	compiler, MC68020 version (linked to **lib/ccom** on MC68020 systems).
/lib/c210	optimizer, MC68010 version (linked to **lib/c2** on MC68010 systems).
/lib/c220	optimizer, MC68020 version (linked to **lib/c2** on MC68020 systems).
/bin/as10	assembler, MC68010 version (linked to **/bin/as** on MC68010 systems).
/bin/as20	assembler, MC68020 version (linked to **/bin/as** on MC68020 systems).
Series 800:	
/lib/libp/libc.a	C library for profiled programs
/usr/lib/cc_msgs.cat	compiler messages catalog

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.

WARNINGS

Options not recognized by *cc* are not passed on to the link editor. The option **-W l,***arg* may be used to pass any such option to the link editor.

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

SEE ALSO

adb(1), cdb(1), cpp(1), as(1), ld(1), prof(1), exit(2), monitor(3C), matherr(3M).

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

INTERNATIONAL SUPPORT

8- and 16-bit data only in strings and comments, 8-bit filenames.
Series 800 does not support 16-bit data.

NAME

cd – change working directory

SYNOPSIS

cd [directory]

DESCRIPTION

If *directory* is not specified, the value of shell parameter **$HOME** is used as the new working directory. If *directory* specifies a complete path starting with /, .. ., *directory* becomes the new working directory. If neither case applies, *cd* tries to find the designated directory relative to one of the paths specified by the **$CDPATH** shell variable. **$CDPATH** has the same syntax as, and similar semantics to, the **$PATH** shell variable. *Cd* 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 is internal to the shell.

VARIABLES

HOME	default working directory
CDPATH	directories to search for directory.

SEE ALSO

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

INTERNATIONAL SUPPORT

8-bit filenames, messages

NAME

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

SYNOPSIS

cdb [**-d** *dir*] [**-r** *file*] [**-p** *file*] [**-S** *num*] [*objectfile* [*corefile*]]

fdb [*cdb* options]

pdb [*cdb* options]

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 having one or more of its component modules compiled with the **-g** option. The main procedure (main program) must have been compiled with **-g**. 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). (done automatically with **-g** option to *cc*(1), *fc*(1), and *pc*(1)). The default for *objectfile* is **a.out**.

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

The options are:

-d *dir* names an alternate directory where source files are located. (The current directory is searched last.)

-r *file* names a *record* file which is invoked immediately (for overwrite, not for append). Used with **Record and Playback Commands**.

-p *file* names a *playback* file which is invoked immediately. Used with **Record and Playback Commands**.

–S *num* sets the size of the string cache to *num* bytes (default is 1024). The string cache holds data read from *objectfile*.

Only one *objectfile* and one *corefile* are allowed per debugging session. The program (*objectfile*) is not invoked as a child process until an appropriate *Job Control Command* command is given. The same program can be restarted many times (as different child processes) during a single debugging session.

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. Used with *Data Viewing Commands.*

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. 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:

$*var* The debugger has room in its own address space for a number of user-created special variables. There are 26 of them by default (this number is adjustable using the **-s** invocation option). They are all of type **long**, and do not take on the type of any expression they are assigned. Names are defined when they are first seen. For example, saying "$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**.

$result
This is used to reference the return value from the last command-line procedure call. Where possible, it takes on the type of the procedure. **$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.

$pagelines
This lets you set the number of lines per "page" of debugger output. The prompt "Hit RETURN for more..." occurs between pages. Values of zero or less turn off paging.

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 are normally silent. To force an expression result to be printed, follow the expression with "**/n**" (print in normal format).

Integer constants may begin with "**0**" for octal or "**0x**" or "**0X**" for hexadecimal (the forms are equivalent). They are **int** (which may be the same as **long**) 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 ***" (i.e. 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 *Data Viewing Commands*).

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 *Data Viewing Commands*). 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.

Array indices are not checked against 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 various commands for viewing and manipulating the program being debugged.

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.

dir *"directory"*
: Add *directory* to the list of alternate source directories. Same as using **-d** invocation option. Main procedure file must reside in the current directory or be specified with the **-d** option.

e
: Show the current file, procedure, line number, and source line.

e (*file* | *proc*)
: Enter (view) *file* or *proc* and print its first executable line. *File* can be any file, but must not be object code.

[*depth*] **E**
: Like "**e**", but sets viewing location to the current location in *proc* on the stack at depth *depth* (not necessarily the first executable line in the procedure). Default *Depth* is zero (where 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 current line (or line number *line*). If multiple lines are printed, current line is marked with "=" in leftmost column.

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

−[*lines*]
: Move to *lines* (default one) lines before current line.

[*line*] **w** [*size*]
: Print window of text containing *size* (default 11) lines centered around current line (or *line*). Target line is marked with "=" in leftmost column if multiple lines printed.

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

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

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

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

?[*string*]
: Search backward for *string*, starting with 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 previous "**/**" or "**?**" command using same *string* as before.

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

Display Formats

A *format* is of the form "[*****][*count*]*formchar*[*size*]". Display formats apply only to Data Viewing Commands, described in the next sub–section.

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

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

Size is the number of bytes to be formatted for each *count* (overrides the default *size* for the format style); must be positive decimal *number* (except short hand notations). *Size* is disallowed with *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*(3S)). 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 (same as "*****_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(arg)/t".

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

S Do a formatted dump of a structure. *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 resemble anything else (such as a command), it is handled as "*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*.

expr?*format*
Print the address of *expr* using *format*.

^ [[/]*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. No "**/**" is needed after the "**^**". To reverse direction again (e.g. start going forward), enter "**.**" (*dot*) (always an alias for the current location) followed by carriage return.

l [*proc*[.*depth*]]
List all parameters and local variables for current procedure (or *proc*, if given, at the specified *depth*, if any). Data display uses "**/n**" format, except arrays and pointers are shown as addresses; only the first word of a 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 arrays and pointers are shown as addresses; structures show first word 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 (includes hitting a breakpoint) or terminated for whatever reason.

r [*arguments*]
Run a new child process with given argument list, if any (an existing child process is terminated first). If no *arguments* are given, those 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; ">>" is not supported.) *Arguments* may contain shell variables, quote marks, or other special syntax (expanded by Bourne shell). "{}" are shell metacharacters, so "r" cannot be safely saved in a breakpoint or assertion command list.

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 **Breakpoint Commands**).

[*count*] **C** [*line*]
Continue like "c", but allow the signal (if any) to be received.

[*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. The child process continues with the current signal, if any (set "$signal = 0" to prevent).

[*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 unless 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. Programs being debugged should not execute a different program via *exec*(2).

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

Breakpoint Commands

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 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 "c" or "C").

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.

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.

Results of expressions followed by ";" or "}" are not printed unless you specify a print format.

Saved commands are not parsed until the breakpoint is recognized. If there are no *commands*, the debugger will wait for command input when the breakpoint is recognized. For immediate continuation, finish the command list with "**c**".

Breakpoint commands:

l b
B Both forms list all breakpoints in the format:

num: count: *nnn proc*: *ln*: *contents*
{*commands*}

The leftmost number *num* 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).

[*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.

bp [*commands*]
Set permanent breakpoints at the beginning (first executable line) of every debuggable procedure. When any procedure breakpoint is hit, *commands* are executed.

D [b] Delete all breakpoints (including "procedure" breakpoints). The "**b**" is optional.

D p Delete all "procedure" breakpoints. All breakpoints set by commands other than "**bp**" remain set.

For the following commands, if the second character is upper case, for example, "**bU**" instead of "**bu**", 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 specified stack *depth*. If *depth* is not specified, use the current procedure (may 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, use the current procedure (may not be the same as the one at *depth* zero). The breakpoint is set 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). Zero *depth* means "current location".

[*depth*] **bt** [*proc*] [*commands*]
[*depth*] **bT** [*proc*] [*commands*]
Trace current procedure (or procedure at *depth*, or *proc*). Set breakpoints at entrance and exit of a procedure. Default entry breakpoint *commands* are "**Q;2t;c**" (show top two procedures on stack and continue). The exit breakpoint always executes "**Q;L;c**" (print current location and continue).

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. *address* can be the name of a procedure or an expression containing such a name. If the child process is stopped in a non-debuggable procedure, or in prologue code (before the first executable line of a procedure), results may seem a little strange.

The next few commands are not strictly part of the breakpoint group, but 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, otherwise it (and the following "**{**", if any) is skipped. 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 can 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. String may contain standard backslashed character escapes, including "**\n**" for newline. Useful for labelling output from breakpoint commands.

Assertion Control Commands

Assertions are command lists that are executed before every statement. Thus, if there is even one active assertion, the program is single stepped at the machine instruction level (runs very slowly). They are primarily used for tracking down nasty bugs (such as the corruption of a global variable).

Assertions can be activated or suspended individually, plus there is an overall mode.

a *commands*
Create new assertion with given command list. List is not parsed until execution time. Command list can be enclosed in "**{}**" to delimit it from other commands on the same line. The "**l a**" command lists 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 do nothing until reactivated.

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

D a Delete all assertions.

[*flag*] **x** Force exit from assertions mode. If *flag* is absent or evaluates to zero, exit immediately. Otherwise, finish executing the current assertion first. If an 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. The current command line is discarded when assertion execution begins.

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.

Signal Control Commands

The debugger catches all signals bound for a child process before the child process sees them (a function of the *ptrace*(2) mechanism).

[*signal*] **z** [**i**][**r**][**s**][**Q**]
: Maintains the "zignal" (signal) handling table. *Signal* is a valid signal number (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 signal state is not printed.

"**l z**" lists the current handling of all signals. "**z**" (with no options) shows 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 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 does not know about it. A signal is never ignored when the child process terminates, only when it stops.

Record and Playback Commands

The debugger supports a record/playback feature to help recreate program states and to record all debugger output.

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* Same as >>*file* 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 can be opened or closed independent of the recordfile. All debugger standard output is copied to the record-all file, including prompts, commands entered, and command output (does not capture child process output).

>(**t** | **f** | **c**)
: Turn recording on ("**t**") or off ("**f**"), or close the recording file ("**c**"). When recording is resumed, new commands are appended to previous file contents. In this context, ">>" is equivalent to ">".

>@(**t** | **f** | **c**) Turn record-all on, off, or close the record-all file. In this context, ">>@" is equivalent to ">@".

> Tell the current recording status (same as ">>").

>@ Tell the current record-all status (same as ">>@").

<*file* Start playback from *file*.

<<*file* Start playback from *file*, using the single-step feature of playback.

Only command lines read from the keyboard or a playback file are recorded in the recordfile.

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

NOTE: The debugger can 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. <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. 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*(3S). Otherwise, the environment variable SHELL gives the name of the shell program to invoke with a –i option, also using *system*(3S). 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*(3S) 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 current procedure). 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 *exprs* given.

q Quit the debugger. 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 300, 500

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.

$short and **$long** are available in addition to **$result**. If command-line procedure call returns a **double**, **$result** is set to the value cast to **long**.

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. To force linking of library routines not otherwise referenced, use the **-u** option to *ld*(1).

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.

FORTRAN ENTRY points are flagged "ENTRY" by the "**l p**" command.

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.

There is no support for: C structure, union, and enumeration tags, C typedefs, and Pascal types.

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.

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.

Series 300

Series 300 supports two types of string formats in addition to null-terminated C strings. FORTRAN *character* variables consist of a string of bytes (no null terminator). Pascal *string* variables consist of a length byte, followed by the string characters. The "\s" and "\a" formats will display these types correctly, only if the current language is FORTRAN or Pascal.

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.

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.

There are no address maps in the usual sense, so the "**M**" command is not supported.

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).

Code and data pointers in *objectfile* both 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.

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.

AUTHOR

Cdb was developed by Third Eye Software.

SEE ALSO

cc(1), echo(1), fc(1), pc(1), ld(1), more(1), creat(2), exec(2), fork(2), open(2), setjmp(3C), printf(3S), system(3S), a.out(4), and the *cdb Debugger* tutorial in *HP-UX Concepts and Tutorials.*

On some systems any of the following may exist: adb(1), ptrace(2), core(4).

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.

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*(3S) 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*(3C), 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".

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
```

yacc(1) will leave out line directives if invoked with the –l option. 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**", "**mod**", "**addr**", and "**sizeof**" are added.

There is no support for FORTRAN **complex** variables, except as a series of two separate **floats** or **doubles**.

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.

CDB does not support identically-named procedures (legal in Pascal if the procedures are in different scopes). CDB will always use the first procedure with the given name.

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.

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. Series 500 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.

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.

INTERNATIONAL SUPPORT

cdb: 8-bit filenames, messages.

NAME

cdc – change the delta commentary of an SCCS delta

SYNOPSIS

cdc –rSID [–**m**[mrlist]] [–**y**[comment]] files

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

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

MRs in a list are separated by blanks and/or tab characters. An unescaped 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 *Source Code Control System User Guide*. 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(4).

Source Code Control System User Guide in the *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.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

cflow – generate C flow graph

SYNOPSIS

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

DESCRIPTION

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

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

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

As an example, given the following in *file.c*:

```
int     i;

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

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

the command

```
cflow –ix file.c
```

produces the output

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

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

The following options are interpreted by *cflow*:

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

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

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

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

DIAGNOSTICS

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

HARDWARE DEPENDENCIES

Series 200, 300, 500

The following option is supported:

–Y Enable support of 16-bit characters inside string literals and comments. Note that 8-bit parsing is always supported. See ***hpnls***(5) for more details on International Support.

SEE ALSO

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

BUGS

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

NAME

chatr – change program's internal attributes

SYNOPSIS

chatr [–**n**] [–**q**] [–**s**] files

Remarks:

This manual page describes *chatr* as implemented on Series 200 computers. Refer to other *chatr*(1) manual pages for information valid for other implementations.

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:

-n change the file from demand loaded to shared.

-q change the file from shared to demand loaded.

–s perform action silently.

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

The error messages produced by *chatr* should be self-explanatory.

NAME

chatr – change program's internal attributes

SYNOPSIS

/lbin/chatr [+c|-c] [+g|-g] [+h|-h] [-mn] [+n|-n] [+p|-p] [-q|-Q] [-s] [+z|-z] file ...

Remarks:

This manual page describes *chatr* as implemented on Series 500 computers. Refer to other *chatr*(1) manual pages for information valid for other implementations.

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.

-q set the demand load bit for each segment.

-Q clear the demand load bit for each segment.

-s perform action silently.

-w*n* change the maximum working set size to *n* bytes.

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;

working set size is larger than heap size.

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** or **-w** 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 ...

DESCRIPTION

The permissions of the named *file*s 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 or sticky); **u**, **g** or **o** indicate that *permission* is to be taken from the current mode. Omitting *permission* is only useful with = to take away all permissions.

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

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

EXAMPLES

The first example denies write permission to others, and the second makes a file executable:

```
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:

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

SEE ALSO

ls(1), chmod(2).

INTERNATIONAL SUPPORT

8-bit filenames.

NAME

chown, chgrp – change file owner or group

SYNOPSIS

chown owner file ...

chgrp group file ...

DESCRIPTION

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

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

In order to change the owner or group, you must own the file or be the super-user. If either command is invoked by other than the super-user, the set-user-ID and set-group-ID bits of the file mode, 04000 and 02000 respectively, will be cleared.

FILES

/etc/group
/etc/passwd

SEE ALSO

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

INTERNATIONAL SUPPORT

8-bit filenames.

NAME

chsh – change default login shell

SYNOPSIS

chsh name [shell]

DESCRIPTION

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

An example use of this command is:

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

AUTHOR

Chsh was developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

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

NAME

clear – clear terminal screen

SYNOPSIS

clear

DESCRIPTION

Clear clears your screen if this is possible. It reads the **TERM** environment variable for the terminal type and then reads the appropriate terminfo data base to figure out how to clear the screen.

FILES

/usr/lib/terminfo/?/* terminal database files

AUTHOR

Clear was developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

terminfo(4).

NAME

cmp – compare two files

SYNOPSIS

cmp [**–l**] [**–s**] file1 file2

DESCRIPTION

The two files are compared. (If *file1* is –, the standard input is used.) Under default 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.

Options:

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

–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.

INTERNATIONAL SUPPORT

8-bit and 16-bit data, 8-bit filenames, messages.

NAME

col – filter reverse line-feeds and backspaces

SYNOPSIS

col [**–blfxp**]

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 **ESCs+1-8**). *Col* is particularly useful for filtering multicolumn 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** (\016) and **SIs+1** (\017) are assumed by *col* to start and end text in an alternate character set. The character set to which each input character belongs is remembered, and on output **SI** and **SO** characters are generated as
appropriate to ensure that each character is printed in the correct character set.

On input, the only control characters accepted are space, backspace, tab, return, 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.

SEE ALSO

nroff(1), tbl(1).

NOTES

The input format accepted by *col* matches the output produced by *nroff* 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.

BUGS

Cannot back up more than 128 lines.
There is a maximum limit for the number of characters, including backspaces and overstrikes, on a line. The maximum limit is at least 800 characters.
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.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

comb – combine SCCS deltas

SYNOPSIS

comb [–psid] [–clist] [–o] [–s] files

DESCRIPTION

Comb generates a shell procedure (see *sh*(1)) which, when run, will reconstruct the given SCCS files. The reconstructed files will, hopefully, be smaller than the original files. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, *comb* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the 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 generated shell procedure is written on the standard output.

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

–p*sid* The *S*CCS *ID*entification string (SID) of the oldest delta to be preserved. All older deltas are discarded in the reconstructed file.

–c*list* A *list* (see *get*(1) for the syntax of a *list*) of deltas to be preserved. All other deltas are discarded.

–o For each **get –e** generated, this argument causes the reconstructed file to be accessed at the release of the delta to be created, otherwise the reconstructed file would be accessed at the most recent ancestor. Use of the **–o** keyletter may decrease the size of the reconstructed SCCS file. It may also alter the shape of the delta tree of the original file.

–s This argument causes *comb* to generate a shell procedure which, when run, will produce a report giving, for each file: the file name, size (in blocks) after combining, original size (also in blocks), and percentage change computed by:

100 * (original – combined) / original

It is recommended that before any SCCS files are actually combined, one should use this option to determine exactly how much space is saved by the combining process.

If no keyletter arguments are specified, *comb* will preserve only leaf deltas and the minimal number of ancestors needed to preserve the tree.

FILES

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

SEE ALSO

admin(1), delta(1), get(1), help(1), prs(1), sh(1), sccsfile(4).

Source Code Control System User Guide in the *HP-UX User's Guide*.

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

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

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

comm – select or reject lines common to two sorted files

SYNOPSIS

comm [– [**123**]] file1 file2

DESCRIPTION

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

Flags 1, 2, or 3 suppress printing of the corresponding column. Thus **comm –12** prints only the lines common to the two files; **comm –23** prints only lines in the first file but not in the second; **comm –123** is a no-op.

SEE ALSO

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

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

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

SYNOPSIS

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

DESCRIPTION

Compact compresses the named files using an adaptive Huffman code. If no file names are given, the standard input is compacted to the standard output. *Compact* operates as an on-line algorithm. Each time a byte is read, it is encoded immediately according to the current prefix code. This code is an optimal Huffman code for the set of frequencies seen so far. It is unnecessary to attach a decoding tree in front of the compressed file since the encoder and the decoder start in the same state and stay synchronized. Furthermore, *compact* and *uncompact* can operate as filters. In particular,

... | compact | uncompact | ...

operates as a (very slow) no-op.

When an argument *file* is given, it is compacted and the resulting file is placed in *file.C; file* is unlinked. The first two bytes of the compacted file code the fact that the file is compacted. This code is used to prohibit recompaction.

The amount of compression to be expected depends on the type of file being compressed. Typical values of compression are: Text (38%), Pascal Source (43%), C Source (36%) and Binary (19%). These values are the percentages of file bytes reduced.

Uncompact restores the original file from a file compressed by *compact*. If no file names are given, the standard input is uncompacted to the standard output.

Ccat cats the original file from a file compressed by *compact*, without uncompressing the file.

RESTRICTION

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

FILES

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

SEE ALSO

Gallager, Robert G., 'Variations on a Theme of Huffman', *I.E.E.E. Transactions on Information Theory*, vol. IT-24, no. 6, November 1978, pp. 668 - 674.

Series 300 and 500 Only

NAME

compress, uncompress, zcat – compress and expand data

SYNOPSIS

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

DESCRIPTION

Compress reduces the size of the named files using adaptive Lempel-Ziv coding. Whenever possible, each file is replaced by one with the extension **.Z,** while keeping the same ownership modes, access and modification times. If no files are specified, the standard input is compressed to the standard output. Compressed files can be restored to their original form using *uncompress* or *zcat.*

The **–f** option will force compression of *name.* This is useful for compressing an entire directory, even if some of the files do not actually shrink. If **–f** is not given and *compress* is run in the foreground, the user is prompted as to whether an existing file should be overwritten.

The **–c** option makes *compress/uncompress* write to the standard output; no files are changed. The nondestructive behavior of *zcat* is identical to that of *uncompress* **–c.**

Compress uses the modified Lempel-Ziv algorithm popularized in "A Technique for High Performance Data Compression", Terry A. Welch, *IEEE Computer,* vol. 17, no. 6 (June 1984), pp. 8-19. Common substrings in the file are first replaced by 9-bit codes 257 and up. When code 512 is reached, the algorithm switches to 10-bit codes and continues to use more bits until the limit specified by the **–b** flag is reached (default 16). *Bits* must be between 9 and 16. The default can be changed in the source to allow *compress* to be run on a smaller machine.

After the *bits* limit is attained, *compress* periodically checks the compression ratio. If it is increasing, *compress* continues to use the existing code dictionary. However, if the compression ratio decreases, *compress* discards the table of substrings and rebuilds it from scratch. This allows the algorithm to adapt to the next "block" of the file.

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

The amount of compression obtained depends on the size of the input, the number of *bits* per code, and the distribution of common substrings. Typically, text such as source code or English is reduced by 50–60%. Compression is generally much better than that achieved by Huffman coding (as used in *pack*), or adaptive Huffman coding (*compact*), and takes less time to compute.

Under the **–v** option, a message is printed yielding the percentage of reduction for each file compressed.

If the **–V** option is specified, the current version and compile options are printed on stderr.

Exit status is normally 0; if the last file is larger after (attempted) compression, the status is 2; if an error occurs, exit status is 1.

SEE ALSO

pack(1), compact(1)

DIAGNOSTICS

Usage: compress [–dfvcV] [–b maxbits] [file ...]
: Invalid options were specified on the command line.

Missing maxbits
: Maxbits must follow **–b.**

file: not in compressed format
: The file specified to *uncompress* has not been compressed.

file: compressed with *xx* bits, can only handle *yy* bits

File was compressed by a program that could deal with more *bits* than the compress code on this machine. Recompress the file with smaller *bits*.

file: already has .Z suffix -- no change

The file is assumed to be already compressed. Rename the file and try again.

file: filename too long to tack on .Z

The file cannot be compressed because its name is longer than 12 characters. Rename and try again. This message does not occur on BSD systems.

file already exists; do you wish to overwrite (y or n)?

Respond "y" if you want the output file to be replaced; "n" if not.

uncompress: corrupt input

A SIGSEGV violation was detected which usually means that the input file has been corrupted.

Compression: *xx.xx%*

Percentage of the input saved by compression. (Relevant only for **-v**.)

-- not a regular file: unchanged

When the input file is not a regular file, (e.g. a directory), it is left unaltered.

-- has *xx* other links: unchanged

The input file has links; it is left unchanged. See *ln*(1) for more information.

-- file unchanged

No savings is achieved by compression. The input remains virgin.

BUGS

Although compressed files are compatible between machines with large memory, **-b12** should be used for file transfer to architectures with a small process data space (64K bytes or less, as exhibited by the Digital Equipment Corporation PDP series, the Intel 80286, etc.)

NAME

cp, ln, mv – copy, link or move files

SYNOPSIS

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

DESCRIPTION

File1 is copied (linked, moved) to *target*. Under no circumstance can *file1* and *target* be the same (take care when using *sh*(1) metacharacters). 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. If *target* is a file, its contents are destroyed.

If the **–r** option is specified, then for each source directory *cp* copies the subtree rooted at that directory to *target*. If *target* exists it must be a directory, in which case *cp* creates a directory within *target* with the same name as *source*, and then copies the subtree rooted at *source* to *target/source*. It is an error if *target/source* already exists. If *target* does not exist, *cp* creates it and then copies the subtree rooted at *source* to *target*. Note that *cp –r* will not merge subtrees. *Target* should not reside beneath *source*, and *source* should not have a cyclic directory structure, since in these cases *cp* will attempt to copy an infinite amount of data.

If *mv* or *ln* determines that the mode of *target* forbids writing, it will ask permission to overwrite the file. This is done by printing the mode (see *chmod*(2)) followed by the first letters for the words *yes* and *no* in the current native language, asking for a response, and reading the standard input for one line. If the response begins with the first of the choices displayed and if permissible, the operation occurs; if not, the command exits. No questions are asked and the *mv* or *ln* is done when the **–f** option is used or if the standard input is not a terminal.

If *file1* is a directory, *mv* renames *file1* to *target* only if the two directories have the same parent. *Ln* does not permit *file1* to be a directory. *Cp* permits *file1* to be a directory only if the **–r** option is specified. If *file1* is a file and *target* is a link to another file with links, the other links remain and *target* becomes a new file. When using *cp*, if *target* is not a file, a new file is created which has the same mode as *file1* except that the sticky bit is not set unless you are super-user; the owner and group of *target* are those of the user. If *target* is a file, copying a file into *target* does not change its mode, owner, nor group. The last modification time of *target* (and last access time, if *target* did not exist) and the last access time of *file1* are set to the time the copy was made. If *target* is a link to a file, all links remain and the file is changed.

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(1M), 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 becomes that of the copying process and any linking relationship with other files is lost. *Ln* cannot not create hard links across file systems.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

cpio – copy file archives in and out

SYNOPSIS

cpio **–o** [**aBcxv**]

cpio **–i** [**BdcrtuxvmfPsSb6R**] [patterns]

cpio **–p** [**aduxvlm**] directory

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. Output is padded to a 512-byte boundary.

Cpio **–i** (copy in) extracts files from the standard input, which is assumed to be the product of a previous **cpio** **–o**. Only files with names that match *patterns* are selected. *Patterns* are given in the name-generating notation of *sh*(1). In *patterns*, meta-characters **?**, *****, and [...] match the slash **/** character. Multiple *patterns* may be specified and if no *patterns* are specified, the default for *patterns* 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. The permissions of the files will be those of the previous **cpio** **–o**. The owner and group of the files will be that of the current user unless the user is super-user, which causes *cpio* to retain the owner and group of the files of the previous **cpio** **–o**.

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. Destination path names are interpreted relative to the named *directory*.

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 5,120 bytes to the record (does not apply to the *pass* option); meaningful only with data directed to or from devices which support variable length records such as magnetic tape.
d	Create directories as needed.
c	Write header information in ASCII character form for portability.
r	Interactively rename files. If the user types a null line, the file is skipped.
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 **–lx** 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. This option does not destroy existing files. Usable only with the **–p** option.
m	Retain previous file modification time. This option is ineffective on directories that are being copied.
f	Copy in all files except those in *patterns*.
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). Files copied in this mode are not changed. Non-ASCII files will probably need further processing to be readable. this processing often requires knowledge of the content of the file and thus cannot always be done by this program. (**PDP–11** and **VAX** are registered trademarks of Digital Equipment Corporation). The **–s** , **–S** and **–b**

options below can be used when swapping all the bytes on the tape, rather than just the headers, is appropriate. In general, text is best processed with **-P** and binary data with one of the other options.

s Swap all bytes of the file. Use only with the **–i** option.

S Swap all halfwords of the file. Use only with the **–i** option.

b Swap both bytes and halfwords. Use only with the **–i** option.

6 Process a UNIX Sixth Edition format file. Only useful with **–i** (copy in).

R Resynchronize automatically when *cpio* gets *"Out of phase,"* (see DIAGNOSTICS).

Note that *cpio* archives created using a raw device file must be read using a raw device file.

When the end of the tape is reached, *cpio* will prompt the user for a new special file and continue.

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

General:

The use of *cpio* with cartridge tape units requires additional comments. For an explanation of the constraints on cartridge tapes, see *ct*(7).

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 suitable for cartridge tapes.

The **–B** option must be used when writing directly (i.e., without using *tcio*(1)) to a CS-80 cartridge tape unit.

At and before release 4.0 on the 500 and 2.2 on the 200 and 300, these systems wrote a format which, when crossing media boundaries on some kinds of disks, differs from the format specified by System V.2 (although it matched that written by System III). The program **/etc/ocpio** will read and write this format and has essentially the same features as *cpio* except that options **–S**, **–b** and **-f** are omitted. However, **/etc/ocpio** is considered obsolescent.

Series 500:

All files with inodes 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.

Note that the **–B** option must *not* be used when performing raw I/O to the internal miniature flexible disk 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/rmt/0m

cd olddir
find . –depth –print | cpio –pdl newdir
```

The trivial case "find . –depth –print | cpio –oB >/dev/rmt/0m" can be handled more efficiently by:

find . –cpio /dev/rmt/0m

SEE ALSO

ar(1), find(1), tar(1), tcio(1), cpio(4).

DIAGNOSTICS

The diagnostic message "Out of phase" indicates that *cpio* could not successfully read its particular "magic number" in the header. Without the R option specified, *cpio* will fail and return exit code 2. With the R option, *cpio* will try resyncing automatically. (Resyncing means that *cpio* tries to find the next good header in the archive and continues processing from there.) If *cpio* tries to resynchronize from being "Out of phase", it will return exit code 1. If resynchronization fails, try changing header mode (**c** option) or byte swapping the header (**P** or **s** options).

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.

BUGS

Path names are restricted to 256 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-AT&T) 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.

The **–plu** option will not link files to existing files.

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.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

cpp – the C language preprocessor

SYNOPSIS

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

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. Thus 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. See *m4*(1) for a general macro processor.

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 options to *cpp* are recognized:

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

–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.

–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:	hp9000s800, hp9000s500, hp9000s300, hp9000s200, hp9000ipc, interdata, pdp11, u370, u3b, u3b5, vax
UNIX systems variant:	RES, RT, TS, PWB, hpux
lint(1):	lint

All HP-UX systems will have the symbols PWB, hpux, and unix defined. Each system will define exactly one hardware variant, as appropriate. The lint symbol will be defined when *lint*(1) is running.

–D*name*

–D*name=def*

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

–T On HP-UX, preprocessor symbols are no longer restricted to eight characters. The **–T** option forces *cpp* to use only the first eight characters for distinguishing different preprocessor names. This behavior is the same as preprocessors on some other systems with respect to the length of names and is included for backward compatability.

–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 file containing the **#include** line, then in directories named in **–I** options in left-to-right order, and last in directories on a standard list. For **#include** files whose names are enclosed in <>, the directory of the file containing the **#include** line is not searched. However, the

directory *dir* is still searched.

-H*nnn* Change the internal macro definition table to be *nnn* bytes in size. The macro symbol table will be increased proportionally. The default table size is at least 36000 bytes. This option serves to eliminate the "too many defines" and "too much defining" errors.

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 #. Any number of blanks and tabs are allowed between the # and the directive. 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*
Notice that there can be no space between *name* and the (. Replace subsequent instances of *name* followed by a (, a list of comma-separated set of tokens, and a) by *token-string*, where each occurrence of an *arg* in the *token-string* is replaced by the corresponding set of tokens in the comma-separated list. When a macro with arguments is expanded, the arguments are placed into the expanded *token-string* unchanged. After the entire *token-string* has been expanded, *cpp* re-starts its scan for names to expand at the beginning of newly created *token-string*.

#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*). See the **-I** option above for more detail.

#line *integer-constant "filename"*
Causes *cpp* to generate line control information for the next pass of the C compiler. *Integer-constant* is the line number of the next line and *filename* is the file where it comes from. 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. This makes it easier, when reading the source, to match **#if**, **#ifdef**, and **#ifndef** directives with their associated **#endif** directive.

#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**.

#ifndef *name*
The lines following will not 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**.

#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 names up to 255 characters in length.

HARDWARE DEPENDENCIES

Series 200, 300:

In the hardware *name* definition associated with predefined symbols (see –U option), two hardware variants are defined instead of one. Both hp9000s200 and hp9000s300 are present, and they are treated synonymously because of the similarity between the two series.

FILES

/usr/include standard directory for **#include** files

SEE ALSO

cc(1), m4(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.

NOTES

When new-line characters were found in argument lists for macros to be expanded, previous versions of *cpp* put out the new-lines as they were found and expanded. The current version of *cpp* replaces these new-lines with blanks to alleviate problems that the previous versions had when this occurred.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

crontab – user crontab file

SYNOPSIS

crontab [file]
crontab -r
crontab -l

DESCRIPTION

Crontab copies the specified file, or standard input if no file is specified, into a directory that holds all users' crontabs. The –r option removes a user's crontab from the crontab directory. *Crontab* –l will list the crontab file for the invoking user.

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

A crontab file consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns that specify the following:

minute (0–59),
hour (0–23),
day of the month (1–31),
month of the year (1–12),
day of the week (0–6 with 0=Sunday).

Each of these patterns may be either an asterisk (meaning all legal values), or a list of elements separated by commas. An element is either a number, or two numbers separated by a minus sign (meaning an inclusive range). Note that the specification of days may be made by two fields (day of the month and day of the week). If both are specified as a list of elements, both are adhered to. For example, 0 0 1,15 * 1 would run a command on the first and fifteenth of each month, as well as on every Monday. To specify days by only one field, the other field should be set to * (for example, 0 0 * * 1 would run a command only on Mondays).

The sixth field of a line in a crontab file is a string that is executed by the shell at the specified times. A percent character in this field (unless escaped by \) is translated to a new-line character. Only the first line (up to a % or end of line) of the command field is executed by the shell. The other lines are made available to the command as standard input.

The shell is invoked from your **$HOME** directory with an **arg0** of **sh.** Users who desire to have their *.profile* executed must explicitly do so in the crontab file. *Cron* supplies a default environment for every shell, defining **HOME**, **LOGNAME**, **SHELL(=/bin/sh)**, and **PATH(=:/bin:/usr/bin:/usr/lbin)**.

NOTE

Users should remember to redirect the standard output and standard error of their commands! If this is not done, any generated output or errors will be mailed to the user.

FILES

/usr/lib/cron	main cron directory
/usr/lib/cron/cron.allow	list of allowed users
/usr/lib/cron/cron.deny	list of denied users
/usr/spool/cron/crontabs	spool area
/usr/lib/cron/log	accounting information

SEE ALSO

cron(1M), sh(1).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

crypt – encode/decode files

SYNOPSIS

crypt [password]

REMARKS

The decryption facilities provided by this software are under control of the United States Government and cannot be exported without special licenses. These capabilities are only available by special arrangement through HP.

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. 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(1), stty(1).

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)).

If two or more files encrypted with the same key are concatenated and an attempt is made to decrypt the result, only the contents of the first of the original files will be decrypted correctly.

NAME

csh - a shell (command interpreter) with C-like syntax

SYNOPSIS

csh [**-cefinstvxTVX**] [command file] [argument list ...]

DESCRIPTION

Csh is a command language interpreter incorporating a command history buffer and a C-like syntax. If supported by the system, *csh* also contains job control facilities.

The command options are interpreted as follows:

-c Commands are read from the (single) following argument which must be present. Any remaining arguments are placed in *argv*.

-e The shell exits if any invoked command terminates abnormally or yields a 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* 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 Command input is taken from the standard input.

-t A single line of input is read and executed.

-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.

-T This option disables the tenex features, which use the ESCAPE key for command/filename completion and control-D for listing available files (see the CSH UTILITES section below)

-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.

Jobs

The shell associates a *job* with each pipeline. It keeps a table of current jobs, printed by the *jobs* command, and assigns them small integer numbers. When a job is started asynchronously with '&', the shell prints a line which looks like:

[1] 1234

indicating that the job which was started asynchronously was job number 1 and had one (top-level) process, whose process id was 1234.

For those systems which support job control, if you are running a job and wish to do something else you may type the currently defined *suspend* character (see *termio*(7)) which sends a stop signal to the current job. The shell will then normally indicate that the job has been 'Stopped', and print another prompt. You can then manipulate the state of this job, putting it in the background with the *bg* command, or run some other commands and then eventually bring the job back into the foreground with the foreground command *fg*. A *suspend* takes effect immediately and is like an interrupt in that pending output and unread input are discarded when it is typed. There is a delayed *suspend* character which does not generate a stop signal until a program attempts to *read*(2) it. This can usefully be typed ahead when you have prepared some commands for a job which you wish to stop after it has read them.

A job being run in the background will stop if it tries to read from the terminal. Background jobs are normally allowed to produce output, but this can be disabled by giving the command "stty tostop". If you set this tty option, then background jobs will stop when they try to produce output like they do when they try to read input.

There are several ways to refer to jobs in the shell. The character '%' introduces a job name. If you wish to refer to job number 1, you can name it as '%1'. Just naming a job brings it to the foreground; thus '%1' is a synonym for 'fg %1', bringing job 1 back into the foreground. Similarly saying '%1 &' resumes job 1 in the background. Jobs can also be named by prefixes of the string typed in to start them, if these prefixes are unambiguous, thus '%ex' would normally restart a suspended *ex*(1) job, if there were only one suspended job whose name began with the string 'ex'. It is also possible to say '%?string' which specifies a job whose text contains *string*, if there is only one such job.

The shell maintains a notion of the current and previous jobs. In output pertaining to jobs, the current job is marked with a '+' and the previous job with a '–'. The abbreviation '%+' refers to the current job and '%–' refers to the previous job. For close analogy with the syntax of the *history* mechanism (described below), '%%' is also a synonym for the current job.

Csh learns immediately whenever a process changes state. It normally informs you whenever a job becomes blocked so that no further progress is possible, but only just before printing a prompt. This is done so that it does not otherwise disturb your work. If, however, you set the shell variable *notify*, *csh* will notify you immediately of changes in status of background jobs. There is also a *csh* built-in command called *notify* which marks a single process so that its status changes will be immediately reported. By default, *notify* marks the current process. You can just say 'notify' after starting a background job to mark it.

When you try to leave the shell while jobs are stopped, you will be warned that 'You have stopped jobs.' You may use the *jobs* command to see what they are. If you do this or immediately try to exit again, the shell will not warn you a second time, and the suspended jobs will be terminated (see *exit*(2)).

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 built-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**.

bg [%*job* ...]
Puts the current (no argument) or specified jobs into the background, continuing them if they were stopped. This command is supported only if job control is available.

break Causes execution to resume after the *end* of the nearest enclosing *foreach* or *while*. The remaining commands on the current line are executed. Multi-level breaks are thus possible by writing them all on one line.

breaksw
Causes a break from a *switch*, resuming after the *endsw*.

case *label*:
A label in a *switch* statement as discussed below.

cd
cd *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).

fg [*%job* ...]
Brings the current (no argument) or specified jobs into the foreground, continuing them if they were stopped. This command is supported only if job control is available.

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 job control is supported and 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 jobs executing in the background. This command is supported only if job control is available.

suspend
Causes *csh* to stop as if it had been sent a *suspend* signal. Since *csh* normally ignores *suspend* signals, this is the only way to suspend the shell. This command gives an error message if attempted from a login shell. This command is supported only if job control is available.

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 only read and execute permission 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* matches nothing.

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* matches nothing.

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.

%*job* Brings the specified job into the foreground. This is supported only if job control is available.

%*job* **&**
Continues the specified job in the background. This is supported only if job control is available.

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. **Di++**).

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.

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 left 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 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 an 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 first argument. (This is equivalent to '1'.)
$	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 single quotes (') and double quotes (") can be used to prevent all or some of the remaining substitutions. Strings enclosed in single quotes 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 (see *Command Substitution* below) 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 \!* | lp'
```

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

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.

$?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 *autologout*, *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.

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 that are terminals. *Csh* will exit normally when it encounters the end-of-file condition, which is control-D typed as the first character on a command line. Setting *ignoreeof* prevents your current shell from being killed by an accidental control-D.

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, 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 execution.

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.

Command Substitution

Command substitution is indicated by a command enclosed in grave accents (` ...`). 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.

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,

../{memo,*box}

might expand to

../memo ../box ../mbox

(Note that "memo" was not sorted with the results of matching ***box**.) As a special case, {, }, and { } are passed undisturbed.

Input/Output

The standard input and standard output of a command may be redirected with the following syntax:

< *name*

Open file *name* (which is first variable, command and filename expanded) as the standard input.

<< *word*

Read the shell input up to a line which is identical to *word*. *Word* is not subjected to variable, filename or command substitution, and each input line is compared to *word* before any substitutions are done on this input line. Unless a quoting \, ", ', or ` appears in *word*, variable and command substitution is performed on the intervening lines, allowing \ to quote $, \ and `. Commands which are substituted have all blanks, tabs, and newlines preserved, except for the final newline which is dropped. The resultant text is placed in an anonymous temporary file which is given to the command as standard input.

> *name*
>! *name*
>& *name*
>&! *name*

The file *name* is used as standard output. If the file does not exist then it is created; if the file exists, it is truncated, and its previous contents are lost.

If the variable *noclobber* is set, then 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 on systems which do not support job control, the default standard input for a command run in background mode is modified to be the empty file */dev/null*.

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** 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 [Escape]
```

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.

Command Line Control

A ^R will re-print the current command line; ^W will erase the last word entered on the current command line.

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

/.cshrc	a csh script sourced (executed) at the beginning of execution by each shell.
/.login	a csh script sourced (executed) by login shell, after .cshrc at login.
/.logout	a csh script sourced (executed) by login shell, at logout.
/etc/passwd	source of home directories for name.
/bin/sh	standard shell, for shell scripts not starting with a #.
/etc/csh.login	a csh script sourced (executed) before /.cshrc and /.login when starting a csh login (analogous to **/etc/profile** in the Bourne shell).
/tmp/sh*	temporary file for <<.

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* substitutions on a single line to 20.

HARDWARE DEPENDENCIES

Series 200, 300, 500

Job control is not supported.

AUTHOR

Csh was developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

sh(1), access(2), exec(2), fork(2), pipe(2), umask(2), wait(2), tty(7), a.out(4), environ(5).

BUGS

On those systems which support job control, when a command is restarted from a stop, *csh* prints the directory it started in if it is different from the current directory; this can be misleading (i.e. wrong) as the job may have changed directories internally.

Shell built-in functions are not stoppable/restartable. Command sequences of the form "a ; b ; c" are also not handled gracefully when stopping is attempted. If you interrupt 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.

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 [SPACE]-[CNTL]-[D].

The csh metasequence ! does not work.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

csplit – context split

SYNOPSIS

csplit [**–s**] [**–k**] [**–f** prefix] file arg1 [... argn]

DESCRIPTION

Csplit reads *file* and separates it into n+1 sections, defined by the arguments *arg1... argn*. By default the sections are placed in xx00 ... xx*n* (*n* may not be greater than 99). These sections get the following pieces of *file*:

00: From the start of *file* up to (but not including) the line referenced by *arg1*.
01: From the line referenced by *arg1* up to the line referenced by *arg2*.
.
.
.
n+1: From the line referenced by *argn* to the end of *file*.

If the *file* argument is a – then standard input is used.

The options to *csplit* are:

–s *Csplit* normally prints the character counts for each file created. If the **–s** option is present, *csplit* suppresses the printing of all character counts.

–k *Csplit* normally removes created files if an error occurs. If the **–k** option is present, *csplit* leaves previously created files intact.

–f *prefix* If the **–f** option is used, the created files are named *prefix***00** ... *prefix**n***. The default is **xx00** ... xx*n*.

The arguments (*arg1* ... *argn*) to *csplit* can be a combination of the following:

/rexp/ A file is to be created for the section from the current line up to (but not including) the line containing the regular expression *rexp*. The current line becomes the line containing *rexp*. This argument may be followed by an optional + or – some number of lines (e.g., **/Page/–5**).

%*rexp*% This argument is the same as */rexp/*, except that no file is created for the section.

lnno A file is to be created from the current line up to (but not including) *lnno*. The current line becomes *lnno*.

{*num*} Repeat argument. This argument may follow any of the above arguments. If it follows a *rexp* type argument, that argument is applied *num* more times. If it follows *lnno*, the file will be split every *lnno* lines (*num* times) from that point.

Enclose all *rexp* type arguments that contain blanks or other characters meaningful to the Shell in the appropriate quotes. Regular expressions may not contain embedded new-lines. *Csplit* does not affect the original file; it is the users responsibility to remove it.

EXAMPLES

```
csplit –f cobol file '/procedure division/' /par5./ /par16./
```

This example creates four files, **cobol00** ... **cobol03**. After editing the "split" files, they can be recombined as follows:

```
cat cobol0[0–3] > file
```

Note that this example overwrites the original file.

```
csplit –k file 100 {99}
```

This example would split the file at every 100 lines, up to 10,000 lines. The **-k** option causes the created files to be retained if there are less than 10,000 lines; however, an error message would still be printed.

```
csplit -k prog.c  '%main(%'  '/^}/+1'  {20}
```

Assuming that **prog.c** follows the normal **C** coding convention of ending routines with a **}** at the beginning of the line, this example will create a file containing each separate **C** routine (up to 21) in **prog.c**.

SEE ALSO

ed(1), sh(1), regexp(5).

DIAGNOSTICS

Self explanatory except for:

arg - out of range

which means that the given argument did not reference a line between the current position and the end of the file.

INTERNATIONAL SUPPORT

8-bit data and filenames.

NAME

ct – spawn getty to a remote terminal (call terminal)

SYNOPSIS

ct [**–wn**] [**–h**] [**–v**] [**–s**speed] telno ...

DESCRIPTION

Ct dials the phone number of a modem that is attached to a terminal, and spawns a *getty* process to that terminal. *Telno* is a telephone number, with equal signs for secondary dial tones and minus signs for delays at appropriate places. If more than one telephone number is specified, *ct* will try each in succession until one answers; this is useful for specifying alternate dialing paths.

Ct will try each line listed in the file **/usr/lib/uucp/L-devices** until it finds an available line with appropriate attributes or runs out of entries. If there are no free lines, *ct* will ask if it should wait for a line, and if so, for how many minutes it should wait before it gives up. *Ct* will continue to try to open the dialers at one-minute intervals until the specified limit is exceeded. The dialogue may be overridden by specifying the **–w***n* option, where *n* is the maximum number of minutes that *ct* is to wait for a line.

Normally, *ct* will hang up the current line, so that that line can answer the incoming call. The **–h** option will prevent this action. If the **–v** option is used, *ct* will send a running narrative to the standard error output stream.

The data rate may be set with the **–s** option, where *speed* is expressed in baud. The default rate is 300.

After the user on the destination terminal logs out, *ct* prompts, **Reconnect?** If the response begins with the letter **n** the line will be dropped; otherwise, *getty* will be started again and the **login:** prompt will be printed.

Of course, the destination terminal must be attached to a modem that can answer the telephone.

FILES

/usr/adm/ctlog
/usr/lib/uucp/L-devices

SEE ALSO

cu(1), getty(1M), login(1), uucp(1).

NAME

ctags – create a tags file

SYNOPSIS

ctags [**–xvFBatwu**] names...

DESCRIPTION

Ctags makes a tags file for *ex*(1) (or *vi*(1)) from the specified C, Pascal and Fortran sources. A *tags* file gives the locations of specified objects (for C, functions, macros with argments, and typedefs; Pascal, procedures, programs and functions; FORTRAN, subroutines, progams and functions) in a group of files. Each line of the *tags* file contains the object name, the file in which it is defined, and an address specification for the object definition. All objects except C *typedefs* are searched with a pattern, *typedefs* with a line number. Specifiers are given in separate fields on the line, separated by blanks or tabs. Using the *tags* file, *ex* can quickly find these objects' definitions.

–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. No *tags* file is created or changed.

–v produces a page index on the standard output. This listing contains the function name, file name, and page number within that file (assuming 56 line pages to match *pr*(1)). Since the output will be sorted into lexicographic order, it may be desired to run the output through **sort –f**. Sample use:

```
ctags –v files | sort –f > index
pr index files
```

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.

Other options are:

–F use forward searching patterns (/.../) (default).

–B use backward searching patterns (?...?).

–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 and then only in very special circumstances.

–t create tags for typedefs.

–w suppressing warning diagnostics.

–u causing the specified files to be *updated* in *tags*, that is, all references to those files are deleted, and the new values are added to the file as in **-a** above. (Beware: this option is implemented in a way which is rather slow; it is usually faster to simply rebuild the *tags* file.)

The tag *main* is treated specially in C programs. The tag formed is created by prepending *M* to the name of the file, with a trailing .c removed, if any, and leading pathname components also removed. This makes use of *ctags* practical in directories with more than one program.

RETURNS

Too many entries to sort.

An attempt to get additional heap space failed; the sort could not be performed.

Duplicate entry in file *file*, line *line*: *name*.
Second entry ignored.

The same name was detected twice in the same file. A *tags* entry was made only for the first name found.

Duplicate entry in files *file1* and *file2*: *name* (Warning only).

The same name was detected in two different files. A *tags* entry was made only for the first name found.

WARNINGS

Recognition of **functions**, **subroutines** and **procedures** for FORTRAN and Pascal is done in a very simple way. No attempt is made to deal with block structure; if there are two Pascal procedures in different blocks with the same name a warning message will be generated.

The method of deciding whether to look for C or Pascal and FORTRAN functions is an approximation and can be fooled by unusual programs.

It does not know about **#ifdefs** and Pascal types.

It relies on the input being well formed to detect *typedefs*.

Use of -tx shows only the last line of typedefs.

Ex(1) is naive about *tags* files with several identical tags; it simply chooses the first entry its (non-linear) search finds with that tag. Such files can be created with either the **-u** or **-a** options or by editing a *tags* file.

If more than one (function) definition appears on a single line, only the first definition will be indexed.

AUTHOR

Ctags was developed by the University of California, Berkeley.

FILES

tags	output tags file
OTAGS	temporary used by -u

SEE ALSO

ex(1), vi(1).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

cu – call another (UNIX) system; terminal emulator

SYNOPSIS

cu [**–s***speed*] [**–l***line*] [**–h**] [**–q**] [**–t**] [**–d**] [**–e**|**–o**] [**–m**] [**–n**] **telno** | **systemname** | **dir**

DESCRIPTION

Cu calls up another system, which will usually be a UNIX operating system, but may be a terminal or a non-UNIX operating system. It manages an interactive conversation with possible transfers of ASCII files. *Cu* accepts the following options and arguments.

–s*speed* Specifies the transmission speed (110, 150, 300, 600, 1200, 2400, 3600, 4800, 7200, 9600, 19200). The default value is 300.

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.

–l*line* Specifies a device name to use as the communication line. This can be used to override searching for the first available line having the right speed. When the **-l** option is used without the **-s** option, the speed of a line is taken from the file **/usr/lib/uucp/L-devices**. When the **-l** and **-s** options are used simultaneously, *cu* will search the L-devices file to check if the requested speed for the requested line is available. If so, the connection will be made at the requested speed; otherwise an error message will be printed and the call will not be made. The specified device is generally a directly connected asynchronous line (e.g., **/dev/ttyab**). In this case a phone number is not required but the string *dir* can be used to specify that a dialer is not required. If the specified device is associated with an auto dialer, a phone number must be provided.

–h Emulates local echo, supporting calls to other computer systems which expect terminals to be set to half-duplex mode.

–q Invokes the use of ENQ/ACK handshake. (Remote sends ENQ, *cu* sends ACK.)

–t Used when dialing an ASCII terminal which has been set to auto answer. Appropriate mapping of carriage-return to carriage-return-line-feed pairs is set.

–d Causes diagnostic traces to be printed.

–e(**–o**) Designates that even (odd) parity is to be generated for data sent to the remote.

–m Designates a direct line which has modem controls. The modem controls are to be ignored by *cu*.

–n Will request the phone number to be dialed from the user rather than taking it from the command line.

telno When using an automatic dialer, the argument is the telephone number with equal signs for secondary dial tone or minus signs for delays, at appropriate places.

systemname A uucp system name may be used rather than a phone number; in this case, *cu* will obtain an appropriate direct line or phone number from **/usr/lib/uucp/L.sys** (the appropriate baud rate is also read along with phone numbers). *Cu* will try each phone number or direct line for *systemname* in the L.sys file until a connection is made or all the entries are tried.

dir Using *dir* ensures that *cu* will use the line specified by the **-l** option.

After making the connection, *cu* runs as two processes: the *transmit* process reads data from the standard input and, except for lines beginning with ~, passes it to the remote system; the *receive*

process accepts data from the remote system and, except for lines beginning with ~, passes it to the standard output. Normally, an automatic DC3/DC1 protocol is used to control input from the remote so the buffer is not overrun. **Prompt handshaking** can be used to control transfer of ASCII files to systems that have no *type-ahead* capability but require data to be sent only after a prompt is given. This is described in detail below. Lines beginning with ~ have special meanings.

The *transmit* process interprets the following:

~. and ~.. terminate the conversation. On a hardwired line (only), ~. sends several EOF characters to log out the session; ~.. will suppress the EOF sequence. In general the remote hardwired machine will be unaware of the disconnect if ~.. is used. ~. and ~.. do not differ for dialup connections.

~! escape to an interactive shell on the local system.

~!*cmd*... run *cmd* on the local system (via **sh** –**c**).

~& similar to ~! but kill the receive process, restarting it upon return from the shell. This is useful for invoking sub-processes that read from the communication line, where the receive process would be otherwise competing for input.

~&*cmd*... run *cmd* on the local system (via **sh** –**c**) and kill the receive process, restarting it later.

~$*cmd*... run *cmd* locally and send its output to the remote system.

~%**cd** change the directory on the local system. **NOTE: ~!cd will cause the command to be run by a sub-shell; probably not what was intended.**

~%**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.

~%**break** transmit a **BREAK** to the remote system.

~%**nostop** toggles between DC3/DC1 input control protocol and no input control. This is useful in case the remote system is one which does not respond properly to the DC3 and DC1 characters.

~%<*file* sends the contents of the local file to the remote system using **prompt handshaking.** The specified file is read a line at a time, and each line is sent to the remote system when the **prompt sequence** is received. If no prompt is received by the time the **prompt timeout** occurs, the line is sent anyway. If the timeout is set to 0 seconds, or if the first character in the prompt sequence is a null character (^@), the handshake will always appear to be satisfied immediately, regardless of whether or not the remote system generates a prompt. This capability is intended mainly to facilitate transfer of ASCII files from HP-UX to an HP3000 system running *MPE*. This is usually accomplished by running the *MPE* utility *FCOPY*, and giving the command "from=;to=destfile;new" and then running the *cu* input diversion to send the file to *FCOPY*, which saves it in "destfile." This facility may also be useful with other systems, an HP1000 running *RTE*, for example.

~%**setpt** *n* this specifies the number of seconds to wait for a prompt before giving up. The default is 2 seconds. Specifying a timeout of 0 seconds will disable handshaking,

i.e., handshake will appear to complete immediately.

~%setps *xy* set the handshake prompt to the characters *xy*. The default is DC1. The prompt may be any one or two characters. A control character X, i.e., Control-X, is specified with a caret (ASCII 94) preceding the character, i.e., ^X. A null character may be specified with ^@. (A null first character in the prompt implies a "null" prompt, which always appears to be satisfied.) A caret is specified by ^^.

~%>[>]*file* divert output from the remote system to the specified file until another **~%>** command is given. When an output diversion is active, typing **~%>** will terminate it, and **~%>** *anotherfile* will terminate it and begin a new one. The output diversion remains active through a **~&** subshell, but unpredictable results can occur if input/output diversions are intermixed with **~%take** or **~%put.** The **~%>>** command will append to the named file. Note that these commands, which are interpreted by the transmit process, are unrelated to the **~>** commands described below, which are interpreted by the receive process.

If the implementation supports HP-UX job control (see *csh*(1)), the following additional command is available:

~*susp* Suspend the *cu* session. *Susp* is the suspend character that was in use when *cu* was invoked (usually ^Z) (see *stty*(1).)

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.

When *cu* is used on system X to connect to system Y and subsequently used on system Y to connect to system Z, commands on system Y can be executed if **~~** is used. For example, uname can be executed on Z, X, and Y as follows:

```
uname
Z
~!uname
X
~~!uname
Y
```

In general, **~** causes the command to be executed on the original machine; **~~** causes the command to be executed on the next machine in the chain.

EXAMPLES

To dial a system whose number is 9 201 555 1212 using 1200 baud:

```
cu -s1200 9=2015551212
```

If the speed is not specified, 300 is the default value.

To login to a system connected by a direct line:
cu -l/dev/ttyXX dir

To dial a system with the specific line and a specific speed:
cu -s1200 -l/dev/ttyXX dir

To dial a system using a specific line:
cu -l/dev/culXX 2015551212

To use a system name:
cu YYYZZZ

To connect directly to a modem:
cu -l/dev/culXX -m dir

FILES

/usr/lib/uucp/L.sys
/usr/lib/uucp/L-devices
/usr/spool/uucp/LCK..(tty-device)
/dev/null

SEE ALSO

cat(1), ct(1), echo(1), stty(1), uname(1), uucp(1).

DIAGNOSTICS

Exit code is zero for normal exit, non-zero (various values) otherwise.

WARNINGS

Cu buffers input internally.

There is an artificial slowing of transmission by *cu* during the **~%put** operation so that loss of data is unlikely.

AUTHOR

Cu was developed by AT&T and HP.

INTERNATIONAL SUPPORT

8- and 16-bit data.

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 ...]

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*.

Cut does not expand tabs. Input should be piped through **expand**(1) if tab expansion is required.

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.

DIAGNOSTICS

line too long — A line can have no more than 1023 characters or fields.

bad list for c/f option — Missing –**c** or –**f** option or incorrectly specified *list*. No error occurs if a line has fewer fields than the *list* calls for.

no fields — The *list* is empty.

SEE ALSO

grep(1), paste(1).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

cxref – generate C program cross-reference

SYNOPSIS

cxref [options] files

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; does not print input file names.
–t	Format listing for 80-column width.

HARDWARE DEPENDENCIES

Series 200, Series 300, Series 500

Cxref uses a special version of the C compiler front end. The size of the internal compiler tables can be adjusted by using the **-W***c* and **-N** options, as described in the manual page for *cc*(1).

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, anyway.

BUGS

Cxref considers a formal argument in a *#define* macro definition to be a declaration of that symbol. For example, a program that *#includes* **ctype.h** will contain many declarations of the variable **c**.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

date – print and set the date

SYNOPSIS

date [mmddhhmm[yy]] [+format]

DESCRIPTION

If no argument is given, or if the argument begins with +, the current date and time are printed. Otherwise the current date is set, provided you are super-user. 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 a.m. 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 by using the TZ environment variable. The rules applied are the converse of the rules described in *ctime*(3C). Inspection of the form of the TZ variable will determine the setting of the *tz_dsttime* variable if *gettimeofday*(2) is supported on a given system. Only no conversion and USA standard daylight conversion are defined for this field. Other conversions defined by TZ will be treated as no conversion for this purpose.

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). Numeric 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 **/etc/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	day of year – 001 to 366
w	day of week – Sunday = 0
a	abbreviated weekday name – Sun to Sat
W	full weekday name – Sunday to Saturday
h	abbreviated month – Jan to Dec
F	full month name – January to December
r	time in a.m./p.m. notation
z	time zone name from TZ variable in user's environment

The full or abbreviated month name and full or abbreviated weekday name are spelled according to user's native language as defined by the LANG variable, see *environ*(5).

If there is no argument, the current date and time are printed according to the D_T_FMT string (see *langinfo*(3C)) that corresponds to the current value of the variable LANG in the user's environment. If the LANG variable is not set, *ctime*(3C) is used to format the date.

HARDWARE DEPENDENCIES

Series 500

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

For example:

```
date '+DATE: %m/%d/%y%nTIME: %H:%M:%S'
```

would have generated the following as output:

```
DATE: 08/01/76
TIME: 14:45:05
```

FILES

/etc/wtmp

SEE ALSO

ctime(3C), langinfo(3C), printf(3S), environ(5).

DIAGNOSTICS

No permission if you are not 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.

WARNING

It is a bad practice to change the date while the system is running multi-user.

AUTHOR

Date was developed by AT&T and HP.

INTERNATIONAL SUPPORT

8-bit data, messages.

NAME

dc – desk calculator

SYNOPSIS

dc [file]

DESCRIPTION

Dc is an arbitrary precision arithmetic package. Ordinarily it operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained. (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. The remainder is calculated according to the current scale factor; it is not the integer modulus function. **7 % 3** yeilds .1 (one tenth) if scale is 1 because **7 / 3** is 2.3 with .1 as the remainder.

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* **!<***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 appropriate relational operator above 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. See below for notes on output base.

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 *bc* for array operations. **Y** generates debugging output for *dc* itself.

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 the letters A–F representing the values from 10 through 16. Bases 0 and 1 generate a string of **1**'s whose length is the value of the number. Base -1 generates a similar 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. "Digits" are separated by spaces. Given the definition of output base, the command **Op** will always yield "10" (in a representation appropriate to the base); **O1-p** yields useful information about the output base.

EXAMPLE

This example prints the first ten values of n! (n factorial):

```
[la1+dsa*pla10>y]sy
0sa1
lyx
```

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] ...

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 blocks before starting copy
seek=*n*	seek *n* blocks from beginning of output file before copying
count=*n*	copy only *n* input blocks
conv=ascii	convert EBCDIC to ASCII
ebcdic	convert ASCII to EBCDIC
ibm	slightly different map of ASCII to EBCDIC
lcase	map alphabetics to 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 block 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 newline 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 block 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 block into the ASCII file **x**:

```
dd if=/dev/rmt/0m 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 block sizes.

SEE ALSO

cp(1), tr(1).

DIAGNOSTICS

f+p blocks in(out) numbers of full and partial blocks 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/c0
```

for output to a cartridge tape, and

```
tcio -ivS 256 /dev/rct/c0 | 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.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

delta – make a delta (change) to an SCCS file

SYNOPSIS

delta [–**r**SID] [–**s**] [–**n**] [–**g**list] [–**m**[mrlist]] [–**y**[comment]] [–**p**] files

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*.

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*(4)) and will cause an error.

A *get* of many SCCS files, followed by a *delta* of those files, should be avoided when the *get* generates a large amount of data. Instead, multiple *get/delta* sequences should be used.

If the standard input (-) is specified on the *delta* command line, the **-m** (if necessary) and **-y** keyletters *must* also be present. Omission of these keyletters causes an error to occur.

Comments are limited to text strings of at most 512 characters.

SEE ALSO

admin(1), bdiff(1), cdc(1), get(1), help(1), prs(1), rmdel(1), sccsfile(4).

SCCS User's Guide in *HP-UX Concepts and Tutorials.*

DIAGNOSTICS

Use *help*(1) for explanations.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

deroff – remove nroff/troff, tbl, and eqn constructs

SYNOPSIS

deroff [**–mx**] [**–w**] [files]

DESCRIPTION

Deroff reads each of the *files* in sequence and removes all *troff* requests, macro calls, backslash constructs, *eqn* constructs (between **.EQ** and **.EN** lines, and between delimiters), and *tbl*(1) descriptions, perhaps replacing them with white space (blanks and blank lines), and writes the remainder of the file on the standard output. *Deroff* follows chains of included files (**.so** and **.nx** *troff* commands); if a file has already been included, a **.so** naming that file is ignored and a **.nx** naming that file terminates execution. If no input file is given, *deroff* reads the standard input.

The **–m** option may be followed by an **m**, **s**, or **l**. The **–mm** option causes the macros 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

nroff(1), tbl(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.

INTERNATIONAL SUPPORT

8-bit filenames.

NAME

diff, diffh – differential file comparator

SYNOPSIS

diff [**–befh**] file1 file2

/usr/lib/diffh file1 file2

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 options are:

–b causes trailing blanks (spaces and tabs) to be ignored and other strings of blanks to compare equal.

–e produces a script of *a, c* and *d* commands for the editor *ed*, which will recreate *file2* from *file1*.

–f produces a script similar to that of **–e**, only it is not useful with *ed*, and it is in the opposite order.

–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**.

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.

FILES

/tmp/d?????
/usr/lib/diffh for **–h**

SEE ALSO

bdiff(1), 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 (.).

WARNINGS

Missing newline at end of file X indicates that the last line of file X did not have a new-line. If the lines are different, they will be flagged and output, although the output will seem to indicate they are the same.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

diff3 – 3-way differential file comparison

SYNOPSIS

diff3 [**–ex3**] file1 file2 file3

DESCRIPTION

Diff3 compares three versions of a file, and publishes disagreeing ranges of text flagged with these codes:

====	all three files differ
====1	*file1* is different
====2	*file2* is different
====3	*file3* is different

The type of change suffered in converting a given range of a given file to some other is indicated in one of these ways:

f **:** *n1* **a**	Text is to be appended after line number *n1* in file *f*, where *f* = 1, 2, or 3.
f **:** *n1* **,** *n2* **c**	Text is to be changed in the range line *n1* to line *n2*. If *n1* = *n2*, the range may be abbreviated to *n1*.

The original contents of the range follows immediately after a **c** indication. When the contents of two files are identical, the contents of the lower-numbered file is suppressed.

Under the **–e** option, *diff3* publishes a script for the editor *ed* that will incorporate into *file1* all changes between *file2* and *file3*, i.e., the changes that normally would be flagged ==== and ====3. Option **–x** (**–3**) produces a script to incorporate only changes flagged ==== (====3). The following command will apply the resulting script to *file1*.

```
(cat script; echo '1,$p') | ed – file1
```

FILES

/tmp/d3*
/usr/lib/diff3prog

SEE ALSO

diff(1).

BUGS

Text lines that consist of a single . will defeat **–e**.
Files longer than 64K bytes will not work.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

diffmk – mark differences between files

SYNOPSIS

diffmk name1 name2 name3

DESCRIPTION

Diffmk compares two versions of a file and creates a third file that includes "change mark" commands for *nroff*(1) or *troff*. *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** commands. 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*(1) 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.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

dircmp – directory comparison

SYNOPSIS

dircmp [**–d**] [**–s**] [**–w***n*] dir1 dir2

DESCRIPTION

Dircmp examines *dir1* and *dir2* and generates various tabulated information about the contents of the directories. Listings of files that are unique to each directory are generated for all the options. If no option is entered, a list is output indicating whether the filenames common to both directories have the same contents.

–d Compare the contents of files with the same name in both directories and output a list telling what must be changed in the two files to bring them into agreement. The list format is described in *diff*(1).

–s Suppress messages about identical files.

–w*n* Change the width of the output line to *n* characters. The default width is 72.

SEE ALSO

cmp(1), diff(1).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

dos2ux, ux2dos – convert ASCII file format

SYNOPSIS

dos2ux file ...
ux2dos file ...

DESCRIPTION

Dos2ux and *ux2dos* read each *file* in sequence and write it on standard output, converting to DOS format or to HP-UX format. *File* can be either DOS format or HP-UX format for either command. Thus:

dos2ux file

prints the file on the terminal screen, while:

ux2dos file1 file2 >file3

converts *file1* and *file2*, to DOS format, then concatenates them together, placing them in *file3*.

If no input file is given or if the argument – is encountered, *dos2ux* and *ux2dos* read from standard input, enabling you to combine standard input with other files.

RETURN VALUE

Both commands return 0 if successful, 2 if the command failed. The only possible failure is the inability to open a specified file, in which case a warning is printed.

WARNING

Command formats resembling

dos2ux file1 file2 >file1

overwrite the data in *file1* before the concatenation begins, causing a loss of the contents of *file1*. Therefore, take care when using shell special characters.

NAME

doschmod – change attributes of a DOS file

SYNOPSIS

doschmod mode device:file ...

DESCRIPTION

Doschmod is the DOS counterpart of *chmod*(1).

A DOS file name is recognized by the presence of an embedded colon (:) delimiter (see *dosif*(4) for DOS file naming conventions).

The attributes of each named file are changed according to *mode*, which is an octal number in the range 00 to 0377. **mode** is constructed from the logical OR of the following modes:

040	Archive bit. This bit is set whenever the file has been written to and closed.
020	This bit defines a sub-directory.
010	This bit signifies that the volume label is contained in the first 11 bytes.
004	System file
002	Hidden file
001	File is marked read only

Doschmod does not restrict the value of **mode**; however, some values will render the file inaccessible to other utilities and operating systems, so special care should be taken in selecting values for **mode**.

EXAMPLES

The first of the following examples marks *file* as a directory. The second makes *file* read-only:

doschmod 020 file

doschmod 041 file

SEE ALSO

dosif(4), chmod(1), chmod(2).

Series 300 Only

NAME

doscp – copy to or from DOS files

SYNOPSIS

doscp file1 file2
doscp file1 [file2...] directory

DESCRIPTION

Doscp is the DOS counterpart of *cp*(1). *Doscp* copies a DOS or HP-UX file to a DOS or HP-UX file, or a list of HP-UX or DOS files to a directory. The last name on the argument list is the destination file or directory.

A DOS file name is recognized by the presence of an embedded colon (:) delimiter (see *dos*(4) for DOS file naming conventions).

The DOS file naming conventions are known only by the DOS utilities. Since file name expansion is done by the shell, this mechanism cannot be used for expansion of DOS file names.

The file name "–" (dash) is interpreted to mean standard input or standard output depending upon its position in the argument list.

Important

Do not mount media before using *doscp*.

EXAMPLES

doscp abc /dev/hd.c:x/y/z

Copy the HPUX file *abc* to the DOS file *x/y/z* through the HP-UX device special file */dev/hd.c*

doscp /dev/fd.0:/backup/log logcopy

Copy DOS file */backup/log* through the HP-UX device special file */dev/fd.0* to HP-UX file *logcopy* located in the current directory.

doscp bb:zulu –

Copy DOS file *zulu* on the DOS volume stored as HP-UX file *bb* to standard output.

SEE ALSO

dos(4), cp(1).

DIAGNOSTICS

Doscp returns 0 if the file is copied successfully. Otherwise, it prints a diagnostic and returns with a non-zero value.

Series 300 Only

NAME

dosdf – report number of free disk clusters

SYNOPSIS

dosdf [file-systems]

DESCRIPTION

Dosdf is the DOS counterpart of *df*(1). It prints out the cluster size in bytes and the number of free clusters on the DOS disc.

SEE ALSO

dos(4), df(1).

Series 300 Only

NAME

dosls, dosll – list contents of DOS directories

SYNOPSIS

dosls [**–adl**] [names]
dosll [**–adl**] [names]

DESCRIPTION

Dosls is the DOS counterpart of *ls*(1).

A DOS file name is recognized by the presence of an embedded colon (:) delimiter (see *dos*(4) for DOS file naming conventions).

For each directory named, *dosls* lists the contents of that directory; for each file named, *dosls* repeats its name and any other information requested.

If you are super-user, *doslss* defaults to listing all files except **.** (current directory) and **..** (parent directory). If invoked by the name *dosll*, the **–l** option is implied.

There are several options:

–a List all entries. In the absence of this option, hidden files, system files and entries whose names begin with a dot (**.**) are **not** listed.

–A Same as **–a**, except that the current directory and the parent directory are not listed. For super-user, this flag defaults to ON, and is disabled by **–A.**

–d If argument is a directory, list only its name. Often used with **–l** to get the status of a directory.

–l List in long format, giving file attribute, size in bytes, and the date and time of last modification for each file. Long listing is disabled if *dosll* is invoked with the **–l** option.

EXAMPLES

The examples that follow assume that an DOS directory structure exists on the device accessed through HP-UX device special file */dev/fd.0*.

This example lists all of the files in the root directory of the DOS directory structure:

dosls –a /dev/fd.0:

This second example produces a long-format listing of all the information about the DOS directory */users/root* but does not list the files in the directory:

dosls –ld /dev/fd.0:/users/root

IMPORTANT REMINDER

To obtain a listing of DOS files on the device accessed HP-UX device special file */dev/fd*, be sure to include the colon as in

dosls /dev/fd:.

If the colon is omitted as in

dosls /dev/fd,

a listing of the HP-UX file */dev/fd* is produced, not the contents of the DOS directory accessed through */dev/fd*.

SEE ALSO

dos(4), ls(1).

Series 300 Only

NAME

dosmkdir – make a DOS directory

SYNOPSIS

dosmkdir device:dirname ...

DESCRIPTION

Dosmkdir is the DOS counterpart of *mkdir*(1).

A DOS file name is recognized by the presence of an embedded colon (:) delimiter (see *dos*(4) for DOS file naming conventions).

Dosmkdir creates specified directories. The standard entries, **.** for the directory itself and **..** for its parent, are made automatically.

EXAMPLES

To create an empty subdirectory named *sysmods* under the directory */usr/lib* on HP-UX device */dev/dos2*, use:

dosmkdir /dev/dos2:/usr/lib/sysmods

SEE ALSO

dos(4), mkdir(1).

DIAGNOSTICS

Dosmkdir returns 0 if all directories were successfully created. Otherwise, it prints a diagnostic and returns non-zero.

Series 300 Only

NAME

dosrm, dosrmdir – remove DOS files or directories

SYNOPSIS

dosrm [**–fri**] device:file ...

dosrmdir device:dir ...

DESCRIPTION

Dosrm and *dosrmdir* are DOS counterparts of *rm*(1) and *rmdir*(1), respectively.

A DOS file name is recognized by the presence of an embedded colon (:) delimiter (see *dos*(4) for DOS file naming conventions).

Dosrm removes the entries for one or more files from a directory.

If a designated file is a directory, an error comment is printed (unless the optional argument **–r** has been used as described below).

The options are:

–f Unconditionally removes the specified file, even if the file is marked read-only.

–r Causes *dosrm* to recursively delete the entire contents of a directory, followed by the directory itself. *Dosrm* can recursively delete up to 17 levels of directories.

–i Causes *dosrm* to ask whether or not to delete each file. If **–r** is also specified, *dosrm* asks whether to examine each directory encountered.

Dosrmdir Removes entries for the named directories, provided they are empty.

EXAMPLES

The following examples assume that an DOS directory structure exists on the HP-UX device file */dev/dosdisc*.

This example recursively combs through the DOS directory */tmp* and asks if each DOS file should be removed (forced, with no file mode checks):

dosrm –irf /dev/dosdisc:/tmp

This example removes the DOS directory */users/doug*:

dosrmdir sc:/users/doug

SEE ALSO

dos(4), rm(1), rmdir(1).

NAME

du – summarize disk usage

SYNOPSIS

du [**–sar**] [names]

DESCRIPTION

Du gives the number of 512-byte 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*.

INTERNATIONAL SUPPORT

8-bit filenames.

NAME

echo – echo (print) arguments

SYNOPSIS

echo [arg] ...

DESCRIPTION

Echo writes its arguments separated by blanks and terminated by a new-line on the standard output. It also understands C-like escape conventions; beware of conflicts with the shell's use of \:

\b	backspace
\c	print line without appending a new-line
\f	form-feed
\n	new-line
\r	carriage return
\t	tab
\v	vertical tab
****	backslash
*****n*	the 8-bit character whose ASCII code is the 1-, 2-, 3- or 4-digit octal number *n*, whose first character must be a zero.

Echo is useful for producing diagnostics in command files and for sending known data into a pipe.

SEE ALSO

sh(1).

NOTES

Berkeley **echo** differs from this implementation. The former does not implement the backslash escapes. However, the semantics of the **\c** escape can be obtained by using the **-n** option. The echo command implemented as a built-in function of *csh*(1) follows the Berkeley semantics.

BUGS

No characters are printed after the first **\c**. This is not normally a problem.

INTERNATIONAL SUPPORT

8- and 16-bit data.

NAME

ed, red – text editor

SYNOPSIS

ed [–] [**–p** string] [**–x**] [file]

red [–] [**–p** string] [**–x**] [file]

REMARKS

The decryption facilities provided by this software are under control by the United States Government and cannot be exported without special licenses. The capabilities are only available by special arrangement through HP.

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*. **–p** option allows the user to specify a prompt string. If **–x** is present, an *x* command is simulated first to handle an encrypted file. *Ed* operates on a copy of the file it is editing; changes made to the copy have no effect on the file until a *w* (write) command is given. The copy of the text being edited resides in a temporary file called the *buffer*. There is only one buffer.

Red is a restricted version of *ed*. It will only allow editing of files in the current directory. It prohibits executing shell commands via *!shell command*. Attempts to bypass these restrictions result in an error message (*restricted shell*).

Both *ed* and *red* support the *fspec*(4) formatting capability. After including a format specification as the first line of *file* and invoking *ed* with your terminal in **stty –tabs** or **stty tab3** mode (see *stty*(1), the specified tab stops will automatically be used when scanning *file*. For example, if the first line of a file contained:

<:t5,10,15 s72:>

tab stops would be set at columns 5, 10, and 15, and a maximum line length of 72 would be imposed. NOTE: while inputting text, tab characters when typed are expanded to every eighth column as is the default.

Commands to *ed* have a simple and regular structure: zero, one, or two *addresses* followed by a single-character *command*, 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 REs* match a *single* character:

1.1 An ordinary character (*not* one of those discussed in 1.2 below) is a one-character RE that matches itself.

1.2 A backslash (\) followed by any special character 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 is a RE that matches whatever the one-character RE matches.

2.2 A one-character RE followed by an asterisk (*) is a RE that matches *zero* or more occurrences of the one-character RE. If there is any choice, the longest leftmost string that permits a match is chosen.

2.3 A one-character RE followed by \{*m*\}, \{*m*,\}, or \{*m*,*n*\} is a RE that matches a *range* of occurrences of the one-character RE. The values of *m* and *n* must be 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.

2.4 The concatenation of REs is a RE that matches the concatenation of the strings matched by each component of the RE.

2.5 A RE enclosed between the character sequences \(and \) is a RE that matches whatever the unadorned RE matches.

2.6 The expression *n* matches the same string of characters as was matched by an expression enclosed between \(and \) *earlier* in the same RE. Here *n* is a digit; the sub-expression specified is that beginning with the *n*-th occurrence of \(counting from the left. For example, the expression **^\(.*\)\1$** matches a line consisting of two repeated appearances of the same string.

Finally, an *entire RE* may be constrained to match only an initial segment or final segment of a line (or both).

3.1 A 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 **l**, **n**, or **p** in which case the current line is either listed, numbered or printed, respectively, as discussed below under the *l*, *n*, and *p* commands.

(.)a
<text>
.

The *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. The maximum number of characters that may be entered from a terminal is 256 per line (including the new-line character).

(.)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* is replaced by !, the rest of the line is taken to be a shell (*sh*(1)) command whose output is to be read. Such a shell command is *not* remembered as the current file name. See also **DIAGNOSTICS** below.

E *file*

The *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. The maximum number of characters that may be entered from a terminal is 256 per line (including the new-line character).

(.,.+1)j

The *j*oin command joins contiguous lines by removing the appropriate new-line characters. If exactly 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* is replaced by **!**, the rest of the line is taken to be a shell (*sh*(1)) command whose output is to be read. For example, "**$r !ls**" appends current directory to the end of the file being edited. Such a shell command is *not* remembered as the current file name.

(**.**,**.**)**s**/*RE*/*replacement*/
or
(**.**,**.**)**s**/*RE*/*replacement*/**g**
or
(**.**,**.**)**s**/*RE*/*replacement*/**n**
n = 1-512

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. If a number n appears after the command, only the n th occurrence of the matched string on each addressed line is replaced. It is an error for the substitution to fail on *all* addressed lines. Any character other than space or 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* is replaced by !, the rest of the line is taken to be a shell (*sh*(1)) command whose standard input is the addressed lines. Such a shell command is *not* remembered as the current file name.

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?

HARDWARE DEPENDENCIES

Series 500

Certain older interface cards do not support **tty -tabs** or **stty tab3**. This precludes the use of the *fspec*(4) formatting capability.

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), edit(1), ex(1), grep(1), sed(1), sh(1), stty(1), vi(1), fspec(4), regexp(5).

The ed Editor, in *HP-UX: Selected Articles.*

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.

CAVEATS AND 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 a new-line character.
The *l* command mishandles DEL.
Files encrypted directly with the *crypt*(1) command with the null key cannot be edited.
If the editor input is coming from a command file (i.e., ed file < ed-cmd-file), the editor will exit at the first failure of a command that is in the command file.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

edit – text editor (variant of ex for casual users)

SYNOPSIS

edit [**–r**] name ...

DESCRIPTION

Edit is a variant of the text editor *ex* recommended for new or casual users who wish to use a command-oriented editor. The following brief introduction should help you get started with *edit*. If you are using a CRT terminal you may want to learn about the display editor *vi*.

BRIEF INTRODUCTION

To edit the contents of an existing file you begin with the command "edit name" to the shell. *Edit* makes a copy of the file which you can then edit, and tells you how many lines and characters are in the file. To create a new file, just make up a name for the file and try to run *edit* on it; you will cause an error diagnostic, but do not worry.

Edit prompts for commands with the character ':', which you should see after starting the editor. If you are editing an existing file, then you will have some lines in *edit's* buffer (its name for the copy of the file you are editing). Most commands to *edit* use its "current line" if you do not tell them which line to use. Thus if you say **print** (which can be abbreviated **p**) and hit carriage return (as you should after all *edit* commands) this current line will be printed. If you **delete** (**d**) the current line, *edit* will print the new current line. When you start editing, *edit* makes the last line of the file the current line. If you **delete** this last line, then the new last line becomes the current one. In general, after a **delete,** the next line in the file becomes the current line. (Deleting the last line is a special case.)

If you start with an empty file or wish to add some new lines, then the **append** (**a**) command can be used. After you give this command (typing a carriage return after the word **append**) *edit* will read lines from your terminal until you give a line consisting of just a ".", placing these lines after the current line. The last line you type then becomes the current line. The command **insert** (**i**) is like **append** but places the lines you give before, rather than after, the current line.

Edit numbers the lines in the buffer, with the first line having number 1. If you give the command "1" then *edit* will type this first line. If you then give the command **delete** *edit* will delete the first line, line 2 will become line 1, and *edit* will print the current line (the new line 1) so you can see where you are. In general, the current line will always be the last line affected by a command.

You can make a change to some text within the current line by using the **substitute** (**s**) command. You say "s/*old*/*new*/" where *old* is replaced by the old characters you want to get rid of and *new* is the new characters you want to replace it with.

The command **file** (**f**) will tell you how many lines there are in the buffer you are editing and will say "[Modified]" if you have changed it. After modifying a file you can put the buffer text back to replace the file by giving a **write** (**w**) command. You can then leave the editor by issuing a **quit** (**q**) command. If you run *edit* on a file, but do not change it, it is not necessary (but does no harm) to **write** the file back. If you try to **quit** from *edit* after modifying the buffer without writing it out, you will be warned that there has been "No **write** since last change" and *edit* will await another command. If you wish not to **write** the buffer out then you can issue another **quit** command. The buffer is then irretrievably discarded, and you return to the shell.

By using the **delete** and **append** commands, and giving line numbers to see lines in the file you can make any changes you desire. You should learn at least a few more things, however, if you are to use *edit* more than a few times.

The **change** (**c**) command will change the current line to a sequence of lines you supply (as in **append** you give lines up to a line consisting of only a "."). You can tell **change** to change more than one line by giving the line numbers of the lines you want to change, i.e., "3,5change". You

can print lines this way too. Thus "1,23p" prints the first 23 lines of the file.

The **undo** (**u**) command will reverse the effect of the last command you gave which changed the buffer. Thus if you give a **substitute** command which does not do what you want, you can say **undo** and the old contents of the line will be restored. You can also **undo** an **undo** command so that you can continue to change your mind. *Edit* will give you a warning message when commands you do affect more than one line of the buffer. If the amount of change seems unreasonable, you should consider doing an *undo* and looking to see what happened. If you decide that the change is ok, then you can *undo* again to get it back. Note that commands such as *write* and *quit* cannot be undone.

To look at the next line in the buffer you can just hit carriage return. To look at a number of lines hit ^D (control key and, while it is held down D key, then let up both) rather than carriage return. This will show you a half screen of lines on a CRT or 12 lines on a hardcopy terminal. You can look at the text around where you are by giving the command "z.". The current line will then be the last line printed; you can get back to the line where you were before the "z." command by saying "''". The **z** command can also be given other following characters "z–" prints a screen of text (or 24 lines) ending where you are; "z+" prints the next screenful. If you want less than a screenful of lines, type in "z.12" to get 12 lines total. This method of giving counts works in general; thus you can delete 5 lines starting with the current line with the command "delete 5".

To find things in the file, you can use line numbers if you happen to know them; since the line numbers change when you insert and delete lines this is somewhat unreliable. You can search backwards and forwards in the file for strings by giving commands of the form **/text/** to search forward for *text* or **?text?** to search backward for *text*. If a search reaches the end of the file without finding the text it wraps, end around, and continues to search back to the line where you are. A useful feature here is a search of the form **/^text/** which searches for *text* at the beginning of a line. Similarly **/text$/** searches for *text* at the end of a line. You can leave off the trailing / or ? in these commands.

The current line has a symbolic name "."; this is most useful in a range of lines as in ".,$print" which prints the rest of the lines in the file. To get to the last line in the file you can refer to it by its symbolic name "$". Thus the command "$ delete" or "$d" deletes the last line in the file, no matter which line was the current line before. Arithmetic with line references is also possible. Thus the line "$–5" is the fifth before the last, and ".+20" is 20 lines after the present.

You can find out which line you are at by doing ".=". This is useful if you wish to move or copy a section of text within a file or between files. Find out the first and last line numbers you wish to copy or move (say 10 to 20). For a move you can then say "10,20delete a" which deletes these lines from the file and places them in a buffer named *a*. *Edit* has 26 such buffers named *a* through *z*. You can later get these lines back by doing "put a" to put the contents of buffer *a* after the current line. If you want to move or copy these lines between files you can give an **edit** (**e**) command after copying the lines, following it with the name of the other file you wish to edit, i.e., "edit chapter2". By changing *delete* to *yank* above you can get a pattern for copying lines. If the text you wish to move or copy is all within one file then you can just say "10,20move $" for example. It is not necessary to use named buffers in this case (but you can if you wish).

SEE ALSO

ex(1), vi(1).

INTERNATIONAL SUPPORT

8-bit and 16-bit data, 8-bit filenames, messages.

NAME

enable, disable – enable/disable LP printers

SYNOPSIS

enable printers
disable [**–c**] [**–r**[reason]] printers

DESCRIPTION

Enable activates the named *printers*, enabling them to print requests taken by *lp*(1). Use *lpstat*(1) to find the status of printers.

Disable deactivates the named *printers*, disabling them from printing requests taken by *lp*(1). By default, any requests that are currently printing on the designated printers will be reprinted in their entirety either on the same printer or on another member of the same class. Use *lpstat*(1) to find the status of printers. Options useful with *disable* are:

–c Cancel any requests that are currently printing on any of the designated printers.

–r[*reason*] Associates a *reason* with the deactivation of the printers. This reason applies to all printers mentioned up to the next **–r** 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).

INTERNATIONAL SUPPORT

8- and 16-bit data, messages

NAME

env – set environment for command execution

SYNOPSIS

env [–] [name=value] ... [command args]

DESCRIPTION

Env obtains the current *environment*, modifies it according to its arguments, then executes 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(4), environ(5).

INTERNATIONAL SUPPORT

8-bit data and filenames.

Series 500 Only

NAME

err – report error information on last failure

SYNOPSIS

err

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 – text editor

SYNOPSIS

ex [–] [**–v**] [**–t** *tag*] [**–r**] [**–R**] [**+***command*] [**–l**] [**–x**] name ...

REMARKS

The decryption facilities provided by this software are under the control of the United States Government and cannot be exported without special licenses. These capabilities are only available by special arrangement through HP.

DESCRIPTION

Ex is the root of a family of editors including: *ex*, *edit* and *vi*. *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 a CRT terminal, you may wish to use a display based editor; in this case see *vi*(1), which is a command which focuses on the display editing portion of *ex*.

DOCUMENTATION

The *Ex Reference Manual* is a comprehensive and complete manual for the command mode features of *ex*, but you cannot learn to use the editor by reading it. For an introduction to more advanced forms of editing using the command mode of *ex*, see the editing documents written by Brian Kernighan for the editor *ed;* the material in the introductory and advanced documents works also with *ex*.

An Introduction to Display Editing with Vi introduces the display editor *vi* and provides reference material on *vi*. The *Vi Quick Reference* card summarizes the commands of *vi* in a useful, functional way, and is useful with the *Introduction*. The *vi*(1) manual page can also be used as reference.

FOR ED USERS

If you have used *ed* you will find that *ex* has a number of new features useful on CRT terminals. Intelligent terminals and high speed terminals are very pleasant to use with *vi*. Generally, the editor uses far more of the capabilities of terminals than *ed* does, and uses the terminal capability data base *terminfo*(4) and the type of the terminal you are using from the variable TERM in the environment to determine how to drive your terminal efficiently. The editor makes use of features such as insert and delete character and line in its **visual** command (which can be abbreviated **vi**) and which is the central mode of editing when using *vi*(1).

Ex contains a number of new features for easily viewing the text of the file. The **z** command gives easy access to windows of text. Hitting ^D causes the editor to scroll a half-window of text and is more useful for quickly stepping through a file than just hitting return. Of course, the screen-oriented **visual** mode gives constant access to editing context.

Ex gives you more help when you make mistakes. The **undo** (**u**) command allows you to reverse any single change which goes astray. *Ex* gives you a lot of feedback, normally printing changed lines, and indicates when more than a few lines are affected by a command so that it is easy to detect when a command has affected more lines than it should have.

The editor also normally prevents overwriting existing files unless you edited them so that you do not accidentally clobber with a *write* a file other than the one you are editing. If the system (or editor) crashes, or you accidentally hang up the phone, you can use the editor **recover** command to retrieve your work. This will get you back to within a few lines of where you left off.

Ex has several features for dealing with more than one file at a time. You can give it a list of files on the command line and use the **next** (**n**) command to deal with each in turn. The **next** command can also be given a list of file names, or a pattern as used by the shell to specify a new set of files to be dealt with. In general, filenames in the editor may be formed with full shell metasyntax. The metacharacter '%' is also available in forming filenames and is replaced by the name of

the current file.

For moving text between files and within a file the editor has a group of buffers, named *a* through *z*. You can place text in these named buffers and carry it over when you edit another file.

There is a command **&** in *ex* which repeats the last **substitute** command. In addition there is a confirmed substitute command. You give a range of substitutions to be done and the editor interactively asks whether each substitution is desired.

It is possible to ignore case of letters in searches and substitutions. *Ex* also allows regular expressions which match words to be constructed. This is convenient, for example, in searching for the word "edit" if your document also contains the word "editor."

Ex has a set of *options* which you can set to tailor it to your liking. One option which is very useful is the *autoindent* option which allows the editor to automatically supply leading white space to align text. You can then use the ^D key as a backtab and space and tab forward to align new code easily.

Miscellaneous new useful features include an intelligent **join** (**j**) command which supplies white space between joined lines automatically, commands < and > which shift groups of lines, and the ability to filter portions of the buffer through commands such as *sort*.

INVOCATION OPTIONS

The following invocation options are interpreted by *ex*:

–	Suppress all interactive-user feedback. This is useful in processing editor scripts.
–v	Invokes *vi*
–t *tag*	Edit the file containing the *tag* and position the editor at its definition.
–r *file*	Recover *file* after an editor or system crash. If *file* is not specified a list of all saved files will be printed.
–R	*Readonly* mode set, prevents accidentally overwriting the file.
+*command*	Begin editing by executing the specified editor search or positioning *command*.
–l	**LISP** mode; indents appropriately for lisp code, the () {} [[and]] commands in *vi* are modified to have meaning for *lisp*.
–x	Encryption mode; a key is prompted for allowing creation or editing of an encrypted file.

The *name* argument indicates files to be edited.

Ex States

Command	Normal and initial state. Input prompted for by **:** . Your kill character cancels partial command.
Insert	Entered by **a i** and **c**. Arbitrary text may be entered. Insert is normally terminated by line having only **.** on it, or abnormally with an interrupt.
Visual	Entered by **vi**, terminates with **Q** or ^\.

Ex command names and abbreviations

abbrev	**ab**	next	**n**	unabbrev	**una**
append	**a**	number	**nu**	undo	**u**
args	**ar**			unmap	**unm**
change	**c**	preserve	**pre**	version	**ve**
copy	**co**	print	**p**	visual	**vi**
delete	**d**	put	**pu**	write	**w**
edit	**e**	quit	**q**	xit	**x**
file	**f**	read	**re**	yank	**ya**

global	**g**	recover	**rec**	window	**z**
insert	**i**	rewind	**rew**	escape	**!**
join	**j**	set	**se**	lshift	<
list	**l**	shell	**sh**	print next	CR
map		source	**so**	resubst	**&**
mark	**ma**	stop	**st**	rshift	>
move	**m**	substitute	**s**	scroll	**^D**

Ex Command Addresses

n	line *n*	*/pat*	next with *pat*
.	current	**?***pat*	previous with *pat*
$	last	*x*-*n*	*n* before *x*
+	next	*x*,*y*	*x* through *y*
−	previous	´*x*	marked with *x*
+*n*	*n* forward	´´	previous context
%	1,$		

Initializing options

EXINIT	place **set**'s here in environment var.
$HOME/.exrc	editor initialization file
./.exrc	editor initialization file
set *x*	enable option
set no*x*	disable option
set *x*=*val*	give value *val*
set	show changed options
set all	show all options
set *x*?	show value of option *x*

Most useful options

autoindent	ai	supply indent
autowrite	aw	write before changing files
ignorecase	ic	in scanning
lisp		( ) { } are s-exp's
list		print ^I for tab, $ at end
magic		. [* special in patterns
number	nu	number lines
paragraphs	para	macro names which start ...
redraw		simulate smart terminal
scroll		command mode lines
sections	sect	macro names ...
shiftwidth	sw	for < >, and input ^D
showmatch	sm	to) and } as typed
showmode	smd	show insert mode in *vi*
slowopen	slow	stop updates during insert
window		visual mode lines
wrapscan	ws	around end of buffer?
wrapmargin	wm	automatic line splitting

Scanning pattern formation

^	beginning of line
$	end of line
.	any character
\<	beginning of word
\>	end of word
[*str*]	any char in *str*

[^*str*]	... not in *str*
[*x*–*y*]	... between *x* and *y*
*	any number of preceding

AUTHOR

Vi and *ex* are based on software developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

FILES

/usr/lib/*/*	describes capabilities of terminals
./.exrc	editor startup file
$HOME/.exrc	editor startup file
/usr/lib/exstrings	error messages
/usr/lib/exrecover	recover command
/usr/lib/expreserve	preserve command
/tmp/Ex*nnnnn*	editor temporary
/tmp/Rx*nnnnn*	named buffer temporary
/usr/preserve	preservation directory

SEE ALSO

awk(1), ctags(1), ed(1), edit(1), grep(1), sed(1), vi(1), curses(3X), term(7), terminfo(4).

CAVEATS AND 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 screen full 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.

INTERNATIONAL SUPPORT

8-bit and 16-bit data, 8-bit filenames, messages.

NAME

expand, unexpand – expand tabs to spaces, and vice versa

SYNOPSIS

expand [–tabstop] [–tab1,tab2,...,tabn] [file ...]
unexpand [**–a**] [file ...]

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.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

expr – evaluate arguments as an expression

SYNOPSIS

expr arguments

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. **expr $a : '.*/\(.*\)' \| $a**
   For $a equal to either "/usr/abc/file" or just "file", this example 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. **expr //$a : '.*/\(.*\)'**
   This is a better representation of example 2. 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 like:

```
expr = = =
```

as the arguments are passed to *expr* (and they will all be taken as the = operator). The following works:

```
expr X$a = X=
```

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

f77, fc – FORTRAN 77 compiler

SYNOPSIS

f77 [options] files
fc [options] files

DESCRIPTION

F77 is the HP-UX FORTRAN 77 compiler. It accepts several types of file arguments, see the HARDWARE DEPENDENCIES section below for exceptions:

(1) Arguments whose names end with **.f** are taken to be FORTRAN 77 source files. They are compiled, and each object file 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.

Arguments can be passed to the compiler through the **FCOPTS** environment variable as well as on the command line. The compiler picks up the value of **FCOPTS** and places its contents before any arguments on the command line. For example,

```
FCOPTS=-v
export FCOPTS
fc -L prog.f
```

is equivalent to

```
fc -v -L prog.f
```

Options

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).

–g causes the compiler to generate additional information needed for the use of a symbolic debugger. (This option may be incompatible with optimization.)

–I2 make default size of integers and logicals **INTEGER*2** and **LOGICAL*2** (same as **$OPTION SHORT**).

–I4 make default size of integers and logicals **INTEGER*4** and **LOGICAL*4**. This is the compiler's default.

–K automatically SAVE all local variables in all subprograms. This option forces static storage for these variables in order 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).

–l*x* causes the linker to search first in the library named by **/lib/lib***x***.a** and then in **/usr/lib/lib***x***.a**. (See *ld*(1).)

–L write a program listing to *stdout* during compilation.

–n causes the output file from the linker to be marked *shared*.

–N causes the output file from the linker to not be marked *shared.*

–o *outfile* name the output file from the linker *outfile* instead of **a.out**.

–onetrip execute any DO loop at least once.

–O invoke the assembly code optimizer.

–p prepare object files for profiling (see *prof*(1)).

–q causes the output file from the linker to be marked *demand load.*

–Q causes the output file from the linker to not be marked *demand load.*

–s causes the output of the linker to be *stripped* of symbol table information (see *ld*(1) and *strip*(1)). (This option is incompatible with symbolic debugging.)

–S compile the named source files and leave the assembly language output in corresponding files whose names are suffixed with **.s** (no **.o** files are created).

–t *c,name* substitute or insert subprocess *c* with *name* where *c* is one or more of an implementation-dependent set of identifiers indicating the subprocess(es). Works in two modes:
1) if *c* is a single identifier, *name* represents the full path name of the new subprocess;
2) if *c* is a set of identifiers, *name* represents a prefix to which the standard suffixes are concatenated to construct the full path names of the new subprocesses.

One or more values that *c* can assume are:
c compiler body (standard suffix is *f77comp*)
0 same as *c*
l linker (standard suffix is *ld*)

For other values that *c* can assume, see the HARDWARE DEPENDENCIES section below.

–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 enable the verbose mode, producing a step-by-step description of the compilation process on *stderr.*

–w suppress warning messages (same as **$OPTION WARNINGS OFF**).

–w66 suppress warnings about FORTRAN 66 features used.

–W *c,arg1*[*,arg2,...,argN*]
causes *arg1* through *argN* to be handed off to subprocess *c*. The *argi* are of the form *-argoption*[*,argvalue*], where *argoption* is the name of an option recognized by the subprocess and *argvalue* is a separate argument to *argoption* where necessary. The values that *c* can assume are those listed under the **–t** option, as well as *d* (driver program) which has a special meaning explained below.

The **–W d** option specification allows additional, implementation-specific options to be recognized and passed through the compiler driver to the appropriate subprocesses (see **–W** above). For example, on the Series 500:

```
-W d,-Q,dfile,-e
```

will send the options *-Q dfile* and *-e* through the compiler driver. Furthermore, a shorthand notation for this mechanism can be used by prepending **+** to the option name; as in:

+Q dfile +e

which is equivalent to the previous option expression. Note that for simplicity this shorthand is applied to each implementation-specific option individually, and that the *argvalue* is no longer separated from the *argoption* by a comma (see **−W**).

HARDWARE DEPENDENCIES

Series 200, 300:

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.

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.

The **−t** and **−W** option have additional values that *c* can assume:

r	ratfor preprocessor (standard suffix is *ratfor*)
1	code generator (standard suffix is *f1*)
2	optimizer (standard suffix is *c2*)
a	assembler (standard suffix is *as*)
c	compiler body (standard suffix is *_f77pass1_*)

Specifying **−W1**, **−l** will cause source file line numbers to be printed as assembly code comments for debugging purposes.

The **−K** option has two side-effects: (a) all non-initialized variables are initialized to zero, and (b) the DATA statement may appear among executable statements.

The **−L** option is not implemented.

The following option is supported:

−Y enables 8- and 16-bit NLS support in strings and comments. In the default case, NLS is not enabled. The **−Y** option has an optional parameter specifying the 16-bit language name. There must be no blanks between **−Y** and the language parameter.

The implementation-specific options on the Series 200 and Series 300 are:

+A causes the compiler to align data using "old" alignment rules where non-character items 4 bytes and larger are aligned on 2-byte boundaries instead of 4-byte boundaries.

+b causes the MC68010 compiler to generate code for floating point operations that will use the 98635 floating point card if it is installed in the computer at run-time (if not installed, operations will be done in software).

+B causes the compiler to treat the backslash character (\ ) as a C-like escape character.

+f causes the MC68010 compiler to generate code for floating point operations that must use the 98635 floating point card. This code does not run unless the floating point card is installed.

+M causes the MC68020 compiler to NOT generate in-line code for the MC68881 math coprocessor. Library routines will be referenced for *matherr* capability.

+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 {**qsxcnaet**}, 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 is 150 table entries).

s maximum size of statement label table (default is 201 table entries).

x maximum size of external symbol table (default is 200 table entries).

c maximum size of control statements table (default is 20 table entries).

n maximum size of the hash table of symbols (default is 401 table entries).

a maximum size of external label name storage table (default is 10000 bytes).

e maximum number of expression tree nodes (default is 1000 entries).

t maximum size of external symbol storage table (default is 40000 bytes).

+s issue warnings for non-ANSI features (same as **$OPTION ANSI ON**).

+U upper and lower case are distinguished (case is significant). Keywords are only recognized in lower case.

+x causes the compiler to generate inline code for the MC68020 and MC68881. This is the default on MC68020 machines.

+X causes the compiler to generate "generic" MC68010 code. The code will also run on MC68020 processors, but it will not take advantage of new architectural capabilities. This is the default on MC68010 machines.

Series 500:

The following options are not implemented: **-O**, **-p**, **-S**, **-w66**.

The following option is supported:

-Y enables 8- and 16-bit NLS support in strings and comments. In the default case, NLS is not enabled.

The implementation-specific options on the Series 500 are:

+e write errors to *stderr*.

+F causes the compiler to generate information used by various program analysis programs.

+Q *dfile* specify *dfile* as the option file.

+s issue warnings for non-ANSI features (same as **$OPTION ANSI ON**).

+T causes the running program to issue a procedure traceback for runtime errors.

+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.

Series 800:

The **-w66** option is not implemented.

The **-O** option has an optional parameter specifying the optimization level. A parameter of '1' causes only level 1 optimizations to be performed, and a parameter of '2' causes all optimizations to be performed. The option **-O** is the same as **-O2**.

The implementation-specific options on the Series 800 are:

+Q *dfile* specify *dfile* as the option file.

+s issue warnings for non-ANSI features (same as **$OPTION ANSI ON**).

+T causes the running program to issue a procedure traceback for runtime errors.

FILES

Series 200, Series 300

a.out	linked executable output file
/lib/c2	assembly code optimizer
/lib/c210	assembly code optimizer (MC68010 version) linked to /usr/lib/c2 on MC68010 systems
/lib/c220	assembly code optimizer (MC68020 version) linked to /usr/lib/c2 on MC68020 systems
/usr/lib/end.o	symbolic debugger string buffer
/lib/f1	compiler pass 2
/lib/f110	compiler pass 2 (MC68010 version) linked to /usr/lib/f1 on MC68010 systems
/lib/f120	compiler pass 2 (MC68020 version) linked to /usr/lib/f1 on MC68020 systems
/usr/lib/f77pass1	compiler pass 1
/usr/lib/f77pass110	compiler pass 1 (MC68010 version) linked to /usr/lib/f77pass1 on MC68010 systems
/usr/lib/f77pass120	compiler pass 1 (MC68020 version) linked to /usr/lib/f77pass1 on MC68020 systems
file.c	input file (C source file)
file.f	input file (FORTRAN source file)
file.o	object file
file.r	input file (*ratfor* source file)
file.s	input file (assembly source file)
/lib/frt0.o	run-time startoff routine
/lib/libc.a	C library; See Section 3 of this manual
/usr/lib/libF77.a	intrinsic function library
/usr/lib/libI77.a	FORTRAN I/O library
/lib/libm.a	math library
/lib/mfrt0.o	startoff with profiling

Series 500

/usr/tmp/*	temporary files used by the compiler; names are created by *tmpnam*(3S).
a.out	linked executable output file
/usr/lib/end.o	symbolic debugger string buffer
/usr/lib/f77comp	compiler
file.f	input file (FORTRAN source file)
file.o	object file
/lib/frt0.o	runtime startup
/lib/libF77.a	FORTRAN math library
/lib/libI77.a	FORTRAN I/O library
/lib/libc.a	C library; See Section 3 of this manual
/lib/libm.a	math library

Series 800

a.out	linked executable output file
/usr/lib/f77comp	compiler
file.f	input file (FORTRAN source file)
file.o	object file
/lib/crt0.o	runtime startup
/usr/lib/libcl.a	FORTRAN math and I/O libraries
/lib/libc.a	C library; See Section 3 of this manual

/lib/libm.a math library

AUTHOR

F77 and *fc* were developed by the Hewlett-Packard Company.

SEE ALSO

as(1), asa(1), cc(1), ld(1), strip(1), matherr(3M).

DIAGNOSTICS

The diagnostics produced by *f77* 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*.

Series 500: 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.

Series 800: Errors will be written to both the listing file and *stderr* if the **−L** option is specified and if *stdout* and *stderr* are not directed to the same place.

WARNINGS

The **−s** option has a new meaning; use **+s** for non-ANSI warnings.

Series 200, Series 300: The **−U** option has a new meaning; use **+U** for case sensitivity.

The **+k** option has been removed for Series 200, Series 300 because it is the default.

Series 500: The **−Q dfile** option has a new meaning; use **+Q dfile** to specify an option file.

INTERNATIONAL SUPPORT

8- and 16-bit data only in strings and comments, 8-bit filenames.

NAME

factor, primes - factor a number, generate large primes

SYNOPSIS

factor [number]

primes [start [stop]]

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

file – determine file type

SYNOPSIS

file [**–m** mfile] [**–c**] [**–f** ffile] arg ...

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 the description of the **–V** option in *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 **–m** option instructs *file* to use an alternate magic file.

The **–c** flag 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*.

SEE ALSO

ld(1).

NAME

find – find files

SYNOPSIS

find path-name-list expression

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 *file* — True if *file* 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**, **n**, or **f** for block special file, character special file, directory, fifo (a.k.a named pipe), network special file, or plain file respectively.

−links *n* — True if the file has *n* links.

−user *uname* — True if the file belongs to the user *uname*. If *uname* is numeric and does not appear as a login name in the **/etc/passwd** file, it is taken as a user ID.

−group *gname* — True if the file belongs to the group *gname*. If *gname* is numeric and does not appear in the **/etc/group** file, it is taken as a group ID.

−size *n*[**c**] — True if the file is *n* blocks long. If *n* is followed by a **c**, the size is in characters.

−atime *n* — True if the file has been accessed in *n* days. The access time of directories in *path-name-list* is changed by *find* itself.

−mtime *n* — True if the file has been modified in *n* days.

−ctime *n* — True if the file has been changed in *n* days.

−exec *cmd* — True if the executed *cmd* returns a zero value as exit status. The end of *cmd* must be punctuated by an escaped semicolon. A command argument **{}** 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.

−cpio *device* — Always true; write the current file on *device* in *cpio*(4) format (5120-byte records).

−newer *file* — True if the current file has been modified more recently than the argument *file*.

−depth — Always true; causes descent of the directory hierarchy to be done so that all entries in a directory are acted on before the directory itself. This can be useful when *find* is used with *cpio*(1) to transfer files that are contained in directories without write permission.

(*expression* **)** — True if the parenthesized expression is true (parentheses are special to the shell and must be escaped).

–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).

The **-type** option also recognizes **n** as a test for a network special file.

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.

FILES

/etc/group
/etc/passwd

SEE ALSO

cpio(1), sh(1), test(1), stat(2), cpio(4), fs(4).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

findmsg, dumpmsg – create message catalog file for modification

SYNOPSIS

findmsg file ...
dumpmsg file ...

DESCRIPTION

Findmsg extracts messages from C program source *file* and writes them to the standard output, along with set information. The source file lines from which the string literals are to be extracted must have *nl_msg* and " in the same line. There are four cases to be handled:

```
printf(nl_msg(1, "message"));

#define NLSMESS "message"    /* nl_msg 1 */

char nlsmess[] = "message"       /* nl_msg 1 */

char *nlsmess[] = {
        "message 1",    /* nl_msg 1 */
        "message 2",    /* nl_msg 2 */
        0
};
```

In each of the latter three cases, there are executable lines elsewhere which contain *nl_msg* in an executable form, along with the necessary reference. Only one message is allowed on each physical line. Each message must appear completely on one line along with the *nl_msg* token.

Findmsg derives message catalog set numbers from source lines which appear as:

```
#define NL_SETN 1
```

Typically a single such line will appear toward the beginning of the source file.

Dumpmsg dumps out messages which are stored in a message catalog file which was generated by the *gencat*(1) command.

The output of either command is in the form:

```
$set 1
1 message1\n
2 message two\n
```

Each message can then be changed as necessary, then processed by the *gencat*(1) command.

AUTHOR

Findmsg was developed by the Hewlett-Packard Company.

SEE ALSO

findstr(1), gencat(1), insertmsg(1), getmsg(3C).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

findstr – find strings for inclusion in message catalogs

SYNOPSIS

findstr file ...

DESCRIPTION

Findstr examines files of C source code for uncommented string constants, which it places along with the surrounding quotes on the standard output, preceding each by the file name, start position, and length. This information will be used by *insertmsg*.

SEE ALSO

insertmsg(1).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

fixman – fix manual pages for faster viewing with man(1)

SYNOPSIS

fixman

DESCRIPTION

This shell script processes all ordinary files under /usr/man/cat* to unexpand all possible spaces to tabs and remove all {character, backspace} pairs. Such pairs usually exist to cause overstriking or underscoring for printer output. They only slow down *man*(1), and use up significant amounts of disk space. The script should be run after running *catman*(1M) to rebuild all cat-able manual entries from pre-nroff forms.

The script does not remove duplicate blank lines, so all files remain a multiple of one page (66 lines) long and can still be passed directly to *lp*(1). (Note that *man*(1) normally uses *rmnl*(1) to accomplish this removal.)

To insure success, the script should be run by the super-user. It can take two to three hours to complete. As a side-effect, file ownerships and permissions may be changed.

FILES

/usr/man/cat*	Directories containing post-nroff versions of manual entries.

AUTHOR

Fixman was developed by the Hewlett-Packard Company.

SEE ALSO

catman(1M), chmod(1), expand(1), lp(1), man(1), mv(1), rmnl(1), sed(1).

NAME

fold – fold long lines for finite width output device

SYNOPSIS

fold [–width] [file ...]

DESCRIPTION

Fold is a filter which will fold the contents of the specified files, or the standard input if no files are specified, breaking the lines to have maximum width *width*. The default for *width* is 80. *Width* should be a multiple of 8 if tabs are present, or the tabs should be expanded using *expand*(1) before coming to *fold*.

SEE ALSO

expand(1)

BUGS

If underlining is present it may be messed up by folding.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

from – who is my mail from?

SYNOPSIS

from [**–s** sender] [user]

DESCRIPTION

From prints out the mail header lines in your mailbox file to show you who your mail is from. If *user* is specified, then *user*'s mailbox is examined instead of your own. If the **–s** option is given, then only headers for mail sent by *sender* are printed.

FILES

/usr/mail/*

AUTHOR

From was developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

biff(1), mail(1), prmail(1).

NAME

ftio – faster tape I/O

SYNOPSIS

ftio –o | O [cvaxEpLM] [-N datefile] [-Z nobufs] [-B blksize] [-S script] [-K comment] [-L filelist] tapedev [pathnames] [-F ignorenames]

ftio –i | I [cdmtuvfxAPEpMR] [-Z nobufs] [-B blksize] [-S script] tapedev [patterns]

ftio –g [v] tapedev [patterns]

DESCRIPTION

Ftio is a tool designed specifically for copying files to 9 track magnetic tape drives. It should perform faster than either *cpio*(1) or *tar*(1) in comparable situations. *Ftio* uses two processes (one reading/writing the file system, one writing/reading the tape device) with large amounts of shared memory between the processes, as well as a large block size for reading and writing to the tape.

The tool is compatible with *cpio* in that output from *cpio*(1) is always readable by *ftio*, and output from *ftio* is readable with exceptions by *cpio*(1). See the **CPIO COMPATIBILITY** section for a more detailed explanation.

Ftio with the **-o** (copy out) option copies files onto *tapedev* together with path name and status information. If *pathnames* was specified, *ftio* recursively descends *pathnames* looking for files, and copies those files onto *tapedev*, if not, *ftio* reads the standard input to obtain a list of path names to copy. In addition to copying the files onto the tape set, *ftio* will generate, for each tape in the tape set, a tape header containing the current tape number, machine nodename and type, operating system name, release and version numbers (all from *uname*(2)), username of the backup initiator, time and date of the backup, number of consecutive times the current media has been used, a comment field, and other items used internally by *ftio*. The tape header is separated from the main body of the archive by an end of file mark. The tape header can be read by invoking *cat*(1) with the device file name as the first argument.

When invoked using **ftio -O**, the default behavior is the same as for **ftio -ocva**. However, if the file **.ftiorc** exists in the user's home directory, *ftio* will open this file, and scan for lines preceded by **O=**. Options defined on matching lines are passed to *ftio* as if they had been passed in the original command. See the **EXAMPLES** section.

Ftio with the **-i** (copy in) option extracts files from *tapedev*, which is assumed to be the product of a previous **ftio -o** operation. Only files with names that match *patterns* are selected. *Patterns* are given in the name-generating notation of *sh*(1). In *patterns*, meta-characters **?**, *****, and **[...]** match the slash **/** character. Multiple *patterns* may be specified and if no *patterns* are specified, the default for *patterns* 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. The permissions of the files will be those of the previous **-o** operation.

When invoked using **ftio -I**, the default behavior is the same as for **ftio -icdmv**. However, if the file **.ftiorc** exists in the user's home directory, *ftio* will open this file, and scan for lines preceded by **I=**. Options defined on matching lines are passed to *ftio* as if they had been passed in the original command. See the **EXAMPLES** section.

Ftio with the **-g** option reads the file list on *tapedev*. If *patterns* is specified, only file names which match will be printed. Note that file names are always preceded by the volume that *ftio* expected the file to be on when the file list was created, and therefore only the last volume is valid in this respect.

Options

The meanings of the available options are:

a After files have been copied out to tape, reset the *access* time so that it appears the file had not been accessed by *ftio*.

d When restoring files, *directories* are to be created as needed.

c Write *header* information in ASCII character form for portability.

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. *Ftio* will use *mknod*(2) to recreate these files during a restore operation. Thus, this option is restricted to the super-user. This is intended for intrasystem (backup) use. Restoring device files onto a different system can be very dangerous.

v *Verbose*: causes a list of file names as well as tape headers to be printed. When used with the **t** option, the table of contents looks like the output of **ls -l** command (see *ls*(1)).

m Retain previous file modification time and ownership of file. Restoration of modification time is ineffective on directories that are being restored.

f Copy in all files except those that match *patterns*.

A Print all file names found on the archive, noting which files have been restored. This is useful when the user is restoring selected files, but wants to know which (if any) files are on the tape.

P On restoration, use *prealloc*(2) to pre-allocate disk space for the file, this vastly improves the localization of file fragments.

E When archiving, all files with absolute path names are archived with a path name relative to the root directory, i.e., all files whose name starts with '/' have the first '/' removed. On restoration, any files in the archive that have an absolute path name, have the leading '/' removed from the path name and are restored relative to the current directory.

p The number of blocks transferred, the total time taken (excluding tape rewind and change reel time), and the effective transfer rate, calculated from these figures, is printed at the end of the backup. If the option is given twice, this will be done at the end of each tape.

Z The next argument (*nobufs*) specifies the number of *blksize* chunks of memory to use as buffer space between the two processes, where *blksize* is the size of blocks written to the tape. More chunks is usually better, but a point is reached where no improvement is gained, and in fact performance may deteriorate as buffer space is swapped out of main memory. A default value of 16 is set, but the use of 32 or 64 may improve performance if your system is relatively unloaded. We recommend performing a backup with the system in single user mode.

B The next argument (*blksize*) specifies the size (in bytes) of blocks written to tape. This number may end with "k", specifying multiplication by 1024. Larger blocks will generally improve performance, as well as improving tape usage. The maximum block size allowed is limited by the tape drive being used. A default of 16384 bytes is set, as this is the maximum block size on most Hewlett-Packard tape drives.

S The next argument (*script*) specifies a command which is invoked every time a tape is completed in a multi-tape backup. The command is invoked by *sh*(1), with the following arguments: *script tape_no user_name*. Script is the string *script* specified by the -S option, *tape_no* is the number of the tape required, and *user_name* is the user who invoked *ftio*. Typically, the string *script* specifies a shell script which is used to notify the user that a tape change is required.

K The next argument (*comment*) specifies a comment to be placed in the *ftio* tape header.

F Arguments following **-F** specify *patterns* that should not be copied to the tape. The same rules apply for *ignorenames* as do for *patterns*, see the previous description for **ftio -i**.

N Only files that are *newer* than the file specified in the following argument (*datefile*) are copied to tape.

L If the **L** option is specified and *pathnames* have been given, *ftio* will perform the file search and generate a list of files to back up prior to actually commencing the backup.

This list is then appended to the tape header of each tape in the backup, as a list of files that *ftio* attempted to fit onto this tape. By definition, the last tape in the backup will contain a catalog of where the files are in the archive set. If *pathnames* is not specified, the file list is taken from standard input before the backup begins. The file list is also left in the current directory as the file **ftio.list**, if the **L** has been used with an argument (*filelist*), then the argument specifies the output file. In addition to generating file lists, the **L** option implements tape checkpointing, allowing the backup to restart from a write failure on a bad medium.

M Do not generate or expect tape headers, and change the default block size to 5120 bytes. This allows for full compatibility with *cpio*(1). See discussion in **CPIO COMPATIBILITY** section.

R *Ftio* will automatically resync when it gets out of phase. This is useful when restoring from a multi tape backup on tapes other than the first. The default behavior is that *ftio* will ask the user if resyncing is required.

When the end of the tape is reached, *ftio* will invoke *script* if the **-S** option has been exercised, rewind the current tape, then ask the user to mount the next tape.

If you want to pass one or more metacharacters to *ftio* without the shell expanding them, be sure to protect them by either preceding each of them with a backslash (e.g., /usr*), or enclosing in protection quotes (e.g., '/usr*').

Device files written with the **–o** option (e.g. **/dev/tty03**) will not transport to other implementations of HP-UX.

EXAMPLES

The first example below copies the entire contents of the file system (including special files) onto the tape drive **/dev/rmt/0h**:

```
ftio -ox /dev/rmt/0h /
```

The following example will restore all the files on **/dev/rmt/0h**, relative to the current directory:

```
ftio -idxE /dev/rmt/0h
```

The following example demonstrates how to list the contents of a backup set created using **ftio -o**.

```
ftio -itv /dev/rmt/0h
```

Note that use of the **-v** option will give a more detailed listing, and will display the contents of tape headers.

The next example demonstrates the use of the **.ftiorc** file. The user has a **.ftiorc** file in their home directory with the following contents:

```
# Example .ftiorc file.
I= cdmuvEpp -B 16k -S /usr/local/bin/ftio.change
O= cavEpp -Z 8 -B 16k -S /usr/local/bin/ftio.change
```

Ftio is invoked with the following command line to backup the users home directory and the system binary directory:

```
ftio -O /dev/rmt/0h /user/my_home /bin
```

Because the **-O** option has been specified, the **.ftiorc** will be checked for additional options. In this case, character headers will be generated, access times will be reset, a listing of the files copied will be printed to standard output, all file names will be copied to **/dev/rmt/0h** with path names relative to '/', performance data will be printed when the backup is complete (and at every tape change), and if the backup goes beyond one media the script "/usr/local/bin/ftio.change" will be invoked by *ftio* after each media is completed.

SIGNAL HANDLING (WARNING)

Ftio uses *System V* shared memory and semaphores for its operation. The resources committed to these functions are not automatically freed by the system when the process terminates. *Ftio* will do this only when it terminates normally, or when it terminates after receiving one the following signals: SIGHUP, SIGINT, SIGTERM. Any other signal will be handled in the default manner described by *signal*(2). Note that the behavior for SIGKILL, is to terminate the process without delay. Thus, if *ftio* receives a SIGKILL signal (as might be produced by the indiscriminate use of **kill**(1) **-9**), system resources used for shared memory and semaphores will not be returned to the system. If it becomes necessary to terminate an invocation of *ftio*, use **kill -15**. Current system usage of shared memory and semaphores can be checked using *ipcs*(1). Committed resources can be removed using ipcrm(1).

CPIO COMPATIBILITY

Ftio uses the same archive format as *cpio*(1). However, the default behavior of *ftio* is to create tape headers and to use a tape block size of 16k bytes. *Cpio*(1) by default uses 512 byte blocks, when used with the **-B** option, *cpio*(1) uses 5120 byte blocks. To achieve full compatibility with *cpio*(1) in either input or output mode, the user should specify the the **-M** option. **Ftio -oM** will create a single or multi tape archive which has no tape headers, and by default the same block size as **cpio -[o|i]B**. An archive created by a **cpio -oB** command may be restored using **ftio -iM**. If the **-M** option of *ftio* is combined with a **-B 512** block size specification, full compatibility with **cpio -[o|i]** (no **-B**) is achieved.

AUTHOR

Ftio was developed by HP.

SEE ALSO

cpio(1), find(1), cpio(4), ipcs(1), ipcrm(1), mt(7).

NAME

gencat - generate a formatted message catalog file

SYNOPSIS

gencat catfile file ...

DESCRIPTION

Gencat merges message source *files* into a formatted *catfile* which can be accessed by *getmsg*(3C). If *catfile* does not exist it will be created. If *catfile* does exist its messages will be included in the new *catfile* unless set and message numbers collide, in which case the new supersedes the old. The *files* consist of sets of messages along with comments.

The format for message source in the *files* has been designed to include compatibility with MPE and RTE. A line which begins with a dollar sign followed by a blank denotes a comment and may appear anywhere in a file.

A message set consists of a line of the form

$set*n* *[comment]*

followed by lines of the form

m message-text

where *n* denotes the set number (1-255) and *m* the message number (1-32767). Typically the set number will be used to identify the language, while the message number denotes which string from a given program is wanted. *Message-text* is a C string, including white space and '\' escapes, without the surrounding quotes. A **$set** line may optionally contain comment text following the set number. Set numbers and message numbers must be in ascending order but need not be contiguous.

If a message source line has a number but no text then the existing message with this number is deleted from the catalog.

To delete an entire message set the directive

$DELSET *set_name*

may be placed at the beginning of a line between sets.

SEE ALSO

findmsg(1), insertmsg(1), getmsg(3C).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

get – get a version of an SCCS file

SYNOPSIS

get [–rSID] [–ccutoff] [–**e**] [–**b**] [–ilist] [–xlist] [–**k**] [–l[p]] [–**p**] [–**s**] [–**m**] [–**n**] [–**g**] [–**t**] [–wstring] [–aseq-no.] file ...

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 can 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 %E% and %U% identification keywords (see below) for nested *gets* within, for example, a *send*(1) command:

~!get "–c%E% %U%" s.file

–**e** Indicates that the *get* is for the purpose of editing or making a change (delta) to the SCCS file via a subsequent use of *delta*(1). The –**e** keyletter used in a *get* for a particular version (SID) of the SCCS file prevents 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**).

–w *string* Substitute *string* for all occurrences of **@%M%** when *get*ing the file.

–a*seq-no.* The delta sequence number of the SCCS file delta (version) to be retrieved (see *sccsfile*(4)). This keyletter is used by the *comb*(1) command; it is not a generally useful keyletter, and users should not use it. If both the **–r** and **–a** keyletters are specified, the **–a** keyletter is used. Care should be taken when using the **–a** keyletter in conjunction with the **–e** keyletter, as the SID of the delta to be created may not be what one expects. The **–r** keyletter can be used with the **–a** and **–e** keyletters to control the naming of the SID of the delta to be created.

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

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

TABLE 1. Determination of SCCS Identification String

SID* Specified	-b Keyletter Used†	Other Conditions	SID Retrieved	SID of Delta to be Created
none‡	no	R defaults to mR	mR.mL	mR.(mL+1)
none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB+1).1
R	no	R > mR	mR.mL	R.1***
R	no	R = mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	–	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 should not have happened" type errors. It is *not* intended to be used on every line to provide sequence numbers.
%Z% The 4-character string @(#) recognizable by *what*(1).
%W% A shorthand notation for constructing *what*(1) strings for HP-UX system program files.
%W% = %Z%%M%<horizontal-tab>%I%
%A% Another shorthand notation for constructing *what*(1) strings for non-HP-UX system program files.
%A% = %Z%%Y% %M% %I%%Z%

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 was not applied and ignored;
* if the delta was not applied and was not ignored.
c. A code indicating a "special" reason why the delta was or was not applied:
"I": Included.
"X": Excluded.
"C": Cut off (by a **−c** keyletter).
d. Blank.
e. SCCS identification (SID).
f. Tab character.
g. Date and time (in the form YY/MM/DD HH:MM:SS) of creation.
h. Blank.
i. Login name of person who created *delta*.

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

The *p-file* is used to pass information resulting from a *get* with an **–e** keyletter along to *delta*. Its contents are also used to prevent a subsequent execution of *get* with an **–e** keyletter for the same SID until *delta* is executed or the joint edit flag, **j**, (see *admin*(1)) is set in the SCCS file. The *p-file* is created in the directory containing the SCCS file and the effective user must have write permission in that directory. Its mode is 644 and it is owned by the effective user. The format of the *p-file* is: the 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(4). *Source Code Control System User's Guide in HP-UX Concepts and Tutorials*

DIAGNOSTICS

Use *help*(1) for explanations.

WARNINGS

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

An l-file cannot be generated when **–g** is used. In other words, **–g –l** does not work.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

getopt – parse command options

SYNOPSIS

getopt optstring args

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. If it is used explicitly, *getopt* will recognize it; otherwise, *getopt* will generate it; in either case, *getopt* will place it at the end of the options. The positional parameters ($1 $2 ...) of the shell are reset so that each option is preceded by a – and is in its own positional parameter; each option argument is also parsed into its own positional parameter.

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** or **b**, as well as 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 2;;
        --)             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*.

INTERNATIONAL SUPPORT

8- and 16-bit data.

NAME

getprivgrp – get special attributes for group

SYNOPSIS

getprivgrp [**-g** | group–name]

DESCRIPTION

Getprivgrp lists the access privileges of privileged groups set by *setprivgrp*(1M). When a group name is supplied access privileges are listed for that group only. When **–g** is supplied access privileges are listed which have been granted to all groups. Otherwise, access privileges are listed for all privileged groups of which the caller is a member. The super-user is considered to be a member of all groups. Access privileges include RTPRIO, MLOCK, and CHOWN.

AUTHOR

Getprivgrp was developed by HP.

SEE ALSO

getprivgrp(2), setprivgrp(1M), privgrp(4). glossary

NAME

grep, egrep, fgrep – search a file for a pattern

SYNOPSIS

Levels B and C

grep [options] expression [files]

Level C Only

egrep [options] [expression] [files]

fgrep [options] [strings] [files]

DESCRIPTION

Commands of the *grep* family search the input *files* (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output. *Grep* patterns are limited regular *expressions* in the style of *ed*(1); it uses a compact 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.

–i Ignore upper/lower case distinction during comparisons.

–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 *expression*, because they are also meaningful to the shell. It is safest to enclose the entire *expression* argument in single quotes ´...´.

Fgrep searches for lines that contain one of the *strings*, 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 (even if matches were found).

BUGS

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

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

Egrep does not recognize ranges, such as **[a-z]**, in character classes.

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.

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

INTERNATIONAL SUPPORT

grep: 8- and 16-bit data, 8-bit filenames, messages

egrep: 8-bit data and filenames

fgrep: 8-bit data and filenames, messages.

NAME

groups – show group memberships

SYNOPSIS

groups [**[-p]** **[-g]** **[-l]** **user**]

DESCRIPTION

The *groups* command shows the groups to which you or the optionally specified user belong. If invoked with no arguments, *groups* prints the current access list returned by *getgroups*(2). Each user belongs to a group specified in the password file **/etc/passwd** and possibly to other groups as specified in the files **/etc/group** and **/etc/logingroup** . A user is granted the permissions of those groups specified in **/etc/passwd** and **/etc/logingroup** at login time. The permissions of the groups specified in **/etc/group** are normally available only with the use of *newgrp*(1). If a user name is specified with no options, *groups* prints the union of all these groups. The **-p**, **-g**, and **-l** options limit the list which is printed to only those groups specified in **/etc/passwd,** **/etc/group,** and **/etc/logingroup,** respectively.

FILES

/etc/group
/etc/logingroup
/etc/passwd

AUTHOR

Groups was developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

id(1), newgrp(1), getgroups(2), initgroups(3C), group(4).

NAME

head – give first few lines

SYNOPSIS

head [–count] [file ...]

DESCRIPTION

This filter gives the first *count* lines of each of the specified files, or of the standard input. If *count* is omitted it defaults to 10.

SEE ALSO

tail(1).

NAME

help – ask for help

SYNOPSIS

help [args]

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.
/usr/lib/help/helploc	file containing locations of help files not in **/usr/lib/help**.

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

Only SCCS and a very few other commands currently use *help*.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

hostname – set or print name of current host system

SYNOPSIS

hostname [nameofhost]

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*.

AUTHOR

Hostname was developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

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**]

DESCRIPTION

Hp supports special functions of the Hewlett-Packard 2640- and 2621- series of terminals, with the primary purpose of producing accurate representations of most *nroff*(1) 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.

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

col(1), greek(1), neqn(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.

Series 800 Only

NAME

hpiutil – ALLBASE/HP-UX HPIMAGE database utilities

SYNOPSIS

hpiutil

REMARKS

The ALLBASE/HP-UX product must be previously installed on the system for *hpiutil* to function.

DESCRIPTION

Hpiutil invokes the HPIMAGE utility program for maintaining and reconfiguring an ALLBASE/HP-UX HPIMAGE network database. No options are available with this command. *Hpiutil* can be executed by all system users on all database files created by them.

AUTHOR

Hpiutil was developed by Hewlett-Packard.

FILES

/usr/bin/hpdbdaemon	cleanup daemon program file
/usr/bin/hpimage	HPIMAGE program file
/usr/bin/hpiutil	HPIUTIL program file
/usr/lib/hpica000	HPIMAGE message catalog file

SEE ALSO

ALLBASE/HP-UX HPIMAGE Reference Manual.

NAME

hyphen – find hyphenated words

SYNOPSIS

hyphen [files]

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 and group IDs and names

SYNOPSIS

id

DESCRIPTION

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

AUTHOR

Id was developed by the Hewlett-Packard Company and AT&T Bell Laboratories.

SEE ALSO

logname(1), getuid(2).

NAME

insertmsg – use findstr(1) output to insert calls to getmsg(3C)

SYNOPSIS

insertmsg stringlist

DESCRIPTION

Insertmsg examines the file *stringlist*, which is assumed to be the output of *findstr* minus the strings which do not need to be localized and have been removed by editing. *Insertmsg* first places the line

```
#include <msgbuf.h>
```

at the beginning of each file named in *stringlist*. Then for each line in *stringlist*, it surrounds the string with an expression of the form

```
(*getmsg(nl_fd, nl_set_num, nl_msg_num, nl_msg_buf, NL_MBUFLN)
        == '\0' ? "saved string" : nl_msg_buf)
```

which evaluates to the original string if the translation cannot be retrieved. The string buffer and other "nl_" variables and constants are defined in *<msgbuf.h>*. *Insertmsg* places the modified source on a file *nl_xx.c* where the original file name was *xx.c*. The user must then hand edit the file to insert a call

```
nl_catopen("/*appropriate message catalog*/");
```

and assign the proper value to *nl_set_num*.

Insertmsg also places on the standard output a file which can be used as input to *gencat*. Again, hand editing is required to define the *£set* number to match *nl_set_num*. Messages will automatically be numbered from 1 upward in the order that they appear in *stringlist*. The same number will also be placed in the call to *getmsg*(3C), as the parameter *msg_num*.

DIAGNOSTICS

If *insertmsg* does not find the opening or closing double quote where it expects it in the strings file, it prints "insertmsg exiting : lost in strings file" and dies. If this happens, check the strings file to make sure that the lines that have been kept there have not been altered.

SEE ALSO

findstr(1), gencat(1), getmsg(3C).

BUGS

Inserts a pointer to a static area which is overwritten on each call.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

insf – install special files

SYNOPSIS

insf [**-f** devfile]
insf [**-f** devfile] **-d cn**
insf [**-f** devfile] **-d diag0**
insf [**-f** devfile] **-d disc0** [**-l** lu]
insf [**-f** devfile] **-d dmem**
insf [**-f** devfile] **-d ectest**
insf [**-f** devfile] **-d gpio0** [**-l** lu]
insf [**-f** devfile] **-d instr0** [**-l** lu]
insf [**-f** devfile] **-d ktest**
insf [**-f** devfile] **-d lpr0** [**-l** lu]
insf [**-f** devfile] **-d mm**
insf [**-f** devfile] **-d mux0** [**-l** lu]
insf [**-f** devfile] **-d mux1** [**-l** lu]
insf [**-f** devfile] **-d pty0**
insf [**-f** devfile] **-d pty1**
insf [**-f** devfile] **-d sd**
insf [**-f** devfile] **-d sw**
insf [**-f** devfile] **-d sy**
insf [**-f** devfile] **-d tape0** [**-l** lu]
insf [**-f** devfile] **-d tape1** [**-l** lu]

DESCRIPTION

Insf installs special files in the current directory. The **-f** option specifies *devfile*, which is a file that describes drivers and pseudo-drivers. This file is generated by *uxgen*(1). If the **-f** option is not given, the file **/etc/devices** is used.

If the **-d** option is not entered, *insf* installs special files for every entry in *devfile*. The **-d** option specifies that only special files for a particular driver are to be installed. If the **-l** option is given, special files for only that logical unit will be installed. The file permissions are set by *insf*. In a few cases, the owner or group id is set.

The following sections show which special files are created and their permissions for each driver:

CN

syscon rw- -w- -w-
systty rw- -w- -w-
console rw- -w- -w-

DIAG0

diag0 rw- --- ---

DMEM

dmem rw- rw- rw-

DISC0

Special file names for **disc0** use the following format: c<lu>d<unit>s<section>. For each logical unit, the following special files are installed:

dsk/c<lu>d0s<section>
    sections 0 to 11, group sys, block entry, rw- r-- ---
rdsk/c<lu>d0s<section>
    sections 0 to 11, group sys, character entry, rw- r-- ---
ct/c<lu>d<unit>s2

units 0 and 1, block entry, rw- rw- rw-
rct/c<lu>d<unit>s2
units 0 and 1, character entry, rw- rw- rw-
diag/dsk/c<lu>d<unit>
units 0 and 1, character entry, rw- --- ---

ECTEST

ectest rw- rw- rw-

GPIO0

For each logical unit, the following special file is installed:

gpio<lu> rw- rw- rw-

INSTR0

For each logical unit, the following special files are installed:

hpib/<lu>a<addr>
addrs 0 to 31, rw- rw- rw-

KTEST

ktest rw- rw- rw-

LPR0

For each logical unit, the following special file is installed:

lp<lu> owner lp, group bin, rw- --- ---

MM

The following special files are installed:

kmem minor 1, group sys, rw- r-- ---
mem minor 0, group sys, rw- r-- ---
null minor 2, rw- rw- rw-

MUX0

For each logical unit, the following special files are installed:

tty<lu>p<port>
ports 0 to 5, direct connect, rw- -w- -w-

MUX1

For each logical unit, the following special file is installed:

mux<lu> rw- --- ---

PTY0

ptyp<number> number is from 0 to 9, rw- rw- rw-

PTY1

ttyp<number> number is from 0 to 9, rw- rw- rw-

SD

hd rw- rw- rw-

SW

swap group sys, rw- r-- ---

SY

tty rw- rw- rw-

TAPE0/TAPE1

For each logical unit, the following special files are installed:

mt/<lu>l
    800 bpi, block entry, rw- rw- rw-
mt/<lu>m
    1600 bpi, block entry, rw- rw- rw-
mt/<lu>h
    6250 bpi, block entry, rw- rw- rw-
mt/<lu>ln
    no rewind, 800 bpi, block entry, rw- rw- rw-
mt/<lu>mn
    no rewind, 1600 bpi, block entry, rw- rw- rw-
mt/<lu>hn
    no rewind, 6250 bpi, block entry, rw- rw- rw-
rmt/<lu>l
    800 bpi, character entry, rw- rw- rw-
rmt/<lu>m
    1600 bpi, character entry, rw- rw- rw-
rmt/<lu>h
    6250 bpi, character entry, rw- rw- rw-
rmt/<lu>ln
    no rewind, 800 bpi, character entry, rw- rw- rw-
rmt/<lu>mn
    no rewind, 1600 bpi, character entry, rw- rw- rw-
rmt/<lu>hn
    no rewind, 6250 bpi, character entry, rw- rw- rw-
diag/mt/<lu>
    character entry, rw- --- ---

AUTHOR

Insf was developed by HP.

FILES

/etc/devices

SEE ALSO

lssf(1), mksf(1).

NAME

iostat – report I/O statistics

SYNOPSIS

iostat [-t] [interval [count]]

DESCRIPTION

Iostat iteratively reports for each disk the number of seeks per second, kilobytes transfered per second, and the milliseconds per average seek. If given a **-t** argument, it also reports the number of characters read from and written to terminals, and the percentage of time the system has spent in user mode, in user mode running low priority (niced) processes, in system mode, and idling.

To compute this information, for each disk, seeks and data transfer completions and number of words transferred are counted; for terminals collectively, the number of input and output characters are counted. Also, the state of each disk is examined HZ times per second (as found in <sys/param.h>) and a tally is made if the disk is active. From these numbers and given the transfer rates of the devices it is possible to determine average seek times for each device.

The optional *interval* argument causes *iostat* to report once each *interval* seconds. The first report is for the time since a reboot and each subsequent report is for the last interval only.

The optional *count* argument restricts the number of reports.

AUTHOR

Iostat was developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

FILES

/dev/kmem
/hp-ux

SEE ALSO

vmstat(1).

NAME

ipcrm – remove a message queue, semaphore set or shared memory id

SYNOPSIS

ipcrm [*options*]

DESCRIPTION

Ipcrm will remove one or more specified messages, semaphore or shared memory identifiers. The identifiers are specified by the following *options*:

–q *msqid* removes the message queue identifier *msqid* from the system and destroys the message queue and data structure associated with it.

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

–s *semid* removes the semaphore identifier *semid* from the system and destroys the set of semaphores and data structure associated with it.

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

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

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

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

SEE ALSO

ipcs(1), msgctl(2), msgget(2), msgop(2), semctl(2), semget(2), semop(2), shmctl(2), shmget(2), shmop(2).

NAME

ipcs – report inter-process communication facilities status

SYNOPSIS

ipcs [options]

DESCRIPTION

Ipcs prints certain information about active inter-process communication facilities. Without *options*, information is printed in short format for message queues, shared memory, and semaphores that are currently active in the system. Otherwise, the information that is displayed is controlled by the following *options*:

–q Print information about active message queues.

–m Print information about active shared memory segments.

–s Print information about active semaphores.

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

–b Print biggest allowable size information. (Maximum number of bytes in messages on queue for message queues, size of segments for shared memory, and number of semaphores in each set for semaphores.) See below for meaning of columns in a listing.

–c Print creator's login name and group name. See below.

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

–p Print process number information. (Process ID of last process to send a message and process ID of last process to receive a message on message queues and process ID of creating process and process ID of last process to attach or detach on shared memory segments) See below.

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

–a Use all print *options*. (This is a shorthand notation for **–b**, **–c**, **–o**, **–p**, and **–t**.)

–C *corefile* Use the file *corefile* in place of **/dev/kmem**.

–N *namelist* The argument will be taken as the name of an alternate *namelist* (**/hp-ux** is the default).

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

T (all)

Type of the facility:

- **q** message queue;
- **m** shared memory segment;
- **s** semaphore.

ID (all)

The identifier for the facility entry.

KEY (all)

The key used as an argument to *msgget*, *semget*, or *shmget* to create the facility entry. (Note: The key of a shared memory segment is changed to **IPC_PRIVATE** when the segment has been removed until all processes attached to the segment detach it.)

MODE (all)

The facility access modes and flags: The mode consists of 11 characters that are interpreted as follows:
The first two characters are:

- **R** if a process is waiting on a *msgrcv*;
- **S** if a process is waiting on a *msgsnd*;
- **D** if the associated shared memory segment has been removed. It will disappear when the last process attached to the segment detaches it;
- **C** if the associated shared memory segment is to be cleared when the first attach is executed;
- **–** if the corresponding special flag is not set.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to the owner's permissions; the next to permissions of others in the user-group of the facility entry; and the last to all others. Within each set, the first character indicates permission to read, the second character indicates permission to write or alter the facility entry, and the last character is currently unused.
The permissions are indicated as follows:

- **r** if read permission is granted;
- **w** if write permission is granted;
- **a** if alter permission is granted;
- **–** if the indicated permission is *not* granted.

OWNER (all)

The login name of the owner of the facility entry.

GROUP (all)

The group name of the group of the owner of the facility entry.

CREATOR (a,c)

The login name of the creator of the facility entry.

CGROUP (a,c)

The group name of the group of the creator of the facility entry.

CBYTES (a,o)

The number of bytes in messages currently outstanding on the associated message queue.

QNUM (a,o)

The number of messages currently outstanding on the associated message queue.

QBYTES (a,b)

The maximum number of bytes allowed in messages outstanding on the associated message queue.

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

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

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

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

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

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

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

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

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

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

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

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

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

HARDWARE DEPENDENCIES

Series 500

The **–C** *corefile* option and the **–N** option are not supported.

FILES

/etc/group	group names
/hp-ux	system namelist
/dev/kmem	memory
/etc/passwd	user names

SEE ALSO

msgop(2), semop(2), shmop(2).

BUGS

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

Series 800 Only

NAME

iquery – ALLBASE/HP-UX HPIMAGE database access interactive tool

SYNOPSIS

iquery

REMARKS

The ALLBASE/HP-UX product must be previously installed on the system for *iquery* to function.

DESCRIPTION

Iquery invokes the interactive programmer and database administrator tool for accessing an ALLBASE/HP-UX HPIMAGE network database. There are no options available with this command. *Iquery* can be executed by all system users.

FILES

/usr/bin/hpdbdaemon	cleanup daemon program file
/usr/bin/hpimage	HPIMAGE program file
/usr/bin/iquery	IQUERY program file
/usr/bin/iquerycf	IQUERY program file
/usr/lib/hpica000	HPIMAGE message catalog file
/usr/lib/hpiqc000	IQUERY message catalog file

AUTHOR

Iquery was developed by Hewlett-Packard.

SEE ALSO

ALLBASE/HP-UX IQUERY Reference Manual.

NAME

isl – initial system loader

DESCRIPTION

Isl implements the operating system independent portion of the bootstrap process. It is loaded and executed after self-test and initialization have completed successfully.

The processor contains special purpose memory for maintaining critical configuration related parameters (e.g. Primary Boot, Alternate Boot, and Console Paths). Two forms of memory are supported: Stable Storage and Non-Volatile Memory (NVM).

Typically, when control is transferred to *isl,* an *autoboot* sequence takes place. An *autoboot* sequence allows a complete bootstrap operation to occur with no intervention from an operator. *Isl* executes commands from the *autoexecute* file in a script-like fashion. *Autoboot* is enabled by a flag in Stable Storage.

Autosearch is a mechanism that automatically locates the boot and console devices. It is currently not implemented on the Model 840 but will be will be implemented on future Series 800 processors.

During an *autoboot* sequence, *isl* displays its revision and the name of any utility it executes. However, if *autoboot* is disabled, after *isl* displays its revision, it then prompts for input from the console device. Acceptable input is any *isl* command name or the name of any utility available on the system. If a non-fatal error occurs or the executed utility returns, *isl* again prompts for input.

Commands

There are several commands available in *isl.* The following is a list of them with a short description. Parameters may be entered on the command line following the command name. They must be separated by spaces. *Isl* prompts for any necessary parameters that are not entered on the command line.

?	
help	Help -- Lists commands and available utilities
listf	
ls	Lists available utilities
autoboot	Enables or disables the *autoboot* sequence Parameter -- on or off
autosearch	Enables or disables the *autosearch* sequence Parameter -- on or off
primpath	Modify the Primary Boot Path Parameter -- Primary Boot Path in decimal
altpath	Modify the Alternate Boot Path Parameter -- Alternate Boot Path in decimal
conspath	Modify the Console Path Parameter -- Console Path in decimal
lsautofl	
listautofl	Lists contents of the *autoexecute* file
display	Displays the Primary Boot, Alternate Boot, and Console Paths
readnvm	Displays the contents of one word of NVM in hexadecimal Parameter -- NVM address in decimal or standard hexadecimal notation
readss	Displays the contents of one word of Stable Storage in hexadecimal Parameter -- Stable Storage address in decimal or standard hexadecimal notation

DIAGNOSTICS

Isl displays diagnostic information through error messages written on the console and display codes on the hexadecimal LED display.

For the display codes, **CE0x** are informative only. **CE1x** and **CE2x** indicate errors, some of which are fatal and cause the system to halt. Other errors merely cause *isl* to display a message. During normal operation, the self-test light is yellow. However, during fatal errors, the self-test light is red.

Non-fatal errors during an *autoboot* sequence cause the *autoboot* sequence to be aborted and *isl* to prompt for input. After non-fatal errors during an interactive *isl* session, *isl* merely prompts for input.

Fatal errors cause the system to halt. The problem must be corrected and the system **RESET** to recover.

CE00	*Isl* is executing.
CE01	*Isl* is *autoboot*ing from the *autoexecute* file.
CE02	Cannot find an *autoexecute* file. *Autoboot* aborted.
CE03	No console found, *isl* can only *autoboot*.
CE05	Directory of utilities is too big, *isl* reads only 2K bytes.
CE06	*Autoexecute* file is inconsistent. *Autoboot* aborted.
CE07	Utility file header inconsistent: SOM values invalid.
CE08	*Autoexecute* file input string exceeds 2048 characters. *Autoboot* aborted.
CE09	*Isl* command or utility name exceeds 10 characters.
CE0F	*Isl* has transferred control to the utility.
CE10	Internal inconsistency: Volume label - **FATAL**.
CE11	Internal inconsistency: Directory - **FATAL**.
CE12	Error reading *autoexecute* file.
CE13	Error reading from console - **FATAL**.
CE14	Error writing to console - **FATAL**.
CE15	Not an *isl* command or utility.
CE16	Utility file header inconsistent: Invalid System ID.
CE17	Error reading utility file header.
CE18	Utility file header inconsistent: Bad magic number.
CE19	Utility would overlay *isl* in memory.
CE1A	Utility requires more memory than is configured.
CE1B	Error reading utility into memory.
CE1C	Incorrect checksum: Reading utility into memory.
CE1D	Console needed - **FATAL**.
CE1E	Internal inconsistency: Boot device class - **FATAL**.
CE21	Destination memory address of utility is invalid.
CE22	Utility file header inconsistent: *pdc_cache* entry.

CE23 Internal inconsistency: *iodc_entry_init* - **FATAL**.

CE24 Internal inconsistency: *iodc_entry_init* - console - **FATAL**.

CE25 Internal inconsistency: *iodc_entry_init* - boot device - **FATAL**.

CE26 Utility file header inconsistent: Bad aux_id.

CE27 Bad utility file type.

SEE ALSO

boot(1M), hpuxboot(1M), pdc(1M).

Series 300, 800 Only

NAME

isql – ALLBASE/HP-UX interactive SQL interface

SYNOPSIS

isql

REMARKS

The ALLBASE/HP-UX product must be previously installed on the system for *isql* to function.

DESCRIPTION

Isql invokes the Interactive SQL interface for defining and accessing an ALLBASE/HP-UX relational DataBase Environment (DBEnvironment). There are no options available with this command. *Isql* can be executed by all system users.

AUTHOR

Isql was developed by Hewlett-Packard.

FILES

/usr/bin/hpdbdaemon	cleanup daemon program file
/usr/lib/hpsqlproc	HP SQL program file
/usr/bin/isql	Interactive SQL program file
/usr/bin/sqlutil	SQLUTIL program file
/usr/lib/hpsqlcat	HP SQL message catalog file
/usr/lib/isqlwel	Interactive SQL welcome banner file

SEE ALSO

ALLBASE/HP-UX ISQL Reference Manual.

NAME

join – relational database operator

SYNOPSIS

join [options] file1 file2

DESCRIPTION

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

File1 and *file2* must be sorted in increasing ASCII collating sequence on the fields on which they are to be joined, normally the first in each line.

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

The default input field separators are blank, tab, or new-line. In this case, multiple separators count as one field separator, and leading separators are ignored. The default output field separator is a blank.

Some of the below options use the argument *n*. This argument should be a **1** or a **2** referring to either *file1* or *file2*, respectively. The following options are recognized:

–a*n* In addition to the normal output, produce a line for each unpairable line in file *n*, where *n* is 1 or 2.

–e *s* Replace empty output fields by string *s*.

–j*n m* Join on the *m*th field of file *n*. If *n* is missing, use the *m*th field in each file. Fields are numbered starting with **1**.

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

–t*c* Use character *c* as a separator (tab character). Every appearance of *c* in a line is significant. The character *c* is used as the field separator for both input and output.

EXAMPLE

The following command line will join the password file and the group file, matching on the numeric group ID, and outputting the login name, the group name and the login directory. It is assumed that the files have been sorted in ASCII collating sequence on the group ID fields.

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

SEE ALSO

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

BUGS

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

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

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

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

kermit – KERMIT-protocol file transfer program

SYNOPSIS

kermit c [lbe line baud escapechar]
kermit r [diflb line baud]
kermit s [diflb line baud] *file...*

MARKETING MODEL

HP+ File Migration Package

TECHNICAL MODEL

Importability

DESCRIPTION

Kermit is a file transfer program in common use on MS-DOS systems. It can also be used to tranfer files between two HP-UX systems when used in conjunction with *cu*(1).

Options

c	connect
r	receive files
s	send files
b	baud rate
d	debug
e	escape char
f	no filename conversion
i	image mode
l	tty line

For remote *kermit*, the format is either

kermit r

to receive files, or

kermit s *file* ...

to send files.

EXAMPLES

A typical *kermit* file transfer in conjunction with *cu*(1) follows:

```
$ cu -lculb0 -qm dir
Connected
% ls
% kermit r
~&kermit slb /dev/culb0 9600 file1 file2
Kermit: Sending file1 as FILE1
Kermit: Sending file2 as FILE2
Kermit: done.
&
% ls
file1  file2
% ~.
Disconnected
$
```

AUTHOR

Kermit is in the public domain.

SEE ALSO

umodem(1), cu(1), uucp(1).

NAME

kill – terminate a process

SYNOPSIS

kill [–signo] PID ...

DESCRIPTION

Kill sends signal 15 (terminate) to the specified processes. This will normally kill processes that do not catch or ignore the signal. The process number of each asynchronous process started with **&** is reported by the Shell (unless more than one process is started in a pipeline, in which case the number of the last process in the pipeline is reported). Process numbers can also be found by using *ps*(1).

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.

If a signal number preceded by – is given as first argument, that signal is sent instead of terminate (see *signal*(2)). In particular "kill –9 ..." is a sure kill.

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.

NAME

last, lastb – indicate last logins of users and teletypes

SYNOPSIS

last [–N] [name ...] [tty ...]
lastb [–N] [name ...] [tty ...]

DESCRIPTION

Last will look 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' would list all of "root's" sessions as well as all sessions on the console terminal. *Last* will print 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 which the session took place on. If the session is still continuing or was cut short by a reboot, *last* so indicates.

The pseudo-user **reboot** logs in at reboots of the system, thus

```
last reboot
```

will give an indication of mean time between reboot.

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.

AUTHOR

Last was developed by the University of California, Berkeley.

FILES

/etc/btmp	bad login data base
/etc/wtmp	login data base

SEE ALSO

login(1), utmp(4).

NAME

ld – link editor

SYNOPSIS

ld [[option] ... [file] ...] ...

DESCRIPTION

Ld takes one or more object files as input and combines them to produce a single (usually executable) file. In doing so it resolves references to external symbols, assigns final addresses to procedures and variables, revises code and data to reflect new addresses (a process called *relocation*), and updates symbolic debug information (when it is present in the file). By default, *ld* processes one or more object files to produce an executable file that can be run by the HP-UX loader *exec*(2). Alternatively, the linker can generate a relocatable file – one suitable for further processing by *ld* (see **–r** below). *Ld* will not generate an output file if any errors occur during its operation.

Ld recognizes two kinds of input files: object files created by the compilers or assembler (also known as '.o' files) and archives of such object files (called libraries). A library contains an index of all the externally-visible symbols from its component object files. (The archiver command *ar*(1) creates and maintains this index.) *Ld* uses this table to resolve references to external symbols.

Ld processes files in the same order as they appear on the command line. It includes code and data from a library element if, and only if, that object module provides a definition for a currently unresolved reference within the user's program. It is common practice to list libraries following the names of all simple object files on the command line.

Options

Ld recognizes the following options:

–d Forces definition of "common" storage, i.e., assign addresses and sizes, even for **–r** output.

–e *epsym* Set the default entry point address for the output file to be that of the symbol *epsym*. (This option only applies to executable files.)

–f *fill* Set the default fill pattern for "holes" within an output file as well as initialized *bss* sections. The argument *fill* is a two-byte constant.

–h *symbol* Prior to writing the symbol table to the output file, mark this name as "local" so that it is no longer externally visible. This ensures that this particular entry will not clash with a definition in another file during future processing by *ld*. (Of course, this only makes sense with the **–r** option.)

–l*x* Search a library **lib***x***.a**, where *x* is up to nine characters. A library is searched when its name is encountered, so the placement of a **–l** is significant. By default, libraries are located in **/lib** and **/usr/lib**.

–m Produce a load map on the standard output.

–n Generate an (executable) output file with code to be shared by all users. Compare with **–N**.

–o *outfile* Produce an output object file by the name *outfile*. (The default name is **a.out**.)

–q Generate an (executable) output file that is demand-loadable. Compare with **–Q**.

–r Retain relocation information in the output file for subsequent re-linking. *Ld* will not report undefined symbols.

-s Strip the output file so that it does not contain symbol table, relocation, and debug support information. This may impair or prevent the use of a symbolic debugger on the resulting program. This option is incompatible with **-r**. (The *strip*(1) command also removes this information.)

-t Print a trace (to standard output) of each input file as *ld* processes it.

-u *symname* Enter *symname* as an undefined symbol in the symbol table. The resulting unresolved reference is useful for linking a program entirely from object files in a library.

-v Display verbose messages during linking. This option may have little or no effect. It is useful for obtaining more information about an error that occurs while linking.

-x Partially strip the output file, i.e., leave out local symbols. The intention is to reduce the size of the output file without impairing the effectiveness of object file utilities. Note: use of **-x** may impact the use of a debugger.

-z Arrange for run-time dereferencing of null pointers to produce a SIGSEGV signal. (This is the complement of the **-Z** option.)

-L *dir* Change the algorithm of searching for **lib***x***.a** to look in *dir* before looking in the default places. This option is effective only if it precedes the **-l** option on the command line.

-N Generate an (executable) output file that is not shareable. This option also causes the data to be placed immediately following the text, and the text to be made writable.

-Q Generate an (executable) output file that is not demand-loadable. (This is the complement of the **-q** option.)

-R *offset* Set the origin (in hexadecimal) for the text (i.e. code) segment.

-V *num* Use *num* as a decimal version stamp identifying the **a.out** file that is produced. (This is not the same as the version information reported by the SCCS *what*(1) command.)

-X *num* Define the initial size for the linker's global symbol table. Thus you can reduce link time for very large programs, i.e., those with very many external symbols.

-Z Arrange for run-time dereferencing of null pointers to be permitted. (See in **cc**(1) the discussions of **-Z** and *pointers*.) (This is the complement of the **-z** option.)

Defaults

Unless otherwise directed, *ld* names its output **a.out**. The **-o** option overrides this. Executable output files are marked as shareable.

EXAMPLES

The following command line links part of a C program for later processing by *ld*. It also specifies a version number of 2 for the output file. (Note the '.o' suffix for the output object file. This is an HP-UX convention for indicating a linkable object file.)

```
ld -V 2 -r file1.o file2.o -o prog.o
```

The next example links a simple FORTRAN program for use with the *cdb*(1) symbolic debugger. The output file name will be **a.out** since there is no **-o** option in the command line. (Note: the particular options shown here are for a Series 200 and 300.)

```
ld -e start /lib/frt0.o ftn.o -lI77 -lF77 -lm -lc /usr/lib/end.o
```

Finally, this command will link a Pascal program on a Series 200 and 300.

```
ld -e start /lib/prt0.o main.o -lpccat -lpc -lm -lc
```

HARDWARE DEPENDENCIES

Series 200, 300

The default entry point is taken to be text location 0x0 (which is also the default origin of the program text). This corresponds to the first procedure in the first input file that the linker reads. Use the **–e** option to select a different entry point.

The version number specified with the **–V** option must be in the range 0 – 32,767.

The Series 200 and 300 linker does not support the following options: **–f**, **–m**, **–v**, **–z**, **–L**, and **–Z**.

Series 500

The linker searches for **_main** (written as **main** in C) as the main entry point for a user program. Use the **–e** option to select a different entry point.

The special names **etext** and **edata** are not supported.

The linker marks output files with the following memory management attributes by default: virtual code, virtual data (both D-data and I-data), and paged I-data. Executable output files are not shareable if they contain symbolic debug information.

The **–N** option does not cause the data to be placed immediately after the text because each is always kept in a separate segment.

The **–t** option displays file names twice, once for each pass over the input.

These options are specific to the Series 500 linker:

-A Put D-data and I-data in separate segments.

-M *maxsize* Merge code segments. The integer argument specifies a target upper bound on the size of output code segments. (The actual size may vary from this.)

-T Put D-data and I-data into the same segment.

The Series 500 linker does not support the following options: **–f**, **–m**, **–x**, **–L**, and **–R**.

Unless the user specifies a **–A** or a **–T** option, the linker puts all data in a single segment (GDS) when the total data size is less than or equal to 16,384 bytes.

Series 800

The linker searches for the symbol **$START$** as the program entry point. This symbol is defined in the file **/lib/crt0.o**, which should be the first file loaded for all programs, regardless of source language.

Nonsharable, executable files generated with the **–N** option cannot be executed via *exec*(2). Typically, **–N** is used when rebuilding the kernel.

The following options are specific to the Series 800 linker:

-y *symname* Indicate each file in which *symname* appears. Many such options may be given to trace many symbols.

-C*n* Set the maximum parameter checking level to *n*. The default maximum is 3. See the language manuals for the meanings of the parameter

checking level.

-D *offset* Set the origin (in hexadecimal) for the data space. The default value for *offset* is 0x40000000.

-G Strip all unloadable data from the output file. This option is typically used to strip debug information.

-S Generate an Initial Program Loader (IPL) auxiliary header for the output file, instead of the default HP-UX auxiliary header.

-T Save the load data in a file instead of memory during linking. This option reduces the virtual memory requirements of the linker.

The Series 800 linker does not support the following options: **–f**, **–h**, **–x**, **–V**, and **–X**.

FILES

/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	for use with *cdb/fdb/pdb*(1)
/lib/libx.a	libraries
/usr/lib/libx.a	libraries
a.out	output file

SEE ALSO

ar(1), cc(1), cdb(1), fc(1), nm(1), pc(1), strip(1), exec(2), end(3C), a.out(4), ar(4).

DIAGNOSTICS

Ld returns a zero when the link is successful. A non-zero return code indicates that an error occurred.

WARNINGS

Ld recognizes several names as having special meanings. The names **end**, **edata**, and **etext** (preceded by an underscore in some implementations) are reserved. (See *end*(3C) for details.) Users must not write alternative (externally-visible) definitions for these names.

Through its options, the link editor gives users great flexibility; however, those who invoke the linker directly must assume some added responsibilities. Input options should ensure the following properties for programs:

- When the link editor is called through *cc*(1), a start-up routine is linked with the user's program. This routine calls *exit*(2) after execution of the main program. If users call *ld* directly, they must ensure that the program always calls *exit*() rather than falling through the end of the entry routine.
- When linking for use with the symbolic debugger *cdb*, the user must ensure that the program contains a routine called *main*. Also, the user must link in the file **/usr/lib/end.o** as the last file named on the command line.

There is no guarantee that the linker will pick up files from libraries and include them in the final program in the same relative order that they occur within the library.

NAME

leave – remind you when you have to leave

SYNOPSIS

leave [hhmm]

DESCRIPTION

Leave waits until the specified time, then reminds you that you have to leave. You are reminded 5 minutes and 1 minute before the actual time, at the time, and every minute thereafter. When you log off, *leave* exits just before it would have printed the next message.

The time of day is in the form hhmm where hh is a time in hours (on a 12 or 24 hour clock). All times are converted to a 12 hour clock, and assumed to be in the next 12 hours.

If no argument is given, *leave* prompts with "When do you have to leave?". A reply of newline causes *leave* to exit, otherwise the reply is assumed to be a time. This form is suitable for inclusion in a *.login* or *.profile*.

Leave ignores interrupts, quits, and terminates. To get rid of it you should either log off or use "kill –9" giving its process id.

AUTHOR

Leave was developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

calendar(1).

NAME

lex – generate programs for lexical analysis of text

SYNOPSIS

lex [**–rctvn**] [**–X<secondary><n>** ...] [file] ...

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*(1). 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.

The **–X<***secondary***><***n***>** option allows the sizes of certain internal *lex* tables to be reset. *Secondary* is one of the letters from the set { **d D s S a c** } and specifies the table; *n* is the new size. Tables whose size can be changed by using secondary letters are:

d	table of definitions; default = 200.
D	table of characters in definition strings; default = 5000.
s	table of start conditions; default = 50.
S	table of characters in start condition names; default = 500.
c	array table for storing character classes; default = 1000.
a	right context/action array table; default = 100.

If an array overflows, *lex* issues a fatal error message including a suggestion of which table to reset. For example:

Definitions too long, try –XD option

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 2500);
%n *n*	number of states is *n* (default is 500);
%e *n*	number of parse tree nodes is *n* (default is 1000);
%a *n*	number of transitions is *n* (default is 2000).
%k *n*	number of packed character classes is *n* (default is 1000);
%o *n*	size of output array 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), malloc(3X).
LEX – Lexical Analyzer Generator, in *HP-UX: Selected Articles.*

BUGS

The **–r** option is not yet fully operational.

The token buffer in the program built by *lex* is of fixed length,

yytext[YYLMAX]

where YYLMAX is defined to be 200 characters. Overflow of this array is not detected in the *yylex.c* program.

INTERNATIONAL SUPPORT

8-bit data and filenames.

NAME

lifcp – copy to or from LIF files

SYNOPSIS

lifcp [**-T**xxx] [**-L**xxx] [**-v**xxx] [**-b**] [**-i**xxx] [**-r**] [**-t**] file1 file2
lifcp [**-T**xxx] [**-L**xxx] [**-v**xxx] [**-b**] [**-i**xxx] [**-r**] [**-t**] file1 [file2 ...] directory

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.

Options may appear singly or be combined in any order before the file names. The space between option and argument is optional.

-Txxx — Used only when copying files to a LIF volume. This option will force the file type of the LIF directory entry to be set to the argument given, which may be decimal, octal or hex in standard "C" notation.

-Lxxx — Used only when copying files to a LIF volume. This option will set the "last volume flag" to xxx (0 or 1). The default "last volume flag" is one.

-vxxx — Used only when copying files to a LIF volume. This option will set the "volume number" to xxx. The default "volume number" is one.

-b — This option will force a BINARY mode of copying regardless of the file type. When copying in BINARY mode from HP-UX to LIF the default file type is BINARY(-2). (For details on available modes of copying refer to *lif*(4)). This option is a no-op when copying from LIF to LIF.

-ixxx — Used only when copying files to a LIF volume. This option sets the "implementation" field of the LIF directory entry to the argument given, which may be decimal, octal or hex in standard "C" notation. The "implementation" field can only be set for file types -2001 to -100000 (octal). The "implementation" field is set to zero for all interchange file types and for file types -2 to -200 (octal).

-r — This option will force a RAW mode of copying regardless of the file type. When copying in RAW mode from HP-UX to LIF the default file type is BIN(-23951). **-T** option will override the default file type. (various modes of copying are explained in *lif*(4).) This option is a no-op in case of LIF to LIF copying.

-t — will cause the HP-UX file names to be translated to a name acceptable by a LIF utility. That is, all the lower-case letters will be up-shifted and all other characters except numeric will be changed to an underscore (_). If the HP-UX file name starts with a non-letter, the file name will be preceded by the capital letter (X). Note that if there are two files named colon (:) and semicolon (;), both of them will be translated to X_. File names will be truncated to a maximum of 10 characters. When copying a LIF file to (HP-UX/LIF) file **-t** is a no-op. Omitting **-t** will cause error to be generated if an improper name is used.

The default copying modes when copying from LIF to HP-UX are summarized in the following table:

file type	default copying mode
ASCII	ASCII
BINARY	BINARY
BIN	RAW
other	RAW

When copying from HP-UX to LIF, the default copying mode is ASCII and an ASCII file is created.

When copying from LIF to LIF, if no options are specified then all the LIF directory fields and content of the file are duplicated from source to destination.

A LIF file name is recognized by the embedded colon (:) delimiter (see *lif*(4) 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) will be interpreted to mean standard input or standard output, depending on its position in the argument list. This is particularly useful if the data requires non-standard translation. When copying from standard input, if no other name can be found, the name "STDIN" is used.

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 while using *lifcp*.

HARDWARE DEPENDENCIES

Series 500:

You **must** use a character special file to access the media.

Series 800:

The following option is also supported:

–Knnn forces each file copied in to begin on a nnn * 1024 byte boundary from the beginning of the volume. This is useful when files are used for Series 800 boot media. This option has no effect when copying from a LIF volume.

EXAMPLES

lifcp abc lifvol:CDE

copy HP-UX file abc to LIF file CDE on LIF volume lifvol which is actually an HP-UX file initialized to be a LIF volume.

lifcp -t * ../lifvol:

will copy all the HP-UX files in the current directory to the LIF volume lifvol which is present in the parent directory. File names are translated to appropriate LIF file names.

lifcp -r -T -5555 -t *.o lifvol:

will copy all the HP-UX object files in the current directory to the LIF volume lifvol. Copying mode is RAW and LIF file types are set to -5555.

lifcp -b *.o lifvol:

All the object files in the current directory are copied to the LIF volume lifvol. Copying mode is BINARY and LIF BINARY files are created.

lifcp -r -t * /lifvol:

All the files in the current directory are copied to the LIF volume lifvol in root directory. Copying mode is RAW and LIF file types are set to BIN.

lifcp abc\\: lifvol:CDE

copy file abc: to LIF file CDE in lifvol.

lifcp -t abc def lifvol:

copy files abc and def to lif files ABC and DEF within lifvol.

lifcp lifvol:ABC .

copy LIF file ABC within lifvol to file ABC within current directory.

lifcp - /dev/dsk/1s2:A_FILE
copy standard input to LIF file A_FILE on LIF volume /dev/dsk/1s2.

lifcp lifvol:ABC /dev/dsk/1s2:CDE
copy LIF file ABC in lifvol to LIF file CDE on /dev/dsk/1s2.

pr abc | lifcp - lifvol:ABC
copy the output of pr to the LIF file ABC.

pr abc | lifcp - lifvol:
copy the output of pr to the LIF volume lifvol. LIF file STDIN is crated since no files names are specified.

lifcp lifvol:ABC -
copy LIF file ABC in lifvol to standard out.

lifcp * ../lifvol:
copy all files within current directory to LIF files of the same name on LIF volume lifvol (may cause errors if file names in the current directory do not obey LIF naming conventions!).

AUTHOR

Lifcp was developed by the Hewlett-Packard Company.

SEE ALSO

lifinit(1), lifls(1), lifrename(1), lifrm(1), lif(4).

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 [–vnnn] [–dnnn] [–n string] file

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 256, 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 not specified, the volume name is set to the last component of the path name specified by *file*. A legal LIF volume name is 6 characters long and is limited to upper case letters (A-Z), digits (0-9) and the underscore character (_). The first character (if any) must be a letter. The utility will automatically perform translation to create legal LIF volume names. Therefore, all lower-case letters are up-shifted and all other characters except numeric and underscore will be replaced with capital letter (X). If the volume name does not start with a letter, the volume name will be preceded by the capital letter (X). The volume name will also be right padded with blanks or truncated as needed to be 6 characters long. If **-n** is used with no *string*, the default volume name is set to 6 blanks.

If *file* does not exist, a regular HP-UX disk file is created and initialized.

The default values for volume size are 256K bytes for regular files, and the actual capacity of the device 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, Series 300

If your media has never been initialized, it must be initialized using *mediainit*(1) before *lifinit* can be used. (Refer to the System Administrator Manual for details concerning *mediainit*.)

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*(1M) before *lifinit* can be used.

Series 800:

The following options are also supported:

-s*nnn*	set the initial system load **(ISL)** start address to nnn in the volume label. This is useful when building boot media for Series 800 systems.
-l*nnn*	specifies the length in bytes of the **ISL** code in the **LIF** volume.
-e*nnn*	set the **ISL** blocksize to nnn bytes.
-K*nnn*	forces the directory start location to be the nearest multiple of nnn * 1024 bytes from the beginning of the volume. This is necessary for booting Series 800 systems off of **LIF** media.

EXAMPLES

lifinit -v500000 -d10 x
lifinit /dev/rdsk/1s2

AUTHOR

Lifinit was developed by the Hewlett-Packard Company.

SEE ALSO

lifcp(1), lifls(1), lifrename(1), lifrm(1), sdfinit(1M), lif(4).

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 a LIF directory

SYNOPSIS

lifls [option] name

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. The following options are available and only one option should be specified at any one time:

-l List in long format, giving volume name, volume size, directory start, directory size, file type, file size, file start, "implementation" field (in hex), date created, last volume and volume number.

-C Force multiple column output format regardless of STDOUT type.

-L Will return the content of the "last volume flag" in decimal.

-i Will return the content of the "implementation" field in hex.

-v Will return the content of the "volume number" in decimal.

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 -C /dev/rdsk/1s2

AUTHOR

Lifls was developed by the Hewlett-Packard Company.

SEE ALSO

lifcp(1), lifinit(1), lifrename(1), lifrm(1), lif(4).

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

DESCRIPTION

Oldfile is a full LIF file specifier (see *lif*(4) for details) for the file to be renamed (e.g. **liffile:A_FILE**). *Newfile* is 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/dsk/1s2:ABC CDE

AUTHOR

Lifrename was developed by the Hewlett-Packard Company.

SEE ALSO

lifcp(1), lifinit(1), lifls(1), lifrm(1), lif(4).

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

DESCRIPTION

Lifrm removes one or more entries from a LIF volume. File name specifiers are as described in *lif*(4).

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/rdsk/1s2.0:F

AUTHOR

Lifrm was developed by the Hewlett-Packard Company.

SEE ALSO

lifcp(1), lifinit(1), lifls(1), lifrename(1), lif(4).

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

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

sh(1), read(2).

INTERNATIONAL SUPPORT

8- and 16-bit data.

Series 500 Only

NAME

linkinfo – object file link information utility

SYNOPSIS

linkinfo [[option] ... [file] ...] ...

Remarks:

Linkinfo is implemented on the Series 500 only.

DESCRIPTION

Linkinfo examines the object files that are part of a program and prints statistics about sizes of the various data areas and symbol table information. *Linkinfo* searches libraries and examines object files in the same fashion as the link editor *ld*. Thus your command line should reflect the same ordering of object files and libraries as it does for the corresponding link.

Linkinfo is intended for developers of large FORTRAN applications who want information about data sizes in order to tune their application for the Series 500 architecture. It prints a file-by-file summary of sizes for code segments and for the D-data and I-data areas (both initialized and uninitialized). There are options for including information about COMMON blocks, linker-generated pointers, and linker symbol entries (again, file-by-file). There is also provision for generating a crude cross-reference of COMMON block usage by file.

Linkinfo options may occur anywhere on the command line after the command name itself. Some options take a modifier immediately following the option letter (e.g. ... **-e** *entryname*). The space between the option and the modifier is optional.

This utility recognizes the following options. Note that a colon indicates that the option takes an argument; the colon itself is **not** a literal, and must not appear when specifying arguments.

-c requests the name and size of COMMON blocks in the input files.

-e: names an alternate entry point for the user program, other than **_main**. The loader calls this alternate entry point at run-time.

-l: abbreviates a library name. *Linkinfo* searches a default set of directories to locate the desired library. These directories are */lib* and */usr/lib*.

The utility searches these directories in the above order, looking for the library **lib***xxx***.a**, where *xxx* is a string of one or more ASCII characters specified as the modifier for the **-l** option. Since *linkinfo* searches a library immediately upon encountering the library's name on the command line, the placement of the **-l** option is significant. A **-l** with no modifier is the same as **-lc**, which causes *linkinfo* to search the standard C library.

-p requests size information on linker-generated pointers.

-s forces inclusion of symbol table size information for each input file.

-u: specifies a name to enter in the symbol table as undefined. This entry appears as an unresolved reference to the command name. You can use it to force loading object information solely from a library.

-x produces a cross-referenced listing of COMMON block usage. This information is saved in the file *xref.out*.

SEE ALSO

ld(1), getopt(1).

DIAGNOSTICS

Linkinfo returns the following exit codes:

0 – no errors
1 – abort (killed by signal)
2 – error during link

NAME

lint – a C program checker/verifier

SYNOPSIS

lint [options] file ...

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 that are currently detected are unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, the usage of functions is checked to find functions that return values in some places and not in others, functions called with varying numbers or types of arguments, and functions whose values are not used or whose values are used but none returned.

Arguments whose names end with **.c** are taken to be C source files. Arguments whose names end with **.ln** are taken to be the result of an earlier invocation of *lint* with either the **–c** or the **–o** option used. The **.ln** files are analogous to **.o** (object) files that are produced by the *cc*(1) command when given a **.c** file as input. Files with other suffixes are warned about and ignored.

Lint will take all the **.c**, **.ln**, and **llib-l***x***.ln** files (specified by –l*x* and process them in their command line order. By default, *lint* appends the standard C lint library (**llib-lc.ln**) to the end of the list of files. However, if the **–p** option is used, the portable C lint library (**llib-port.ln**) is appended instead. When the **–c** option is not used, the second pass of *lint* checks this list of files for mutual compatibility. When the **–c** option is used, the **.ln** and the **llib-l***x***.ln** files are ignored.

Any number of *lint* options may be used, in any order, intermixed with file name arguments. 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 many such complaints).

–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:

–l*x* Include additional lint library **llib-l***x***.ln**. For example, you can include a lint version of the Math Library **llib-lm.ln** by inserting **–lm** on the command line. This argument does not suppress the default use of **llib-lc.ln**. These lint libraries must be in the assumed directory. This option can be used to reference local lint libraries and is useful in the development of multi-file projects.

–n Do not check compatibility against either the standard or the portable lint library.

–p Attempt to check portability to other dialects of C. Along with stricter checking, this option causes all non-external names to be truncated to eight characters and all external names to be truncated to six characters and one case.

–c Cause *lint* to produce a **.ln** file for every **.c** file on the command line. These **.ln** files are the product of *lint*'s first pass only, and are not checked for inter-function compatibility.

–o lib Cause *lint* to create a lint library with the name **llib-l***lib***.ln**. The **–c** option nullifies any use of the **–o** option. The lint library produced is the input that is given to *lint*'s second pass. The **–o** option simply causes this file to be saved in the named lint library. To produce a **llib-l***lib***.ln** without extraneous messages, use of the **–x** option is suggested. The **–v** option is useful if the source file(s) for the lint library are just external interfaces (for example, the way the file **llib-lc** is written). These option settings are also available through the use of "lint comments" (see below).

The **–D**, **–U**, and **–I** options of *cpp*(1) and the **–g**, and **–O**, options of *cc*(1) are also recognized as separate arguments. The **–g** and **–O** options are ignored, but, by recognizing these options, *lint*'s behavior is closer to that of the *cc*(1) command. Other options are warned about and ignored. The pre-processor symbol "lint" is defined to allow certain questionable code to be altered or removed for *lint*. Therefore, the symbol "lint" should be thought of as a reserved word for all code that is planned to be checked by *lint*.

Certain conventional comments in the C source will change the behavior of *lint*:

/*NOTREACHED*/
at appropriate points stops comments about unreachable code. (This comment is typically placed just after calls to functions like *exit*(2)).

/*VARARGS*n**/
suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0.

/*ARGSUSED*/
turns on the **–v** option for the next function.

/*LINTLIBRARY*/
at the beginning of a file shuts off complaints about unused functions and function arguments in this file. This is equivalent to using the **–v** and **–x** options.

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, if the **–c** option is not used, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a 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.

The behavior of the **–c** and the **–o** options allows for incremental use of *lint* on a set of C source files. Generally, one invokes *lint* once for each source file with the **–c** option. Each of these invocations produces a **.ln** file which corresponds to the **.c** file, and prints all messages that are about just that source file. After all the source files have been separately run through *lint*, it is invoked once more (without the **–c** option), listing all the **.ln** files with the needed **–l***x* options. This will print all the inter-file inconsistencies. This scheme works well with *make*(1); it allows *make* to be used to *lint* only the source files that have been modified since the last time the set of source files were *lint*ed.

HARDWARE DEPENDENCIES

Series 200, Series 300, Series 500

Lint utilizes a special version of the C compiler front end. The size of the internal compiler tables can be adjusted by using the -N option. The syntax for this option is described in the HARDWARE DEPENDENCIES section of the manual page for *cc*(1).

The following option is supported:

–Y Enable support of 16-bit characters inside string literals and comments. Note that 8-bit parsing is always supported. See *hpnls*(5) for more details on International Support.

FILES

/usr/lib	the directory where the lint libraries specified by the –l*x* option must exist
/usr/lib/lint[12]	first and second passes
/usr/lib/llib–lc.ln	declarations for C Library 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/lib/llib-lm.ln	declarations for Math Library functions (binary format; source is in **/usr/lib/llib-lm**)
/usr/tmp/*lint*	temporaries

SEE ALSO

cc(1), cpp(1), make(1).

WARNINGS

Exit(2), *longjmp* (on *setjmp*(3C)), and other functions that do not return are not understood; this causes various inaccuracies.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

lock – reserve a terminal

SYNOPSIS

lock

DESCRIPTION

Lock requests a password from the user, then prints "This terminal is locked -- do not disturb" on the terminal and refuses to relinquish the terminal until the password is repeated. If the user forgets the password, he has no other recourse but to login elsewhere and kill the lock process.

BUGS

Should timeout after 15 minutes.

NAME

login – sign on

SYNOPSIS

login [name [env-var ...]]
login - -r rhost

DESCRIPTION

The *login* command is used at the beginning of each terminal session and allows you to identify yourself to the system. It may be invoked as a command or by the system when a connection is first established. Also, it is invoked by the system when a previous user has terminated the initial shell by typing a *cntrl-d* to indicate an "end-of-file."

If *login* is invoked as a command it must replace the initial command interpreter. This is accomplished by typing:

exec login

from the initial shell.

Login asks for your user name (if not supplied as an argument), and, if appropriate, your password. Echoing is turned off (where possible) during the typing of your password, so it will not appear on the written record of the session. An invalid login name will cause a request for a password. This is done to make it more difficult for an unauthorized user to log in on the system by trial and error. After three unsuccessful login attempts, a *hangup* signal is issued.

The **-r** option is only useful on those installations which support the Berkeley remote login service *rlogin*. This option is used by the *rlogin* server to inform *login* that a remote login is being attempted from the given remote hostname *rhost*. *Login* then reads the remote user's name *remuser*, the local user's name *locuser*, and the user's remote terminal type. *Login* then uses the following three conditions to decide if the user can be logged in without asking for a password:

The remote host *rhost* appears in the file **/etc/hosts.equiv** and *remuser* = *locuser*.

The file **$HOME/.rhosts** contains a line listing just *rhost* and *remuser* = *locuser*, where **$HOME** is locuser's login directory.

The file **$HOME/.rhosts** contains a line listing the remote host *rhost* followed by the remote user *remuser*, seperated by exactly one space.

If none of these conditions are met, then a password is prompted for as if *locuser* had been specified as the user name on the command line. Once the user is logged in, *login* proceeds as in a normal login.

For security reasons, the following conditions also apply to the **-r** option:

Login must be running as the super-user (uid = 0).

If attempting to login as the super-user (uid = 0), then the file **/etc/hosts.equiv** is not checked, though the file **$HOME/.rhosts** is still searched.

The file **$HOME/.rhosts** must be owned either by *locuser* or by the super-user.

The file **$HOME/.rhosts** must not be a symbolic link on those installations which support them.

At some installations, an option may be invoked that will require you to enter a second "dialup" password. This will occur only for dial-up connections, and will be prompted by the message "dialup password:". Both passwords are required for a successful login. See *dialups*(4) for details on dialup security.

If password aging has been invoked by the super-user on your behalf, your password may have expired. In this case, you will be 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 will be silently disconnected.

After a successful login, the accounting files are updated, the command interpreter (usually *sh*(1)) is determined, and the user and group id's, group access list, and working directory are initialized. These specifications are found in the **/etc/passwd** and **/etc/logingroup** file entries for the user. The name of the command interpreter as passed to it is – followed by the last component of the interpreter's pathname (i.e., **–sh**). If this field in the password file is empty, then the default command interpreter, **/bin/sh** is used. The command interpreter performs its own initialization, and does login initialization if the name by which it is called starts with –.

If *sh*(1) is the command interpreter, it executes the profile files **/etc/profile** and **$HOME/.profile** 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.

If the command name field is "*", then a *chroot*(2) is done to the directory named in the directory field of the entry. At that point *login* is re-executed at the new level which must have its own root structure, including **/etc/login** and **/etc/passwd**.

The basic *environment* (see *environ*(5)) is initialized to:

HOME=*your-login-directory*
PATH=:/bin:/usr/bin
SHELL=*last-field-of-passwd-entry*
MAIL=/usr/mail/*your-login-name*
TZ=*timezone-specification*

For the super-user, PATH is augmented to include **/etc**. In the case of a remote login, the enviroment variable TERM is also set to the remote user's terminal type.

The environment may be expanded or modified by supplying additional arguments to *login*, either at execution time or when *login* requests your login name. The arguments may take either the form *xxx* or *xxx=yyy*. Arguments without an equal sign are placed in the environment as

L*n*=xxx

where *n* is a number starting at 0 and is incremented each time a new variable name is required. Variables containing an = are placed into the environment without modification. If they already appear in the environment, then they replace the older value. There are two exceptions. The variables **PATH** and **SHELL** cannot be changed. This prevents people, logging into restricted shell environments, from spawning secondary shells which are not restricted. Both *login* and *getty* understand simple single-character quoting conventions. Typing a backslash in front of a character quotes it and allows the inclusion of such things as spaces and tabs.

If **/etc/btmp** is present, all unsuccessful 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.**

FILES

$HOME/.profile	personal profile (individual user initialization)
$HOME/.rhosts	personal equivalence file for the remote login server
/etc/btmp	history of bad login attempts
/etc/d_passwd	dialup security encrypted passwords

/etc/dialups	lines which require dialup security
/etc/hosts.equiv	system list of equivalent hosts allowing logins without passwords
/etc/logingroup	group file – defines group access lists
/etc/motd	message-of-the-day
/etc/passwd	password file – defines users, passwords, and primary groups
/etc/profile	system profile (initialization for all users)
/etc/securetty	list of valid ttys for root login
/etc/utmp	users currently logged in
/etc/wtmp	history of logins, logouts, and date changes
/usr/mail/*your-name*	mailbox for user *your-name*

VARIABLES

HOME	The users home directory.
PATH	The path to be searched for commands.
SHELL	Which command interpreter is being used.
MAIL	Where to look for mail.
TERM	The user's terminal type.
TZ	The current timezone.
xxx	User specified named variables.
L *xxx*	User specified unnamed variables.

SEE ALSO

last(1), mail(1), newgrp(1), passwd(1), sh(1), su(1), getty(1M), initgroups(3C), dialups(4), group(4), passwd(4), profile(4), utmp(4), environ(5).

DIAGNOSTICS

The following diagnostics will appear if problems occur:

Login incorrect:
if the user name or the password cannot be matched.

No shell, cannot open password file, or **no directory**:
consult your system manager.

Your password has expired. Choose a new one:
if password aging is implemented.

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 /bin/login or /etc/login on root:
same as above except sub-root login command not found.

Bad user id. or **Bad group id.**:
setuid or *setgid* failed.

Unable to change to directory <name>:
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.

Sorry, single-user:
occurs if the version field from *uname*(2) starts with **A** (or if the *uname* system call fails) and if your terminal name is not /dev/console and if your home shell is not named /usr/lib/uucp/uucico. You are not logged in.

No utmp entry. You must exec "login" from the lowest level "sh":
if you attempted to execute *login* as a command without using the shell's *exec* internal command or from other than the initial shell.

.rhosts is a soft link:
if your personal equivalence file is a symbolic link.

Bad .rhosts ownership:
if your personal equivalence file is not owned by the local user or by the super-user.

Remuser too long, locuser too long, or **terminal type too long**:
if the indicated string was too long for *login's* internal buffer.

AUTHOR

Login was developed by AT&T and HP.

NAME

logname – get login name

SYNOPSIS

logname

DESCRIPTION

Logname returns the contents of the environment variable **$LOGNAME**, which is set when a user logs into the system.

FILES

/etc/profile

SEE ALSO

env(1), login(1), logname(3X), environ(5).

NAME

lorder – find ordering relation for an object library

SYNOPSIS

lorder file ...

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). Note that the link editor *ld*(1) is capable of multiple passes over an archive in the archive format and does not require that *lorder*(1) be used when building an archive. The usage of the *lorder*(1) command may, however, allow for a slightly more efficient access of the archive during the link edit process.

The following example builds a new library from existing **.o** files.

```
ar cr library `lorder *.o | tsort`
```

FILES

*symref, *symdef temporary files

SEE ALSO

ar(1), ld(1), tsort(1).

BUGS

Object files whose names do not end with **.o**, even when contained in library archives, are overlooked. Their global symbols and references are attributed to some other file.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

lp, cancel – send/cancel requests to an LP line printer

SYNOPSIS

lp [**–c**] [**–d***dest*] [**–m**] [**–n***number*] [**–o***option*] [**–s**] [**–t***title*] [**–w**] files
cancel [ids] [printers]

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 be linked into a spool directory. Ownership and mode of the linked *files* remains unchanged. If the **–c** option is given or linking is not possible then *files* are copied, in which case the ownership and mode are set to allow read access to owner *lp* and group *bin* only. It should be noted that if the *files* are linked rather than copied, 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*(1M) and *lpstat*(1)). By default, *dest* is taken from the environment variable **LPDEST** (if it is set). Otherwise, a default destination (if one exists) for the computer system is used. Destination names vary between systems (see *lpstat*(1)).

–m 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 of 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 as *options* for printers supported on your hardware, see the *mklp*(1M) script.

–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.

EXAMPLES

Assuming there is an existing Hewlett-Packard 2934A line printer named *lp2*, configured with the **hp2934a** model interface program. This model has the **-c** option which will cause the printer to print in a compressed mode. To obtain compressed print on *lp2*, use the command:

lp -dlp2 -oc files

FILES

/usr/spool/lp/*

HARDWARE DEPENDENCIES

Series 200, 300, 800

See also *slp*(1).

SEE ALSO

accept(1M), enable(1), lpadmin(1M), lpsched(1M), lpstat(1), mail(1), mklp(1M).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

lpstat – print LP status information

SYNOPSIS

lpstat [options]

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).

INTERNATIONAL SUPPORT

messages.

NAME

ls, l, ll, lsf, lsr, lsx – list contents of directories

SYNOPSIS

ls [**–RaAdCxmlnog1rtucpFbqisf**] [names]
l [ls options] [names]
ll [ls options] [names]
lsf [ls options] [names]
lsr [ls options] [names]
lsx [ls options] [names]

DESCRIPTION

For each directory argument, *ls* lists the contents of the directory; for each file argument, *ls* repeats its name and any other information requested. The output is sorted alphabetically by default. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but file arguments appear before directories and their contents.

If you are the super-user, all files except . and .. are listed by default.

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, the **–C** and **–x** options enable multi-column formats, and the **–m** option enables stream output format in which files are listed across the page, separated by commas. In order to determine output formats for the **–C**, **–x**, and **–m** options, *ls* uses an environment variable, **COLUMNS**, to determine the number of character positions available on one output line. If this variable is not set, the *terminfo* database is used to determine the number of columns, based on the environment variable **TERM**. If this information cannot be obtained, 80 columns is assumed.

Options

There are numerous options:

–R Recursively list subdirectories encountered.

–a List all entries; usually 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**.

–d If an argument is a directory, list only its name (not its contents); often used with **–l** to get the status of a directory.

–C Multi-column output with entries sorted down the columns.

–x Multi-column output with entries sorted across rather than down the page.

–m Stream output format.

–l List in long format, giving mode, number of links, owner, group, size in bytes, and time of last modification for each file (see below). If the file is a special file, the size field will instead contain the major and minor device numbers rather than a size.

–n The same as **–l**, except that the owner's **UID** and group's **GID** numbers are printed, rather than the associated character strings.

–o The same as **–l**, except that only the owner is printed (group is omitted.) (If both **-l** and **-o** are specified, the group is not printed.)

-g The same as **-l**, except that only the group is printed (owner is omitted.) (If both **-l** and **-g** are specified, the owner is not printed.)

-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.

-r Reverse the order of sort to get reverse alphabetic or oldest first as appropriate.

-t Sort by time modified (latest first) instead of by name.

-u Use time of last access instead of last modification for sorting (with the **-t** option) or printing (with the **-l** option).

-c Use time of last modification of the inode (file created, mode changed, etc.) for sorting (**-t**) or printing (**-l**).

-p Put a slash (/) after each file name if that file is a directory.

-F Put a slash (/) after each file name if that file is a directory, and put an asterisk (*) after each file name if that file is executable.

-b Force printing of non-graphic characters to be in the octal \ddd notation.

-q Force printing of non-graphic characters in file names as the character (?).

-i For each file, print the i-number in the first column of the report.

-s Give size in blocks, including indirect blocks, for each entry.

-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.

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 10 characters that are interpreted as follows:

The first character is:

d if the entry is a directory;
b if the entry is a block special file;
c if the entry is a character special file;
p if the entry is a fifo (a.k.a. "named pipe") special file;
n if the entry is a network special file;
- if the entry is an ordinary file.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to the owner's permissions, the next to permissions of others in the user-group of the file, and the last to all others. Within each set, the three characters indicate permission to read, to write, and to execute the file as a program, respectively. For a directory, "execute" permission is interpreted to mean permission to search the directory for a specified file.

The permissions are indicated as follows:

r if the file is readable;
w if the file is writable;
x if the file is executable;
– if the indicated permission is *not* granted.

The group-execute permission character is given as **s** if the file has set-group-ID mode; likewise, the user-execute permission character is given as **s** if the file has set-user-ID mode. The last character of the mode (normally **x** or –) is **t** if the 1000 (octal) bit of the mode is on; see *chmod*(1) for the meaning of this mode. The indications of set-ID and 1000 bits of the mode are capitalized (**S** and **T** respectively), if the corresponding execute permission is not set.

When the sizes of the files in a directory are listed, a total count of blocks, including indirect blocks, is printed.

HARDWARE DEPENDENCIES

Integral PC:

The file type **n** is not supported.

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**.
/usr/lib/terminfo/?/*	to get terminal information.

AUTHOR

Ls was developed by AT&T and the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

chmod(1), find(1).

BUGS

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* almost inevitably fail.

Unprintable characters in file names may confuse the columnar output options.

INTERNATIONAL SUPPORT

ls: 8-bit filenames, messages.

NAME

lsdev – list device drivers in the system

SYNOPSIS

/etc/lsdev [major...]

Remarks:

This manual page describes *Lsdev* as implemented on the Series 200.

Not supported on the Integral Personal Computer.

DESCRIPTION

With no arguments, *lsdev* lists the major device numbers, for block and character files, and driver names of all device drivers configured into the system and available for invocation via special files. A "-1" in either the block or character column means that a major number does not exist for that type.

If there are any arguments, they must represent major device numbers. The corresponding driver name, if any, will be printed for each argument. Some numbers will return two driver names, one for the block and one for the character.

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

lsdev – list device drivers in the system

SYNOPSIS

/etc/lsdev [major...]

Remarks:

This manual page describes *lsdev* as implemented on the Series 500.
not supported on the Integral Personal Computer.

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.

Series 800 Only

NAME

lssf – list a special file

SYNOPSIS

lssf [**-f** devfile] path...

DESCRIPTION

Lssf lists a special file. The **-f** option specifies *devfile*, which is a file that describes drivers and pseudo-drivers. This file is generated by *uxgen*(1). If the **-f** option is not present, then the file **/etc/devices** is used.

For each path, *lssf* determines the major number of the special file and whether it is block or character (using *stat*(2)). *Lssf* scans *devfile* for the driver which matches the major number of the special file. When the driver is found, the minor number of the special file is decoded and a mnemonic description is printed on standard output. The mnemonics used to describe the minor number fields are closely related to the options used with *mksf*(1) which makes a special file.

As an example, suppose a special file is created with this command "mksf -d disc0 -l 1 -u 2 -s 3 dsk/c1d2s3". The command "lssf dsk/c1d2s3" will output "disc0 lu 1 unit 2 section 3 dsk/c1d2s3".

AUTHOR

Lssf was developed by HP.

FILES

/etc/devices

SEE ALSO

mksf(1), insf(1).

NAME

m4 – macro processor

SYNOPSIS

m4 [options] [files]

DESCRIPTION

M4 is a macro processor intended as a front end for Ratfor, C, and other languages. Each of the argument files is processed in order; if there are no files, or if a file name is –, the standard input is read. The processed text is written on the standard output.

The options and their effects are as follows:

–e Operate interactively. Interrupts are ignored and the output is unbuffered. Using this mode may be very difficult.

–s Enable line sync output for the C preprocessor (#line ...)

–B*int* Change the size of the push-back and argument collection buffers from the default of 4,096.

–H*int* Change the size of the symbol table hash array from the default of 199. The size should be prime.

–S*int* Change the size of the call stack from the default of 100 slots. Macros take three slots, and non-macro arguments take one.

–T*int* Change the size of the token buffer from the default of 512 bytes.

To be effective, these flags must appear before any file names and before any **–D** or **–U** flags:

–D*name*[=*val*]
Defines *name* to *val* or to null in *val*'s absence.

–U*name*
undefines *name*.

Macro calls have the form:

name(arg1,arg2, ..., argn)

The (must immediately follow the name of the macro. If the name of a defined macro is not followed by a (, it is deemed to be a call of that macro with no arguments. Potential macro names consist of alphabetic letters, digits, and underscore _, where the first character is not a digit.

Leading unquoted blanks, tabs, and new-lines are ignored while collecting arguments. Left and right single quotes are used to quote strings. The value of a quoted string is the string stripped of the quotes.

When a macro name is recognized, its arguments are collected by searching for a matching right parenthesis. If fewer arguments are supplied than are in the macro definition, the trailing arguments are taken to be null. Macro evaluation proceeds normally during the collection of the arguments, and any commas or right parentheses which happen to turn up within the value of a nested call are as effective as those in the original input text. After argument collection, the value of the macro is pushed back onto the input stream and rescanned.

M4 makes available the following built-in macros. They may be redefined, but once this is done

the original meaning is lost. Their values are null unless otherwise stated.

define — the second argument is installed as the value of the macro whose name is the first argument. Each occurrence of **$***n* in the replacement text, where *n* is a digit, is replaced by the *n*-th argument. Argument 0 is the name of the macro; missing arguments are replaced by the null string; **$#** is replaced by the number of arguments; **$*** is replaced by a list of all the arguments separated by commas; **@ $@** is like **$***, but each argument is quoted (with the current quotes).

undefine — removes the definition of the macro named in its argument.

defn — returns the quoted definition of its argument(s). It is useful for renaming macros, especially built-ins.

pushdef — like *define*, but saves any previous definition.

popdef — removes current definition of its argument(s), exposing the previous one, if any.

ifdef — if the first argument is defined, the value is the second argument, otherwise the third. If there is no third argument, the value is null. The word *unix* is predefined on HP-UX system versions of *m4*.

shift — returns all but its first argument. The other arguments are quoted and pushed back with commas in between. The quoting nullifies the effect of the extra scan that will subsequently be performed.

changequote — change quote symbols to the first and second arguments. The symbols may be up to five characters long. *Changequote* without arguments restores the original values (i.e., ` ').

changecom — change left and right comment markers from the default # and new-line. With no arguments, the comment mechanism is effectively disabled. With one argument, the left marker becomes the argument and the right marker becomes new-line. With two arguments, both markers are affected. Comment markers may be up to five characters long.

divert — *m4* maintains 10 output streams, numbered 0-9. The final output is the concatenation of the streams in numerical order; initially stream 0 is the current stream. The *divert* macro changes the current output stream to its (digit-string) argument. Output diverted to a stream other than 0 through 9 is discarded.

undivert — causes immediate output of text from diversions named as arguments, or all diversions if no argument. Text may be undiverted into another diversion. Undiverting discards the diverted text.

divnum — returns the value of the current output stream.

dnl — reads and discards characters up to and including the next new-line.

ifelse — has three or more arguments. If the first argument is the same string as the second, then the value is the third argument. If not, and if there are more than four arguments, the process is repeated with arguments 4, 5, 6 and 7. Otherwise, the value is either the fourth string, or, if it is not present, null.

incr — returns the value of its argument incremented by 1. The value of the argument is calculated by interpreting an initial digit-string as a decimal number.

decr — returns the value of its argument decremented by 1.

eval — evaluates its argument as an arithmetic expression, using 32-bit arithmetic. Operators include **+**, **−**, *****, **/**, **%**, ******
(exponentiation), bitwise **&**, **|**, **^**, and **~**; relationals; parentheses. Octal and hex numbers may be specified as in C. The second argument specifies the radix for the

result; the default is 10. The third argument may be used to specify the minimum number of digits in the result.

len	returns the number of characters in its argument.
index	returns the position in its first argument where the second argument begins (zero origin), or −1 if the second argument does not occur.
substr	returns a substring of its first argument. The second argument is a zero origin number selecting the first character; the third argument indicates the length of the substring. A missing third argument is taken to be large enough to extend to the end of the first string.
translit	transliterates the characters in its first argument from the set given by the second argument to the set given by the third. No abbreviations are permitted.
include	returns the contents of the file named in the argument.
sinclude	is identical to *include*, except that it says nothing if the file is inaccessible.
syscmd	executes the HP-UX system command given in the first argument. No value is returned.
sysval	is the return code from the last call to *syscmd*.
maketemp	fills in a string of XXXXX in its argument with the current process ID.
m4exit	causes immediate exit from *m4*. Argument 1, if given, is the exit code; the default is 0.
m4wrap	argument 1 will be pushed back at final EOF; example: m4wrap(`cleanup()')
errprint	prints its argument on the diagnostic output file.
dumpdef	prints current names and definitions, for the named items, or for all if no arguments are given.
traceon	with no arguments, turns on tracing for all macros (including built-ins). Otherwise, turns on tracing for named macros.
traceoff	turns off trace globally and for any macros specified. Macros specifically traced by *traceon* can be untraced only by specific calls to *traceoff*.

SEE ALSO

cc(1), cpp(1).

NAME

hp9000s200, hp9000s300, hp9000s500, hp9000s800, pdp11, u3b, u3b5, vax – provide truth value about your processor type

SYNOPSIS

hp9000s200

hp9000s300

hp9000s500

hp9000s800

pdp11

u3b

u3b5

vax

DESCRIPTION

The following commands will return a true value (exit code of 0) if you are on a processor that the command name indicates.

hp9000s200
True if you are on a Hewlett-Packard 9000 Series 200.

hp9000s300
True if you are on a Hewlett-Packard 9000 Series 300.

hp9000s500
True if you are on a Hewlett-Packard 9000 Series 500.

hp9000s800
True if you are on a Hewlett-Packard 9000 Series 800.

pdp11 True if you are on a PDP-11/45 or PDP-11/70.

u3b True if you are on a 3B 20S computer.

u3b5 True if you are on a 3B 5 computer.

vax True if you are on a VAX-11/750 or VAX-11/780.

The commands that do not apply will return a false (non-zero) value. These commands are often used within *make*(1) makefiles and shell procedures to increase portability.

SEE ALSO

make(1), sh(1), test(1), true(1).

NAME

mail, rmail – send mail to users or read mail

SYNOPSIS

mail [**–epqr**] [**–f** file]

mail [**–t**] [**–d**] persons

rmail [**–t**] [**–d**] persons

DESCRIPTION

Note: An enhanced user mail interface is presented in *mailx*(1).

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>.
n	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).
y [*files*]	Same as **w**.
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 command interpreter to do *command*.
*****	Print a command summary.

The optional arguments alter the printing of the mail:

–e	causes mail not to be printed. An exit value is returned: 0 = mail present 1 = no mail 2 = other error
–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.
–r	causes messages to be printed in first-in, first-out order.
–f*file*	causes *mail* to use *file* (e.g., **mbox**) instead of the default *mailfile*.

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 >. The **–t** option causes the message to be preceded by all *persons* the *mail* is sent to. 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 file **dead.letter** will be saved to allow editing and resending. Note that this is regarded as a temporary file in that it is recreated every time needed, erasing the previous contents of **dead.letter**.

The **-d** option causes *mail* to deliver mail directly. This isolates *mail* from making routing decisions and allows it to be used as a local delivery agent. Typically this option is used by autorouting facilities when they deliver mail locally.

To denote a recipient on a remote system, prefix *person* by the system name and exclamation mark (see *uucp*(1)). 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**. *Mail* will not use *uucp* if the remote system is the local system name (i.e., localsystem!user).

The *mailfile* may be manipulated in two ways to alter the function of *mail*. The *other* permissions of the file may be read-write, read-only, or neither read nor write to allow different levels of privacy. If changed to other than the default, the file will be preserved even when empty to perpetuate the desired permissions. The file may also contain the first line:

Forward to *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. In order for forwarding to work properly the *mailfile* should have "mail" as group ID, and the group permission should be read-write.

Rmail only permits the sending of mail; *uucp*(1) uses *rmail* as a security precaution.

When a user logs in, the presence of mail, if any, is indicated. Also, notification is made if new mail arrives while using *mail*.

FILES

/usr/mail/*.lock	lock for mail directory
dead.letter	unmailable text
/tmp/ma*	temporary file
$MAIL	variable containing path name of *mailfile*
$HOME/mbox	saved mail
/etc/passwd	to identify sender and locate persons
/usr/mail/*user*	incoming mail for *user*; i.e., the *mailfile*

WARNINGS

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**.

SEE ALSO

login(1), mailx(1), uucp(1), write(1).

INTERNATIONAL SUPPORT

mail: 8- and 16-bit data, 8-bit filenames.

NAME

mailx – interactive message processing system

SYNOPSIS

mailx [*options*] [*name...*]

DESCRIPTION

The command *mailx* provides a comfortable, flexible environment for sending and receiving messages electronically. When reading mail, *mailx* provides commands to facilitate saving, deleting, and responding to messages. When sending mail, *mailx* allows editing, reviewing and other modification of the message as it is entered.

Incoming mail is stored in a standard file for each user, called the system *mailbox* for that user. When *mailx* is called to read messages, the *mailbox* is the default place to find them. As messages are read, they are marked to be moved to a secondary file for storage, unless specific action is taken, so that the messages need not be seen again. This secondary file is called the *mbox* and is normally located in the user's HOME directory (see **MBOX**, in ENVIRONMENT VARIABLES below for a description of this file). Messages remain in this file until specifically removed.

On the command line, *options* start with a dash (-) and any other arguments are taken to be destinations (recipients). If no recipients are specified, *mailx* will attempt to read messages from the *mailbox*. Command line options are:

–d	Turn on debugging output. Neither particularly interesting nor recommended.
–e	Test for presence of mail. *Mailx* prints nothing and exits with a successful return code if there is mail to read.
–f [*filename*]	Read messages from *filename* instead of *mailbox*. If no *filename* is specified, the *mbox* is used.
–F	Record the message in a file named after the first recipient. Overrides the "record" variable, if set (see ENVIRONMENT VARIABLES).
–h *number*	The number of network "hops" made so far. This is provided for network software to avoid infinite delivery loops.
–H	Print header summary only.
–i	Ignore interrupts. See also "ignore" (ENVIRONMENT VARIABLES).
–n	Do not initialize from the system default *Mailx.rc* file.
–N	Do not print initial header summary.
–r *address*	Pass *address* to network delivery software. All tilde commands are disabled.
–s *subject*	Set the Subject header field to *subject*.
–u *user*	Read *user*'s *mailbox*. This is only effective if *user*'s *mailbox* is not read protected.
–U	Convert *uucp* style addresses to internet standards. Overrides the "conv" environment variable.

When reading mail, *mailx* is in *command mode*. A header summary of the first several messages is displayed, followed by a prompt indicating *mailx* can accept regular commands (see COMMANDS below). When sending mail, *mailx* is in *input mode*. If no subject is specified on the command line, a prompt for the subject is printed. As the message is typed, *mailx* will read the message and store it in a temporary file. Commands may be entered by beginning a line with the tilde (~) escape character followed by a single command letter and optional arguments. See TILDE ESCAPES for a summary of these commands.

At any time, the behavior of *mailx* is governed by a set of *environment variables*. These are flags and valued parameters which are set and cleared via the **set** and **unset** commands. See ENVIRONMENT VARIABLES below for a summary of these parameters.

Recipients listed on the command line may be of three types: login names, shell commands, or alias groups. Login names may be any network address, including mixed network addressing. If the recipient name begins with a pipe symbol (|), the rest of the name is taken to be a shell command to pipe the message through. This provides an automatic interface with any program that reads the standard input, such as *lp*(1) for recording outgoing mail on paper. Alias groups are set by the **alias** command (see COMMANDS below) and are lists of recipients of any type.

Regular commands are of the form

[**command**] [*msglist*] [*arguments*]

If no command is specified in *command mode*, **print** is assumed. In *input mode*, commands are recognized by the escape character, and lines not treated as commands are taken as input for the message.

Each message is assigned a sequential number, and there is at any time the notion of a 'current' message, marked by a '>' in the header summary. Many commands take an optional list of messages (*msglist*) to operate on, which defaults to the current message. A *msglist* is a list of message specifications separated by spaces, which may include:

n	Message number **n**.
.	The current message.
^	The first undeleted message.
$	The last message.
*****	All messages.
n–m	An inclusive range of message numbers.
user	All messages from **user**.
/string	All messages with **string** in the subject line (case ignored).
:*c*	All messages of type *c*, where *c* is one of:
	d deleted messages
	n new messages
	o old messages
	r read messages
	u unread messages

Note that the context of the command determines whether this type of message specification makes sense.

Other arguments are usually arbitrary strings whose usage depends on the command involved. File names, where expected, are expanded via the normal shell conventions (see *sh*(1)). Special characters are recognized by certain commands and are documented with the commands below.

At start-up time, *mailx* reads commands from a system-wide file (**/usr/lib/mailx/mailx.rc**) to initialize certain parameters, then from a private start-up file (**$HOME/.mailrc**) for personalized variables. Most regular commands are legal inside start-up files, the most common use being to set up initial display options and alias lists. The following commands are not legal in the start-up file: **!**, **Copy**, **edit**, **followup**, **Followup**, **hold**, **mail**, **preserve**, **reply**, **Reply**, **shell**, and **visual**. Any errors in the start-up file cause the remaining lines in the file to be ignored.

COMMANDS

The following is a complete list of *mailx* commands:

! *command* Escape to the shell. See "SHELL" (ENVIRONMENT VARIABLES).

*comment* Null command (comment). This may be useful in *.mailrc* files.

= Print the current message number.

? Prints a summary of commands.

alias *alias name...*
group *alias name...*
Declare an alias for the given names. The names will be substituted when *alias* is used as a recipient. Useful in the *.mailrc* file.

alternates *name...*
Declares a list of alternate names for your login. When responding to a message, these names are removed from the list of recipients for the response. With no arguments, **alternates** prints the current list of alternate names. See also "allnet" (ENVIRONMENT VARIABLES).

cd [*directory*]
chdir [*directory*] Change directory. If *directory* is not specified, $HOME is used.

copy [*filename*]
copy [*msglist*] *filename*
Copy messages to the file without marking the messages as saved. Otherwise equivalent to the **save** command.

Copy [*msglist*] Save the specified messages in a file whose name is derived from the author of the message to be saved, without marking the messages as saved. Otherwise equivalent to the **Save** command.

delete [*msglist*] Delete messages from the *mailbox*. If "autoprint" is set, the next message after the last one deleted is printed (see ENVIRONMENT VARIABLES).

discard [*header-field...*]
ignore [*header-field...*]
Suppresses printing of the specified header fields when displaying messages on the screen. Examples of header fields to ignore are "status" and "cc." The fields are included when the message is saved. The **Print** and **Type** commands override this command.

dp [*msglist*]
dt [*msglist*] Delete the specified messages from the *mailbox* and print the next message after the last one deleted. Roughly equivalent to a **delete** command followed by a **print** command.

echo *string...* Echo the given strings (like *echo*(1)).

edit [*msglist*] Edit the given messages. The messages are placed in a temporary file and the "EDITOR" variable is used to get the name of the editor (see ENVIRONMENT

VARIABLES). Default editor is *ed*(1).

exit
xit Exit from *mailx*, without changing the *mailbox*. No messages are saved in the *mbox* (see also **quit**).

file [*filename*]
folder [*filename*]
Quit from the current file of messages and read in the specified file. Several special characters are recognized when used as file names, with the following substitutions:

% the current *mailbox*.
%**user**
the *mailbox* for **user**.
the previous file.
& the current *mbox*.
Default file is the current *mailbox*.

folders Print the names of the files in the directory set by the "folder" variable (see ENVIRONMENT VARIABLES).

followup [*message*]
Respond to a message, recording the response in a file whose name is derived from the author of the message. Overrides the "record" variable, if set. See also the **Followup**, **Save**, and **Copy** commands and "outfolder" (ENVIRONMENT VARIABLES).

Followup [*msglist*]
Respond to the first message in the *msglist*, sending the message to the author of each message in the *msglist*. The subject line is taken from the first message and the response is recorded in a file whose name is derived from the author of the first message. See also the **followup**, **Save**, and **Copy** commands and "outfolder" (ENVIRONMENT VARIABLES).

from [*msglist*] Prints the header summary for the specified messages.

group *alias name...*
alias *alias name...*
Declare an alias for the given names. The names will be substituted when *alias* is used as a recipient. Useful in the *.mailrc* file.

headers [*message*]
Prints the page of headers which includes the message specified. The "screen" variable sets the number of headers per page (see ENVIRONMENT VARIABLES). See also the **z** command.

help Prints a summary of commands.

hold [*msglist*]
preserve [*msglist*]
Holds the specified messages in the *mailbox*.

if *s|r*
mail-commands
else
mail-commands
endif Conditional execution, where *s* will execute following *mail-commands*, up to an **else** or **endif**, if the program is in *send* mode, and *r* causes the *mail-commands* to be executed only in *receive* mode. Useful in the *.mailrc* file.

ignore *header-field...*

discard *header-field...*
Suppresses printing of the specified header fields when displaying messages on the screen. Examples of header fields to ignore are "status" and "cc." All fields are included when the message is saved. The **Print** and **Type** commands override this command.

list Prints all commands available. No explanation is given.

mail *name...* Mail a message to the specified users.

mbox [*msglist*] Arrange for the given messages to end up in the standard *mbox* save file when *mailx* terminates normally. See "MBOX" (ENVIRONMENT VARIABLES) for a description of this file. See also the **exit** and **quit** commands.

next [*message*] Go to next message matching *message*. A *msglist* may be specified, but in this case the first valid message in the list is the only one used. This is useful for jumping to the next message from a specific user, since the name would be taken as a command in the absence of a real command. See the discussion of *msglists* above for a description of possible message specifications.

pipe [*msglist*] [*command*]
| [*msglist*][*command*]
Pipe the message through the given *command*. The message is treated as if it were read. If no arguments are given, the current message is piped through the command specified by the value of the "cmd" variable. If the "page" variable is set, a form feed character is inserted after each message (see ENVIRONMENT VARIABLES).

preserve [*msglist*]
hold [*msglist*] Preserve the specified messages in the *mailbox*.

Print [*msglist*]
Type [*msglist*] Print the specified messages on the screen, including all header fields. Overrides suppression of fields by the **ignore** command.

print [*msglist*]
type [*msglist*] Print the specified messages. If "crt" is set, the messages longer than the number of lines specified by the "crt" variable are paged through the command specified by the "PAGER" variable. The default command is *pg*(1) (see ENVIRONMENT VARIABLES).

quit Exit from *mailx*, storing messages that were read in *mbox* and unread messages in the *mailbox*. Messages that have been explicitly saved in a file are deleted.

Reply [*msglist*]
Respond [*msglist*]
Send a response to the author of each message in the *msglist*. The subject line is taken from the first message. If "record" is set to a file name, the response is saved at the end of that file (see ENVIRONMENT VARIABLES).

reply [*message*]
respond [*message*]
Reply to the specified message, including all other recipients of the message. If "record" is set to a file name, the response is saved at the end of that file (see ENVIRONMENT VARIABLES).

Save [*msglist*] Save the specified messages in a file whose name is derived from the author of the first message. The name of the file is taken to be the author's name with all network addressing stripped off. See also the **Copy**, **followup**, and **Followup** commands and "outfolder" (see ENVIRONMENT VARIABLES).

save [*filename*]
save [*msglist*] *filename*
Save the specified messages in the given file. The file is created if it does not exist. The message is deleted from the *mailbox* when *mailx* terminates unless "keepsave" is set (see also ENVIRONMENT VARIABLES and the **exit** and **quit** commands).

set
set *name*
set *name=string*
set *name=number*
Define a variable called *name*. The variable may be given a null, string, or numeric value. **Set** by itself prints all defined variables and their values. See ENVIRONMENT VARIABLES for detailed descriptions of the *mailx* variables.

shell Invoke an interactive shell (see also "SHELL" (ENVIRONMENT VARIABLES)).

size [*msglist*] Print the size in characters of the specified messages.

source *filename* Read commands from the given file and return to command mode.

top [*msglist*] Print the top few lines of the specified messages. If the "toplines" variable is set, it is taken as the number of lines to print (see ENVIRONMENT VARIABLES). The default is 5.

touch [*msglist*] Touch the specified messages. If any message in *msglist* is not specifically saved in a file, it will be placed in the *mbox* upon normal termination. See **exit** and **quit**.

Type [*msglist*]
Print [*msglist*] Print the specified messages on the screen, including all header fields. Overrides suppression of fields by the **ignore** command.

type [*msglist*]
print [*msglist*] Print the specified messages. If "crt" is set, the messages longer than the number of lines specified by the "crt" variable are paged through the command

specified by the "PAGER" variable. The default command is *pg*(1) (see ENVIRONMENT VARIABLES).

undelete [*msglist*]
Restore the specified deleted messages. Will only restore messages deleted in the current mail session. If "autoprint" is set, the last message of those restored is printed (see ENVIRONMENT VARIABLES).

unset *name...* Causes the specified variables to be erased. If the variable was imported from the execution environment (i.e., a shell variable) then it cannot be erased.

version Prints the current version and release date.

visual [*msglist*] Edit the given messages with a screen editor. The messages are placed in a temporary file and the "VISUAL" variable is used to get the name of the editor (see ENVIRONMENT VARIABLES).

write [*msglist*] *filename*
Write the given messages on the specified file, minus the header and trailing blank line. Otherwise equivalent to the **save** command.

xit
exit Exit from *mailx*, without changing the *mailbox*. No messages are saved in the *mbox* (see also **quit**).

z[+|-] Scroll the header display forward or backward one screen–full. The number of headers displayed is set by the "screen" variable (see ENVIRONMENT VARIABLES).

TILDE ESCAPES

The following commands may be entered only from *input mode*, by beginning a line with the tilde escape character (~). See "escape" (ENVIRONMENT VARIABLES) for changing this special character.

~!*command* Escape to the shell.

~. Simulate end of file (terminate message input).

~:*mail-command*
~_ *mail-command*
Perform the command-level request. Valid only when sending a message while reading mail.

~? Print a summary of tilde escapes.

~**A** Insert the autograph string "Sign" into the message (see ENVIRONMENT VARIABLES).

~**a** Insert the autograph string "sign" into the message (see ENVIRONMENT VARIABLES).

~**b** *name ...* Add the *names* to the blind carbon copy (Bcc) list.

~c *name ...* Add the *names* to the carbon copy (Cc) list.

~d Read in the *dead.letter* file. See "DEAD" (under ENVIRONMENT VARIABLES) for a description of this file.

~e Invoke the editor on the partial message. See also "EDITOR" (ENVIRONMENT VARIABLES).

~f [*msglist*] Forward the specified messages. The messages are inserted into the message, without alteration.

~h Prompt for Subject line and To, Cc, and Bcc lists. If the field is displayed with an initial value, it may be edited as if you had just typed it.

~i *string* Insert the value of the named variable into the text of the message. For example, **~A** is equivalent to '**~i** Sign.'

~m [*msglist*] Insert the specified messages into the letter, shifting the new text to the right one tab stop. Valid only when sending a message while reading mail.

~p Print the message being entered.

~q Quit from input mode by simulating an interrupt. If the body of the message is not null, the partial message is saved in *dead.letter*. See "DEAD" (under ENVIRONMENT VARIABLES) for a description of this file.

~r *filename*
~< *filename*
~< !*command* Read in the specified file. If the argument begins with an exclamation point (!), the rest of the string is taken as an arbitrary shell command and is executed, with the standard output inserted into the message.

~s *string ...* Set the subject line to *string*.

~t *name ...* Add the given *names* to the To list.

~v Invoke a preferred screen editor on the partial message. See also "VISUAL" (ENVIRONMENT VARIABLES).

~w *filename* Write the partial message onto the given file, without the header.

~x Exit as with **~q** except the message is not saved in *dead.letter*.

~| *command* Pipe the body of the message through the given *command*. If the *command* returns a successful exit status, the output of the command replaces the message.

ENVIRONMENT VARIABLES

The following are environment variables taken from the execution environment and are not alterable within *mailx*.

HOME *directory*
The user's base of operations.

MAILRC *filename*
The name of the start-up file. Default is $HOME/.mailrc.

The following variables are internal *mailx* variables. They may be imported from the execution environment or set via the **set** command at any time. The **unset** command may be used to erase variables.

allnet All network names whose last component (login name) match are treated as identical. This causes the *msglist* message specifications to behave similarly. Default is **noallnet**. See also the **alt**ernates command and the "metoo" variable.

append Upon termination, append messages to the end of the *mbox* file instead of prepending them. Default is **noappend.**

askcc Prompt for the Cc list after message is entered. Default is **noaskcc**.

asksub Prompt for subject if it is not specified on the command line with the **–s** option. Enabled by default.

autoprint Enable automatic printing of messages after **d**elete and **u**ndelete commands. Default is **noautoprint**.

bang Enable the special-casing of exclamation points (!) in shell escape command lines as in *vi*(1). Default is **nobang**.

cmd = *command*
Set the default command for the **pipe** command. No default value.

conv = *conversion*
Convert uucp addresses to the specified address style. The only valid conversion now is *internet*, which requires a mail delivery program conforming to the RFC822 standard for electronic mail addressing. Conversion is disabled by default. See also "sendmail" and the –U command line option.

crt = *number* Pipe messages having more than *number* lines through the command specified by the value of the "PAGER" variable (*pg*(1) by default). Disabled by default.

DEAD = *filename*
The name of the file in which to save partial letters in case of untimely interrupt or delivery errors. Default is $HOME/dead.letter.

debug Enable verbose diagnostics for debugging. Messages are not delivered. Default is **nodebug**.

dot Take a period on a line by itself during input from a terminal as end-of-file. Default is **nodot**.

EDITOR = *command*
The command to run when the **e**dit or ~**e** command is used. Default is *ed*(1).

escape = *c* Substitute *c* for the ~ escape character.

folder = *directory*
The directory for saving standard mail files. User specified file names beginning with a plus (+) are expanded by preceding the file name with this directory name to obtain the real file name. If *directory* does not start with a slash (/), $HOME is prepended to it. In order to use the plus (+) construct on a *mailx* command line, "folder" must be an exported *sh* environment variable. There is no default for the "folder" variable. See also "outfolder" below.

header Enable printing of the header summary when entering *mailx*. Enabled by default.

hold Preserve all messages that are read in the *mailbox* instead of putting them in the standard *mbox* save file. Default is **nohold**.

ignore Ignore interrupts while entering messages. Handy for noisy dial-up lines. Default is **noignore**.

ignoreeof Ignore end-of-file during message input. Input must be terminated by a period (.) on a line by itself or by the ~. command. Default is **noignoreeof**. See also "dot" above.

keep When the *mailbox* is empty, truncate it to zero length instead of removing it. Disabled by default.

keepsave Keep messages that have been saved in other files in the *mailbox* instead of deleting them. Default is **nokeepsave**.

MBOX = *filename*
The name of the file to save messages which have been read. The xit command overrides this function, as does saving the message explicitly in another file. Default is $HOME/mbox.

metoo If your login appears as a recipient, do not delete it from the list. Default is **nometoo**.

LISTER = *command*
The command (and options) to use when listing the contents of the "folder" directory. The default is *ls*(1).

onehop When responding to a message that was originally sent to several recipients, the other recipient addresses are normally forced to be relative to the originating author's machine for the response. This flag disables alteration of the recipients' addresses, improving efficiency in a network where all machines can send directly to all other machines (i.e., one hop away).

outfolder Causes the files used to record outgoing messages to be located in the directory specified by the "folder" variable unless the pathname is absolute. Default is **nooutfolder**. See "folder" above and the **Save**, **Copy**, **followup**, and **Followup** commands.

page Used with the **pipe** command to insert a form feed after each message sent through the pipe. Default is **nopage**.

PAGER = *command*
The command to use as a filter for paginating output. This can also be used to specify the options to be used. Default is *pg*(1).

prompt = *string*
Set the *command mode* prompt to *string*. Default is "? ".

quiet Refrain from printing the opening message and version when entering *mailx*. Default is **noquiet**.

record = *filename*
Record all outgoing mail in *filename*. Disabled by default. See also "outfolder" above.

save Enable saving of messages in *dead.letter* on interrupt or delivery error. See "DEAD" for a description of this file. Enabled by default.

screen = *number*
Sets the number of lines in a screen-full of headers for the **headers** command.

sendmail = *command*
Alternate command for delivering messages. Default is *mail*(1).

sendwait Wait for background mailer to finish before returning. Default is **nosendwait**.

SHELL = *command*
The name of a preferred command interpreter. Default is *sh*(1).

showto When displaying the header summary and the message is from you, print the recipient's name instead of the author's name.

sign = *string* The variable inserted into the text of a message when the ~**a** (autograph) command is given. No default (see also ~**i** (TILDE ESCAPES)).

Sign = *string* The variable inserted into the text of a message when the ~**A** command is given. No default (see also ~**i** (TILDE ESCAPES)).

toplines = *number*
The number of lines of header to print with the **top** command. Default is 5.

VISUAL = *command*
The name of a preferred screen editor. Default is *vi*(1).

FILES

/usr/mail/*	post office directory
$HOME/.mailrc	personal start-up file
/usr/lib/mailx/mailx.help*	help message files
/usr/lib/mailx/mailx.rc	global start-up file
$HOME/mbox	secondary storage file
/tmp/R[emqsx]*	temporary files

SEE ALSO
mail(1), pg(1), ls(1).

BUGS

Where *command* is shown as valid, arguments are not always allowed. Experimentation is recommended.

Internal variables imported from the execution environment cannot be **unset**.

The full internet addressing is not fully supported by *mailx*. The new standards need some time to settle down.

Attempts to send a message having a line consisting only of a "." are treated as the end of the message by *mail*(1) (the standard mail delivery program).

NAME

make – maintain, update, and regenerate groups of programs

SYNOPSIS

make [**–f** makefile] [**–p**] [**–i**] [**–k**] [**–s**] [**–r**] [**–n**] [**–b**] [**–e**] [**–t**] [**–d**] [**–q**] [names]

DESCRIPTION

The following is a brief description of all options and some special names. Options can occur in any order.

–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.

–p Print out the complete set of macro definitions and target descriptions.

–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 When a command returns nonzero status, abandon work on the current entry, but continue on other branches that do not depend on that entry.

–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.

–r Do not use the built-in rules.

–n No execute mode. Print commands, but do not execute them. Even lines beginning with an **are printed.**

–b Compatibility mode for old (Version 7) *makefiles*.

–e Environment variables override assignments within *makefiles*.

–t Touch the target files (causing them to be up-to-date) rather than issue the usual commands.

–d Debug mode. Print out detailed information on files and times examined. (This is intended for debugging the *make* command itself.)

–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.

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 no **–f** option is present, **makefile**, **Makefile**, **s.makefile**, and **s.Makefile** are tried in order. If *makefile* is –, the standard input is taken. 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, see the **Environment** section below about **SHELL**. 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. Everything printed by *make* (except the initial tab) is passed directly to the shell as is. Thus,

```
echo a
b
```

will produce

```
ab
```

exactly the same as the shell would.

Sharp (#) and new-line surround comments before the rules. Comments in the rules depend on the setting of the **SHELL** macro.

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. The first one or two characters in a command can be the following: -, @, -@, or @-. If @ is present, printing of the command is suppressed. If - is present, *make* ignores an error. A line is printed when it is executed unless the **-s** option is present, or the entry **.SILENT:** is in *makefile*, or unless the initial character sequence contains a @. The **-n** option specifies printing without execution; however, if the command line has the string **$(MAKE)** in it, the line is always executed, see discussion below of the **MAKEFLAGS** macro under **Environment**. Note that this feature does not work if **MAKE** is enclosed in braces, as in **${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 the initial character sequence of the command contains -. 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 or implicit) 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.

Each of the commands in the rules is given to a shell to be executed. The shell that is used is determined by the **SHELL** environment variable, which is usually set to the shell with which the user logs in. To ensure the same shell is used each time a *makefile* is executed, the line:

SHELL=/bin/sh

should be put in the macro definition section of the *makefile*.

Macros

Entries of the form *string1* = *string2* are macro definitions. *String2* is defined as all characters up to a comment character or an unescaped new-line. 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 that 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 that 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 that 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 that 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 (if using **/bin/sh** as a shell):

```
make -fp - 2>/dev/null </dev/null
```

The only peculiarity in this output is the (**null**) string that *printf*(3S) prints when handed a null string.

A tilde in the above rules refers to an SCCS file, see *sccsfile*(4). 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 parentheses, it is assumed to be an archive library, the string within parentheses referring to a member within the library. Thus **lib(file.o)** and **$(LIB)(file.o)** both refer to an archive library that contains **file.o**. (This assumes the **LIB** macro has been previously defined.) The expression **$(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 very cumbersome if the archive library contains a mix of assembly programs and C programs.

FILES

[Mm]akefile and s.[Mm]akefile

SEE ALSO

cc(1), cd(1), lex(1), sh(1), yacc(1).

WARNINGS

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 that 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.

File names with the characters @ = : @ will not work.

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~)** does not 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.

INTERNATIONAL SUPPORT

8-bit data.

NAME

makekey – generate encryption key

SYNOPSIS

/usr/lib/makekey

REMARKS

The decryption facilities provided by this software are under control by the United States Government and cannot be exported without special licenses. These capabilities can be sold only to domestic customers at this time.

DESCRIPTION

Makekey improves the usefulness of encryption schemes depending on a key by increasing the amount of time required to search the key space. It reads 10 bytes from its standard input, and writes 13 bytes on its standard output. The output depends on the input in a way intended to be difficult to compute (i.e., to require a substantial fraction of a second).

The first eight input bytes (the *input key*) can be arbitrary ASCII characters. The last two (the *salt*) are best chosen from the set of digits, ., /, and upper- and lower-case letters. The salt characters are repeated as the first two characters of the output. The remaining 11 output characters are chosen from the same set as the salt and constitute the *output key*.

The transformation performed is essentially the following: the salt is used to select one of 4,096 cryptographic machines all based on the National Bureau of Standards DES algorithm, but broken in 4,096 different ways. Using the *input key* as key, a constant string is fed into the machine and recirculated a number of times. The 64 bits that come out are distributed into the 66 *output key* bits in the result.

Makekey is intended for programs that perform encryption (e.g., *ed*(1) and *crypt*(1)). Usually, its input and output will be pipes.

SEE ALSO

crypt(1), ed(1), passwd(4).

NAME

man – find manual information by keywords; print out the manual

SYNOPSIS

man **–k** keyword ...
man **–f** file ...
man [–] [section] title ...

DESCRIPTION

Man is a program which gives information from the programmer's 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 online 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.

When neither **–k** nor **–f** is specified, *man* formats a specified set of manual pages. If a section specifier is given, *man* looks in that section of the manual for the given *titles*. *Section* is an arabic section number, e.g. 3, which may be followed by a single letter classifier, e.g. 1g indicating a graphics program in section 1. If *section* is omitted, *man* searches all sections of the manual, giving preference to commands over subroutines in system libraries, and printing the first section it finds, if any. Each *title* is truncated to at most 11 characters to assure that there will be room for the *section*. The files in the **/usr/man** directories also truncate the *title* portion of the file name at 11 characters to assure room for the suffix in the 14 character file name.

If the standard output is a teletype, or if the flag – is given, *man* pipes its output through *more (1)*, with the **–s** option, to stop after each page.

Man searches in three directories for the target file. First *man* searches in **/usr/man**, then in **/usr/contrib/man**, and finally in **/usr/local/man**. Within each of these directories, first *man* searches in the cat* subdirectory, and then in the man* subdirectory. If, on first access to the cat* subdirectory, the target file is not present, *man* retrieves it from the man* subdirectory, formats it, and (if cat* exists) installs it in cat*. If only the cat* subdirectory is present and/or nroff is not installed, only those pages which have been preformatted are displayable.

FILES

/usr/contrib/man/cat*/*
/usr/contrib/man/man*/*
/usr/local/man/cat*/*
/usr/local/man/man*/*
/usr/man/cat*/*
/usr/man/man*/*

SEE ALSO

catman(1M), more(1), rmnl(1), ul(1), whereis(1).

BUGS

The manual is supposed to be reproducible either on the phototypesetter or on a typewriter. However, on a typewriter some information is necessarily lost.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

man - find manual information by keywords; print out the manual

SYNOPSIS

man –**k** keyword ...
man –**f** file ...
man [–] [section] title ...

DESCRIPTION

Man is a program which gives information from the programmer's 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 online 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.

When neither –**k** nor –**f** is specified, *man* formats a specified set of manual pages. If a section specifier is given, *man* looks in that section of the manual for the given *titles*. *Section* is an arabic section number, e.g. 3, which may be followed by a single letter classifier, e.g. 1g indicating a graphics program in section 1. If *section* is omitted, *man* searches all sections of the manual, giving preference to commands over subroutines in system libraries, and printing the first section it finds, if any. Each *title* is truncated to at most 11 characters to assure that there will be room for the *section*. The files in the **/usr/man** directories also truncate the *title* portion of the file name at 11 characters to assure room for the suffix in the 14 character file name.

If the standard output is a teletype, or if the flag – is given, *man* pipes its output through *more (1)*, with the –**s** option, to stop after each page.

Man searches in three directories for the target file. First *man* searches in **/usr/man**, then in **/usr/contrib/man**, and finally in **/usr/local/man**. Within each of these directories, *man* searches in the cat*.Z subdirectory, the man*.Z subdirectory, the cat* subdirectory, and the man* subdirectory to find the most recent version of the manual entry. If the man* or man*.Z file is most recent, or if the file is not present in cat* or cat*.Z, *man* retrieves it from the man* or man*.Z subdirectory and formats it. If the cat*.Z subdirectory exists, the formatted version is compressed and installed in cat*.Z. Otherwise, if the cat* subdirectory exists, the formatted version is installed in cat*. The files in man*.Z and cat*.Z are in compressed form and must be uncompressed before they may be displayed. If only the cat* or cat*.Z subdirectory is present and/or nroff is not installed, only those pages which have been preformatted are displayable.

FILES

/usr/man/cat*[.Z]/*
/usr/man/man*[.Z]/*
/usr/contrib/man/cat*[.Z]/*
/usr/contrib/man/man*[.Z]/*
/usr/local/man/cat*[.Z]/*
/usr/local/man/man*[.Z]/*

SEE ALSO

catman(1M), more(1), rmnl(1), ul(1), whereis(1), compress(1).

BUGS

The manual is supposed to be reproducible either on a phototypesetter or on a typewriter. However, on a typewriter some information is necessarily lost.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

mediainit – initialize hard disk, flexible disk, or cartridge tape media

SYNOPSIS

mediainit [–**vr**] [–**f** fmt_optn] [–**i** interleave] pathname

DESCRIPTION

Mediainit initializes mass storage media by formatting the media, writing and reading test patterns to verify media integrity, then sparing any defective blocks found. This process prepares the disk or tape for error-free operation. Initialization destroys all existing user data in the area being initialized.

The following command options are recognized. They can be specified in any order, but all must precede the pathname. Options without parameters can be listed individually or grouped together. Options with parameters must be listed individually, but white space between the option and its parameter is discretionary.

–**v** Normally, *mediainit* provides only fatal error messages, and they are directed to diagnostic output (stderr). The –**v** (verbose) option sends device-specific information related to low-level operation of *mediainit* to standard output (stdout). This option is most useful to trained service personnel because it usually requires detailed knowledge of device operation before the information can be interpreted correctly.

–**r** The –**r** (re-certify) option forces a complete tape certification whether or not the tape has been certified previously. All record of any previously spared blocks is discarded, so any bad blocks will have to be rediscovered. This option should be used only if: (a) it is suspected that numerous blocks on the tape have been spared which should not have been, or (b) it is necessary to destroy (overwrite) all previous data on the tape.

–**f** *fmt_optn* The format option is a device-specific number in the range **0** through **239.** It is intended solely for use with certain SS/80 devices that support multiple media formats (independent from interleave factor). For example, certain microfloppy drives support 256-, 512-, and 1024-byte sectors. *Mediainit* passes any supplied format option directly through to the device. The device then either accepts the format option if it is supported or rejects it if it is not supported. Refer to device operating manuals for additional information. The default format option is **0**.

–**i** *interleave* The interleave factor, interleave, refers to the relationship between sequential logical records and sequential physical records. It defines the number of physical records on the media that lie between the beginning points of two consecutively numbered logical records. The choice of interleave factor can have a substantial impact on disk performance. For CS/80 and SS/80 drives, consult the appropriate device operating manual for details. For Amigo drives, see HARDWARE DEPENDENCIES

Pathname *Pathname* is the path name to the character (raw) device special file associated with the device unit or volume that is to be initialized. *Mediainit* aborts if you lack either read or write permission to the device special file, or if the device is currently open for any other process. This prevents accidental initialization of the root device or any mounted volume. See HARDWARE DEPENDENCIES for additional Series 800 requirements. *Mediainit* locks the unit or volume being initialized so that no other processes can access it.

When a given CS/80 or SS/80 device contains multiple units or a given unit contains multiple volumes as defined by the drive controller, any available unit or volume associated with that controller can be initialized, independent of other units and volumes that share the same controller.

Thus, you can initialize one unit or volume to any format or interleave factor without affecting formats or data on companion units or volumes. However, be aware that the entire unit or volume (as defined by the drive controller) is initialized without considering the possibility that it may be subdivided into smaller structures by the the operating software. When such structures exist, unexpected loss of data is possible.

Mediainit dominates controller resources and limits access by competing processes to other units or volumes sharing the came controller. If other simultaneous processes need access to the same controller, some access degradation can be expected until initialization is complete; especially if you are initializing a tape cartridge in a drive that shares the root disk controller. See Series 800 HARDWARE DEPENDENCIES for additional Series 800 information.

In general, *mediainit* attempts to carefully check any **-f** (format option) or **-i** (interleave options) supplied, and aborts if an option is out of range or inappropriate for the media being initialized. Specifying an interleave factor or format option value of **0** has the same effect as not specifying the option at all.

For disks that support interleave factors, the acceptable range is usually **1** (no interleave) through *N*-**1**, where *N* is the number of sectors per track. With SS/80 hard disks, the optimum interleave factor is usually determined by the speed (normal or high) of the HP-IB interface card used and whether DMA is present in the system. The optimum interleave factor for SS/80 flexible disk drives is usually a constant (often **2**), and is independent of the type of HP-IB interface used. The optimum interleave factor for CS/80 disks is usually **1** and is also usually not related to the type of HP-IB interface being used. In any case, refer to the appropriate device operating manual for recommended values.

If a disk being initialized requires an interleave factor but none is specified, *mediainit* provides an appropriate, though not necessarily optimum default. For CS/80 and SS/80 disks, *mediainit* uses whatever the device reports as its current interleave factor. SS/80 floppy drives report their minimum (usually best) interleave factor, if the currently installed media is unformatted.

When a given device supports format options, the allowable range of interleave factors may be related to the specified format option. In such instances, *mediainit* cannot check the interleave factor if one is specified.

RETURNS

Mediainit returns a value of **0** upon successful completion, a value of **1** if there was a device-related error, or a value of **2** if there was a syntax-related error.

Appropriate error messages are printed out to stderr during the execution of *mediainit.*

EXAMPLES

The following example formats an HP 9122 SS/80 3-1/2″ flexible disk with an interleave factor of 2, 1024-byte sectors, double-sided HP format:

```
mediainit -i 2 -f 3 /dev/r9122
```

HARDWARE DEPENDENCIES

Series 200, Series 300

Series 200, and Series 300 systems support various Amigo disk drives. Acceptable interleave factors for Amigo devices are as follows:

Device	Range	Default
HP 9895 SS/DS	1 - 29	2
HP 8290X	1 - 15	3
HP 9121	1 - 15	2
HP 9133V	na	9
HP 9133XV	na	9
HP 9134XV	na	9

Series 800

Pathname must be a device special file whose minor number for the section of the device being initialized has the diagnostic bit set (the diagnostic bit is the most significant bit in the minor number).

For a device that contains multiple units on a single controller or multiple sections within a unit, each unit or section can be initialized independently from any other unit or section. It should be noted, however, that *mediainit* requires that there be no other processes accessing the unit or section before initialization begins, regardless of which unit or section is being initialized. If there are accesses currently in progress, *mediainit* aborts. During the initialization process, *open*(2) rejects all other accesses to the device being initialized, producing the error [EACCES].

WARNINGS

Aborting *mediainit* is likely to leave the medium in a corrupt state, even if it was previously initialized. To recover, the initialization must be restarted.

AUTHOR

Mediainit was developed by HP.

SEE ALSO

mkfs(1M), newfs(1M), lifinit(1).

NOTES

Most types of mass storage media must be initialized before they can be used. HP hard disks, flexible disks, and cartridge tapes require some form of initialization, but 9-track tapes do not. Initialization usually involves formatting the media, writing and reading test patterns, then sparing any defective blocks. Depending upon the media and device type, none, some, or all of the initialization process may have been performed at the factory. *Mediainit* completes whatever steps are appropriate to prepare the media for error-free operation.

Most HP hard disks are formatted and exhaustively tested at the factory by use of a process more thorough but also more time-consuming than appropriate for *mediainit*. However, *mediainit* is still valuable for ensuring the integrity of the media after factory shipment, formatting with the correct interleave factor, and sparing any blocks which may have become defective since original factory testing was performed.

HP flexible disks are not usually formatted prior to shipment, so they must undergo the entire initialization process before they can be used.

All HP CS/80 cartridge tapes are certified and formatted prior to shipment from the factory, When a tape is certified, it is throroughly tested and defective blocks are spared at that time. While *Mediainit* usually certifies a tape only if it has not been certified previously. If the tape has been previously certified and spared, *mediainit* usually reorganizes the tape's spare block table, retaining any previous spares, but optimizing their assignment for maximum performance under sequential access. Reorganizing the spare block table takes only a few seconds, whereas complete certification takes about a half-hour for 150-foot tapes, and over an hour for 600-foot tapes.

HP CS/80 cartridge tape drives have a feature called "auto-sparing", where if under normal usage the drive has trouble reading a block, the drive logs the fact then automatically spares out that

block the next time data is written to it. Thus, as a tape is used, any marginal blocks that were not spared during certification are spared automatically if they cause problems. This sparing is automatic within the device, and is totally independent of *Mediainit*.

Reorganization of a tape's spare block table technically renders any existing data undefined, but the data is not usually destroyed by overwriting. To ensure that old tape data is destroyed (useful for security reasons among other things), complete tape re-certification can forced with the **-r** option.

Some applications may require that a file system be placed on the media before use. *Mediainit* does not create a file system; it only prepares media for writing and reading. Other utilities such as *newfs*(1M), *lifinit*(1) or *mkfs*(1M). must be invoked after running *mediainit,* if such a file system is required.

NAME

mesg – permit or deny messages to terminal

SYNOPSIS

mesg [**n**] [**y**]

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 ...

DESCRIPTION

Mkdir creates specified directories in mode 777 (possibly altered by *umask*(1)). 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), sh(1), umask(1).

DIAGNOSTICS

Mkdir returns exit code 0 if all directories were successfully made; otherwise, it prints a diagnostic and returns non-zero.

INTERNATIONAL SUPPORT

8-bit filenames, messages.

NAME

mksf – make a special file

SYNOPSIS

mksf [**-f** devfile] **-d disc0** [**-l** lu] [**-u** unit] [**-s** *section*] [**-c**] [**-t**] [*path...*]

mksf [**-f** devfile] **-d gpio0** [**-l** lu] [*path...*]

mksf [**-f** devfile] **-d instr0** [**-l** lu] [**-a** address] [*path...*]

mksf [**-f** devfile] **-d lpr0** [**-l** lu] [**-c**] [**-n**] [**-r**] [**-t**] [*path...*]

mksf [**-f** devfile] **-d mux0** [**-l** lu] [**-p** port] [**-h**|**-i**|**-o**] [**-c**] [*path...*]

mksf [**-f** devfile] **-d mux1** [**-l** lu] [*path...*]

mksf [**-f** devfile] **-d pseudo** [**-m** minor] path...

mksf [**-f** devfile] **-d tape0** [**-l** lu] [**-b** bpi] [**-s** *section*] [**-a**|**-u**] [**-c**] [**-n**] [**-t**] [**-w**] [*path...*]

mksf [**-f** devfile] **-d tape1** [**-l** lu] [**-b** bpi] [**-s** *section*] [**-a**|**-u**] [**-c**] [**-n**] [**-t**] [**-w**] [*path...*]

DESCRIPTION

Mksf makes a special file. The **-f** option specifies *devfile*, which is a file that describes drivers and pseudo-drivers. This file is generated by *uxgen*(1). If the **-f** option is not present, then the file **/etc/devices** is used. The **-d** option specifies the driver name. Other options depend on the driver name.

Mksf scans *devfile* to determine the major number of the driver. While scanning *devfile*, the lu (logical unit) is checked for validity. Hence, a special file may not be created for a device which is not in the *devfile*.

Some of the common arguments used are:

-l	lu is the logical unit number of a device. lu is assigned by *uxgen*(1).
-t	transparent mode (normally used by diagnostics).
path	name of the special file. *Path* is created in the current directory. If the file already exists, it will be removed and then created. Most drivers have a default name for *path*.

Each driver has a specific set of arguments which are shown in the following sections.

DISC0

-c	This argument must be present if the unit is a cartridge tape.
-u *unit*	The cs80 unit number (eg. unit 0 - disc, unit 1 - tape).
-r	Raw, use character major not block major.
-s *section*	The section number.
path	The default *path* name depends on the arguments **-r** **-c**: rct/c<lu>d<unit>s<section>, if **-r** and **-c** rdsk/c<lu>d<unit>s<section>, if **-r** and not **-c** ct/c<lu>d<unit>s<section>, if not **-r** and **-c**

dsk/c<lu>d<unit>s<section>, if not **-r** and not **-c**

GPIO0

path The default path name is gpio<lu>.

INSTR0

-a *address* The hpib instrument address (0-31).

path The default path name is hpib/<lu>a<address>.

LPR0

-c Caps. Uppercase all output.

-n No form-feed.

-r Raw.

path default path name is lpr<lu> or rlpr<lu> (if **-r**).

MUX0

-c CCITT.

-h Hardwired (direct connect).

-i Callin.

-o Callout.

-p *port* Multiplexor port number (0-5).

path The default path name is tty<lu>p<port>

MUX1

path Default path name is mux<lu>.

TAPE0/TAPE1

-a Att style rewind/close.

-b *bpi* Bits per inch. Values of bpi are 800, 1600, 6250.

-c RTE compatible close.

-n No rewind on close.

-u UC Berkeley style rewind/close.

.rm]B

NAME

mkstr – extract error messages from C source into a file

SYNOPSIS

mkstr [–] messagefile prefix file ...

DESCRIPTION

Mkstr examines a C program and creates a file containing error message strings used by the program. Programs with many error diagnostics can be made much smaller by referring to places in the file, and reduce system overhead in running the program.

Mkstr processes each of the specified *files*, placing a revised version of each in a file whose name consists of the specified *prefix* concatenated in front of the original name. A typical usage of *mkstr* would be

```
mkstr mystrings xx *.c
```

This command would cause all the error messages from the C source files in the current directory to be placed in the file *mystrings* and revised copies of the source for these files to be placed in files whose names are prefixed with *xx*.

When processing the error messages in the source for transfer to the message file, *mkstr* searches for the string **error(**" in the input file. Each time it is encountered, the C string starting after the leading quote is placed in the message file, followed by a null character and a new-line character. The null character terminates the message so that it can be easily used when retrieved, and the new-line character makes it possible to conveniently *cat* the error message file to review its contents.

The modified copy of the input file is identical to the original, except that each occurrence of any string that was moved to the error message file is replaced by an offset pointer usable by *lseek* to retrieve the message.

If the command line includes the optional –, extracted error messages are placed at the end of the specified message file instead of overwriting it. This enables you to process individual files that are part of larger programs that have been previously processed by *mkstr* without reprocessing all the files.

All functions used by the original program whose names end in "error" that also can take a constant string as their first argument should be rewritten so that they search for the string in the error message file.

For example, a program based on the previous example usage would resemble the following:

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>

char errfile[] = "mystrings";

error(offset, a2, a3, a4)
int offset, a1, a2, a3;
{
    char msg[256];
    static int fd = -1;

        if (fd < 0) {
            fd = open(errfile, O_RDONLY);
```

```
            if (fd < 0) {
                perror(errfile);
                exit(1);
            }
        }

        if (lseek(fd, (off_t) offset, 0) | | read(fd, msg, 256) <= 0) {
            printf("? Can't find error message in %s:\n", errfile);
            perror(errfile);
            exit(1);
        }

        printf(msg, a1, a2, a3);
    }
```

SEE ALSO

lseek(2), perror(3C), xstr(1).

BUGS

Strings in calls to functions whose names end in 'error', notably *perror*(3C), may be replaced with offsets by *mkstr*.

Calls to error functions whose first argument is not a string constant are left unmodified without warning.

NAME

mm, osdd – print/check documents formatted with the MM macros

SYNOPSIS

mm [options] [files]

osdd [options] [files]

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 **$TERM** from the environment (see *profile*(4) and *environ*(5)) as the value of *term*, if **$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*.

–t Causes *mm* to invoke *tbl*(1).

–E Invokes the **–e** option of *nroff*.

–y Causes *mm* to use the non-compacted version of the macros (see *mm*(5)).

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 –
```

HINTS

1. *Mm* invokes *nroff* with the **–h** flag. With this flag, *nroff* assumes that the terminal has tabs set every 8 character positions.
2. Use the **–o***list* option of *nroff* 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* 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* (to stop between pages of output), use line-feed (rather than return or new-line) to restart the output. The **–s** option of *nroff* 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), cw(1), env(1), nroff(1), tbl(1), profile(4), mm(5), term(5).

MM–Memorandum Macros in *HP-UX Selected Articles.*

DIAGNOSTICS

mm "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 [**–n**] [**–cdflsu**] [+linenumber] [+/pattern] [name ...]

page [more options]

REMARKS:

The use of *pg*(1) is preferred.

DESCRIPTION

More is a filter which allows examination of continuous text, one screenful at a time, on a soft-copy terminal. It is quite similar to *pg*(1), and the user is encouraged to use *pg* instead. It is retained for backward compatibility. It normally pauses after each screenful, printing **--More--** at the bottom of the screen. If the user then types a carriage return, one more line is displayed. If the user hits 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 postions, 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" terminfo 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 uses terminfo descriptor files to determine terminal characteristics, and to determine the default window size, see *term*(4). 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* or the *csh* command *setenv MORE -c* would cause all invocations of *more*, including invocations by programs such as *man* and *msgs*, to use this mode. Normally, the user will place the command sequence which sets up the *MORE* environment variable in the *.profile* or *.cshrc* 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).
^D	display 11 more lines (a "scroll"). If *i* is given, then the scroll size is set to *i*.
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 with *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 n 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* will stop sending output, and will display 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*(1), 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

/usr/lib/more.help	help file
/usr/lib/terminfo/?/*	compiled terminal capability data base

VARIABLES

MORE	Default paging mode.

AUTHOR

More was developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

csh(1), man(1), pg(1), sh(1), term(4), terminfo(4), environ(5).

INTERNATIONAL SUPPORT

more: 8- and 16-bit data, 8-bit filenames, messages.

NAME

mt – magnetic tape manipulating program

SYNOPSIS

mt [**–t** tapename] command [count]

DESCRIPTION

Mt is used to give commands to the tape drive. If *tapename* is not specified, /dev/mt/0mn 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/rmt/0mn	(or whatever drive is used) must be described as a Berkeley-compatibility mode tape drive (without rewind) for *mt* to operate as expected.

AUTHOR

Mt was developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

dd(1), mt(7).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

neqn – format mathematical text for nroff

SYNOPSIS

neqn [**–dxy**] [**–sn**] [**–fn**] [**–pn**] [files]

DESCRIPTION

Neqn is a preprocessor for *nroff*(1) for typesetting mathematical text on typewriter-like terminals. Usage is almost always:

neqn files | nroff

or equivalent.

If no files are specified (or if – is specified as the last argument), *nroff* reads from the standard input. A line beginning with **.EQ** marks the start of an equation; the end of an equation is marked by a line beginning with **.EN**. Neither of these lines is altered, so they may be defined in macro packages to get centering, numbering, etc. It is also possible to designate two characters as *delimiters*; subsequent text between delimiters is then treated as *neqn* input. Delimiters may be set to characters *x* and *y* with the command-line argument **–d***xy* or (more commonly) with **delim** *xy* between **.EQ** and **.EN**. The left and right delimiters may be the same character; the dollar sign is often used as such a delimiter. Delimiters are turned off by **delim off**. All text that is neither between delimiters nor between **.EQ** and **.EN** is passed through untouched.

Tokens within *neqn* are separated by spaces, tabs, new-lines, braces, double quotes, tildes, and circumflexes. Braces { } are used for grouping; generally speaking, anywhere a single character such as *x* could appear, a complicated construction enclosed in braces may be used instead. Tilde (~) represents a full space in the output, circumflex (^) half as much.

Subscripts and superscripts are produced with the keywords **sub** and **sup**. Thus:

x sub j makes $x sub j$,

a sub k sup 2 produces:

$a sub k sup 2$,

while:

$e sup {x sup 2 + y sup 2}$ is made with *e sup {x sup 2 + y sup 2}*.

Fractions are made with **over**:

a over b yields $a over b$;

sqrt makes square roots:

1 over sqrt {ax sup 2+bx+c} results in $1 over sqrt {ax sup 2 +bx+c}$.

The keywords **from** and **to** introduce lower and upper limits:

$lim from {n -> inf} sum from 0 to n x sub i$ is made with *lim from {n -> inf } sum from 0 to n x* sub i" .

Left and right brackets, braces, etc., of the right height are made with **left** and **right**:

left [x sup 2 + y sup 2 over alpha right] ~=~ 1 produces $left [x sup 2 + y sup 2 over alpha right] ~=~ 1$.

Legal characters after **left** and **right** are braces, brackets, bars, **c** and **f** for ceiling and floor, and *""* for nothing at all (useful for a right-side-only bracket). A **left** *thing* need not have a matching **right** *thing*.

Vertical piles of *things* are made with **pile**, **lpile**, **cpile**, and **rpile**:

pile {a above b above c} produces $pile {a above b above c}$.

Piles may have arbitrary numbers of elements; **lpile** left-justifies, **pile** and **cpile** center (but with different vertical spacing), and **rpile** right justifies.

Matrices are made with **matrix**:

matrix { lcol { x sub i above y sub 2 } ccol { 1 above 2 } } produces $matrix { lcol { x sub i above y sub 2 } ccol { 1 above 2 } }$.

In addition, there is **rcol** for a right-justified column.

Diacritical marks are made with **dot**, **dotdot**, **hat**, **tilde**, **bar**, **vec**, **dyad**, and **under**:

x dot = f(t) bar is $x dot = f(t) bar$,
y dotdot bar ~=~ n under is $y dotdot bar ~=~ n under$, and
x vec ~=~ y dyad is $x vec ~=~ y dyad$.

Point sizes and fonts can be changed with **size** *n* or **size** ±*n*, **roman**, **italic**, **bold**, and **font** *n*. Point sizes and fonts can be changed globally in a document by **gsize** *n* and **gfont** *n*, or by the command-line arguments **–s***n* and **–f***n*.

Normally, subscripts and superscripts are reduced by 3 points from the previous size; this may be changed by the command-line argument **–p***n*.

Successive display arguments can be lined up. Place **mark** before the desired lineup point in the first equation; place **lineup** at the place that is to line up vertically in subsequent equations.

Shorthands may be defined or existing keywords redefined with **define**:

define thing % replacement %

defines a new token called *thing* that will be replaced by *replacement* whenever it appears thereafter. The *%* may be any character that does not occur in *replacement*.

Keywords such as **sum** (sum), **int** (int), **inf** (inf), and shorthands such as >= (>=), != (!=), and -> (->) are recognized. Greek letters are spelled out in the desired case, as in **alpha** (alpha), or **GAMMA** (GAMMA). Mathematical words such as **sin**, **cos**, and **log** are made Roman automatically. *Nroff*(1) four-character escapes such as **\(dd** (‡) and **\(bu** (•) may be used anywhere. Strings enclosed in double quotes (*"..."*) are passed through untouched; this permits keywords to be entered as text, and can be used to communicate with *nroff*(1) when all else fails. Details are given in the manuals cited below.

SEE ALSO

Typesetting Mathematics–User's Guide by B. W. Kernighan and L. L. Cherry.

New Graphic Symbols for EQN and NEQN by C. Scrocca.
cw(1), mm(1), nroff(1), tbl(1), mm(5).

BUGS

To embolden digits, parentheses, etc., it is necessary to quote them, as in **bold "12.3"**.
See also *BUGS* under ***nroff***(1).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

newform – change or reformat a text file

SYNOPSIS

newform [–**i**tabspec] [–**o**tabspec] [–**l**n] [–**b**n] [–**e**n] [–**c**char] [–**p**n] [–**a**n] [–**f**] [–**s**] [files]

DESCRIPTION

Newform reads lines from the named *files*, or the standard input if no input file is named, and reproduces the lines on the standard output. Lines are reformatted in accordance with command line options in effect.

Except for –**s**, command line options may appear in any order, may be repeated, and may be intermingled with the optional *files*. Command line options are processed in the order specified. This means that option sequences like "–**e**15 –**l**60" will yield results different from "–**l**60 –**e**15". Options are applied to all *files* on the command line.

–**i***tabspec* Input tab specification: expands tabs to spaces, according to the tab specifications given. *Tabspec* recognizes all tab specification forms described in *tabs*(1). In addition, *tabspec* may be —, in which *newform* assumes that the tab specification is to be found in the first line read from the standard input (see *fspec*(4)). If no *tabspec* is given, *tabspec* defaults to –**8**. A *tabspec* of –**0** expects no tabs; if any are found, they are treated as –**1**.

–**o***tabspec* Output tab specification: replaces spaces by tabs, according to the tab specifications given. The tab specifications are the same as for –**i***tabspec*. If no *tabspec* is given, *tabspec* defaults to –**8**. A *tabspec* of –**0** means that no spaces will be converted to tabs on output.

–**l***n* Set the effective line length to *n* characters. If *n* is not entered, –**l** defaults to 72. The default line length without the –**l** option is 80 characters. Note that tabs and backspaces are considered to be one character (use –**i** to expand tabs to spaces).

–**b***n* Truncate *n* characters from the beginning of the line when the line length is greater than the effective line length (see –**l***n*). Default is to truncate the number of characters necessary to obtain the effective line length. The default value is used when –**b** with no *n* is used. This option can be used to delete the sequence numbers from a COBOL program as follows:

```
newform –l1 –b7 file-name
```

The –**l1** must be used to set the effective line length shorter than any existing line in the file so that the –**b** option is activated.

–**e***n* Same as –**b***n* except that characters are truncated from the end of the line.

–**c***k* Change the prefix/append character to *k*. Default character for *k* is a space.

–**p***n* Prefix *n* characters (see –**c***k*) to the beginning of a line when the line length is less than the effective line length. Default is to prefix the number of characters necessary to obtain the effective line length.

–**a***n* Same as –**p***n* except characters are appended to the end of a line.

–**f** Write the tab specification format line on the standard output before any other lines are output. The tab specification format line which is printed will correspond to the format specified in the *last* –**o** option. If no –**o** option is specified, the line which is printed will contain the default specification of –**8**.

–**s** Shears off leading characters on each line up to the first tab and places up to 8 of the sheared characters at the end of the line. If more than 8 characters (not counting the first tab) are sheared, the eighth character is replaced by a * and any characters to the right of it are discarded. The first tab is always discarded.

An error message and program exit will occur if this option is used on a file without a tab on each line. The characters sheared off are saved internally until all other options specified are applied to that line. The characters are then added at the end of the processed line.

For example, to convert a file with leading digits, one or more tabs, and text on each line, to a file beginning with the text, all tabs after the first expanded to spaces, padded with spaces out to column 72 (or truncated to column 72), and the leading digits placed starting at column 73, the command would be:

newform –s –i –l –a –e file-name

DIAGNOSTICS

All diagnostics are fatal.

usage: ...	*Newform* was called with a bad option.
not -s format	There was no tab on one line.
can't open file	Self-explanatory.
internal line too long	A line exceeds 512 characters after being expanded in the internal work buffer.
tabspec in error	A tab specification is incorrectly formatted, or specified tab stops are not ascending.
tabspec indirection illegal	A *tabspec* read from a file (or standard input) may not contain a *tabspec* referencing another file (or standard input).

EXIT CODES

0 – normal execution
1 – for any error

SEE ALSO

fspec(4), csplit(1), tabs(1).

BUGS

Newform normally only keeps track of physical characters; however, for the **–i** and **–o** options, *newform* will keep track of backspaces in order to line up tabs in the appropriate logical columns.

Newform will not prompt the user if a *tabspec* is to be read from the standard input (by use of **–i—** or **–o—**).

If the **–f** option is used, and the last **–o** option specified was **–o—**, and was preceded by either a **–o—** or a **–i—**, the tab specification format line will be incorrect.

NAME

newgrp – log in to a new group

SYNOPSIS

newgrp [–] [group]

DESCRIPTION

Newgrp changes a user's group identification. The user remains logged in and the current directory is unchanged, but calculations of access permissions to files are performed with respect to the new real and effective group IDs. The user is always given a new shell, replacing the current shell, by *newgrp*, regardless of whether it terminated successfully or due to an error condition (i.e., unknown group).

Exported variables retain their values after invoking *newgrp*; however, all unexported variables are either reset to their default value or set to null. Environment variables (such as PS1, PS2, PATH, MAIL, and HOME), unless exported by the system or explicitly exported by the user, are reset to default values. For example, a user has a primary prompt string (**PS1**) other than **$** (default) and has not exported **PS1**. After an invocation of *newgrp* , successful or not, their **PS1** will now be set to the default prompt string **$**. Note that the shell command *export* (see *sh*(1)) is the method to export variables so that they retain their assigned value when invoking new shells.

With no arguments, *newgrp* changes the group identification back to the group specified in the user's password file entry.

If the first argument to *newgrp* is a –, the environment is changed to what would be expected if the user actually logged in again.

A password is demanded if the group has a password and the user does not, or if the group has a password and the user is not listed in **/etc/group** as being a member of that group.

FILES

/etc/group	system's group file
/etc/passwd	system's password file

SEE ALSO

login(1), sh(1), group(4), passwd(4), environ(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.

INTERNATIONAL SUPPORT

8- and 16-bit data.

NAME

news – print news items

SYNOPSIS

news [**–a**] [**–n**] [**–s**] [items]

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."

Options

–a Causes *news* to print all items, regardless of currency. In this case, the stored time is not changed.

–n Causes *news* to report the names of the current items without printing their contents, and without changing the stored time.

–s 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

/usr/news/*
$HOME/.news_time
/etc/profile

SEE ALSO

mail(1), profile(4), environ(5).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

nice – run a command at low priority

SYNOPSIS

nice [–increment] command [arguments]

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**.

An *increment* larger than 19 is equivalent to 19.

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.

NOTES

Nice is built into *csh*(1) with a slightly different syntax than described here. The form "nice +10" nices to positive nice, and "nice -10" can be used by the super-user to give a process more of the processor.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

nl – line numbering filter

SYNOPSIS

nl [–btype] [–htype] [–ftype] [–p] [–vstart#] [–iincr] [–ssep] [–wwidth] [–nformat] [–lnum] [–ddelim] file

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 told 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 meanings 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 suppressed;

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).

INTERNATIONAL SUPPORT

8-bit data and filenames.

NAME

nm – print name list of common object file

SYNOPSIS

nm [**–oxefuV**] file...

DESCRIPTION

The *nm* command displays the symbol table of each common object file *file*. *File* may be relocatable or absolute common object file, or it may be an archive of relocatable or absolute common object files. For each symbol, at least the following information will be printed:

Name The name of the symbol.

Value Its value expressed as an offset or an address depending on its storage class.

Size Its size in bytes, if available.

The output of *nm* may be controlled using the following options:

–o Print the *value* and *size* of a symbol in octal instead of decimal.

–x Print the *value* and *size* of a symbol in hexadecimal instead of decimal.

–e Print only external and static symbols.

–f Produce full output. Print redundant symbols (.text, .data and .bss), normally suppressed.

–u Print undefined symbols only.

–V Print the version of the *nm* command executing on the standard error output.

HARDWARE DEPENDENCIES

Series 800

The size of a symbol is not available, so there is no column for it in the output. The following additional fields are printed for each symbol:

Scope The scope of the symbol (undefined, static, or external).

Type The type of the symbol (code, data, common, absolute, etc.).

Subspace The subspace to which the symbol belongs.

Additional options on Series 800 systems:

–h Do not display the output header data.

–v Sort symbols by *value* before they are printed.

–n Sort symbols by *name* before they are printed.

–p Produce easily parsable, terse output. Each symbol *name* is preceded by its value (blanks if undefined) and one of the letters **U** (undefined), **A** (absolute), **T** (text symbol), **D** (data symbol), **B** (bss symbol), or **C** (common symbol). If the symbol is local (non-external), the type letter is in lower case.

–r Prefix each output line with the name of the object file or archive.

–T By default, *nm* prints the entire name of the symbols listed. Since object files can have symbol names with an arbitrary number of characters, a name that is longer than the width of the column set aside for names will overflow its column, forcing every column after the name to be misaligned. The **–T** option causes *nm* to truncate every name that would otherwise overflow its column and place an asterisk as the last character in the displayed name to mark it as truncated.

The **–e** and **–f** options are meaningless on Series 800 systems and are ignored.

SEE ALSO

cc(1), ld(1).

INTERNATIONAL SUPPORT

8-bit filenames.

NAME

nm – print name list (symbol table) of object file

SYNOPSIS

nm [–**gnoprsu**] [file ...]

DESCRIPTION

The *nm* command 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 (blanks if undefined) and one of the letters **U** (undefined), **A** (absolute), **T** (text segment symbol), **D** (data segment symbol), or **B** (bss segment symbol). If the symbol is local (non-external) the type letter is in lower case. The output is sorted alphabetically.

The options are:

–g	Print only global (external) symbols.
–n	Sort numerically rather than alphabetically.
–o	Precede each output line with the file or archive element name, rather than printing the file or archive element name only once. This option can be used to make piping to *grep*(1) more meaningful.
–p	Do not 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(4), ar(4).

NAME

nm – print name list (symbol table) of object file

SYNOPSIS

nm [**–gnopru**] [file ...]

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]

DESCRIPTION

Nohup executes *command* with hangups and quits ignored. If output is not re-directed by the user, both standard output and standard error are 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**.

EXAMPLE

It is frequently desirable to apply *nohup* to pipelines or lists of commands. This can be done only by placing pipelines and command lists in a single file, called a shell procedure. One can then issue:

nohup sh file

and the *nohup* applies to everything in *file*. If the shell procedure *file* is to be executed often, then the need to type *sh* can be eliminated by giving *file* execute permission. Add an ampersand and the contents of *file* are run in the background with interrupts also ignored (see *sh*(1)):

nohup file &

An example of what the contents of *file* could be is:

tbl ofile | eqn | nroff > nfile

SEE ALSO

chmod(1), nice(1), sh(1), signal(2).

WARNINGS

Be careful to place punctuation properly, for example in the following command:

nohup command1; command2

nohup applies only to *command1*, and the following command is syntactically incorrect:

nohup (command1; command2)

Be careful of where standard error is redirected. The following command may put error messages on tape, making it unreadable:

nohup cpio –o <list >/dev/rmt/1m&

while

nohup cpio –o <list >/dev/rmt/1m 2>errors&

puts the error messages into file *errors*.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

nroff – format text

SYNOPSIS

nroff [options] [files]

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.

Nroff is best not used directly, but rather via macro packages such as *mm* or *ms* which provide a high-level interface to document processing, as opposed to the very low level one provided directly in *nroff*.

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].***name*.

–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 *names* 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, **2631** for the Hewlett Packard 2631 line printer, and **klp** for a (generic) 16-bit character printer having ratio of 2 to 3 in 8-bit and 16-bit character width.

–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/macros/*	standard macro files
/usr/lib/term/*	terminal driving tables for *nroff*
/usr/lib/suftab	suffix hyphenation tables
/tmp/ta$#	temporary file
/usr/lib/tmac/tmac.*	standard macro files and pointers

SEE ALSO

mm(1).

NROFF/TROFF User's Manual in *HP-UX: Selected Articles.*

BUGS

When *nroff* is used with the **-o***list* option inside a pipeline, it may cause a harmless "broken pipe" diagnostic if the last page of the document is not specified in *list*.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

od, xd – octal and hexadecimal dump

SYNOPSIS

od [**–bcdosx**] [file] [[**+**][**0x**]offset[**.**][**b**]]
xd [**–bcdosx**] [file] [[**+**][**0x**]offset[**.**][**b**]]

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. An offset field is inserted at the beginning of each line. For *od*, the offset is in octal, for *xd* the offset is in hexadecimal.

Options

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.

–s Interpret 16-bit words in signed decimal.

–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, and is normally interpreted as octal bytes. Interpretation can be altered as follows:

offset must be preceded by **+** if the file argument is omitted.

offset preceded by **0x** is interpreted in hexadecimal.

offset followed by **.** is interpreted in decimal.

offset followed by **b** is interpreted in blocks of 512 bytes.

Dumping continues until end-of-file.

SEE ALSO

adb(1).

INTERNATIONAL SUPPORT

od: 8- and 16-bit data, 8-bit filenames, messages.

NAME

pack, pcat, unpack – compress and expand files

SYNOPSIS

pack [–] [**–f**] name ...

pcat name ...

unpack name ...

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 will force packing of *name*. This is useful for causing an entire directory to be packed even if some of the files will not benefit. If *pack* is successful, *name* will be 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;
the file is empty;
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;
if the unpacked file cannot be created.

SEE ALSO

cat(1).

NAME

pam – Personal Applications Manager, a visual shell

SYNOPSIS

pam [**–c** args ...]

DESCRIPTION

Pam is a program that helps provide a friendlier, less intimidating means of communication between HP-UX and system users. It provides many of the traditional capabilities supported by other shell programs such as executing commands as foreground processes (where you must wait until one command has been completed before the system accepts the next command) or background processes (where the command runs in the background while you perform other tasks in the foreground). *Pam* also supports other useful capabilities such as using substituted files instead of standard input and standard output, pipelining several processes into a single command, and handling shell scripts and programs. *Pam* maintains a continuous display of the open folder (current directory), and makes use of windowing and mouse I/O facilities when they are available on the system.

Display

The *pam* display has two parts. The top two lines are called the command area, while the remainder of the display is the folder (directory) area. The first line in the command area displays messages (such as prompts and errors) from the system to the user, while the second line displays input commands and text from the user to the system. *Pam* maintains a buffer of 20 command lines. You can use shifted arrow keys, BACK SPACE, INSERT CHARACTER, and DELETE CHARACTER to access and edit any existing current or previous command line in the 20-line command buffer. For example, each time you press SHIFT–UP ARROW, the next previous command line in the buffer is displayed.

The folder (lower) area is used by *pam* to display those files that reside in the currently open folder; that is, the current directory. One of the file names displayed in the folder area is highlighted. This highlighted filename identifies which file in the folder is to be used as a filename parameter for *pam* commands that are invoked using the *pam* menu. The highlighted file name can be changed by using TAB, SHIFT–TAB and arrow keys.

Commands

A command is a sequence of non-blank words separated by blanks. In general, the first word is the name of the command, and the words that follow are passed as arguments to the invoked command. Two or more commands (together with their associated arguments, if any) separated by a vertical bar (|) form a pipeline. To provide a path for passing data between commands in a pipeline, the standard output from one command in the pipeline is connected to the standard input of the next command in the pipe.

To force *pam* to complete execution of the current command or pipeline before running another command, place a semicolon (;) at the end of the line. If you prefer to perform other tasks while the command or pipeline is being executed, run the first command as a "background" process by adding an ampersand (&) at the end of the command line. *Pam* then starts the command, and, without waiting for completion, returns for your next instruction.

In a windowed system, interactive command inputs are treated as background processes (&) unless a semicolon is present at the end of the line. In non-windowed systems, commands taken from a script or from interactive command inputs are run to completion before the next command is accepted for execution (;) unless an ampersand is present at the end of the command line.

Sequences of more than one command or pipeline can be joined on a single command line by placing a semicolon or ampersand (but not both) between each adjacent pair of commands/pipelines in the line. In such constructs, commands separated by ";" are executed

in sequence (the first command is run to completion before the next is begun). Commands separated by "&" are executed simultaneously on a timesharing basis (this does not necessarily result in the most efficient use of computer resources due to timesharing overhead as sharing processes compete for processor time). Note that when ";" or "&" is used to separate commands, standard output from one command is not automatically connected to standard input for the next. Use the pipeline connector (|) instead when data must be passed between successive commands or programs, or redirect standard output and input to and from a specified file.

Pam runs commands based on the file type of the command name:

program (executable) - The command name is run (exec'ed) or, if it is a shell script, the commands in the script are run.

folder (directory) - The command name (folder) becomes the new open folder. This is equivalent to *cd* folder.

data (non-executable) - The command name (data file) is displayed one page at a time.

Standard Input, Output, and Error Files

The standard input, output and error of a command can be redirected using the following syntax:

< *name* Use the file *name* as standard input for the command.

> *name* Use the file *name* as standard output for the command.

>> *name* Use the file *name* as standard output for the command, but concatenate the output to the end of the file.

^ *name* Use the file *name* as standard error for the command.

^^ *name* Use the file *name* as standard error for the command, but concatenate the output to the current end of the file, if it exists.

name [For windowed systems only] Use the named window as standard input, output and error for the command. If the window doesn't exist then a window is created. Specific redirection of I/O with >, >>, <, |, ^, or ^^ overrides any redirection specified with "#".

I/O redirection is possible only with an associated command. Multiple redirections of standard input, output, and error associated with a command are not allowed. The I/O redirection can be placed anywhere in the command.

If a command is followed by "&" (background process), the default standard input for the command is the empty file "/dev/null".

Using Patterns to Represent Filenames

Each word in the command line (command name, parameter, redirection file name, window name) is scanned for the characters *, ?, and [. If one of these characters appears, the word is treated as a pattern that represents more than one filename. *Pam* replaces the pattern word with alphabetically sorted filenames corresponding to the pattern. If no file name is found that matches the pattern, the word is left unchanged. A period character (.) at the start of a filename 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

Characters can be quoted on the *pam* command line to prevent *pam* from performing special processing on the characters (such as <, >, #, space, |, ;, &, *, ?, [). A pair of double (") or single (') quote characters can be used to enclose the character string being quoted. The \ character quotes only the single following character.

Scripts

A script is an executable file containing command lines and comments. A comment is a line that begins with "!" or "#"; comments are ignored by *pam*. The command lines in the script file are executed in sequence (unless non-sequential execution is explicitely specified using the "&" character).

Script arguments that are specified when a script is run can be accessed by script commands using the notation "$1" for the first argument, "$2" for the second argument, etc. All arguments can be accessed at once using "$*". The name of the script can be accessed using "$0".

Autost

If a file named *Autost* exists in the open folder when *pam* is started, it is automatically processed as a command. If *Autost* is a script file, it is run as **source** *Autost* (see Built-In Commands described later). Otherwise, it is processed as if it were entered as a command (for example, if *Autost* is a data file, it is viewed). *Pam* does not process any command input until processing of the *Autost* file is complete.

Environment

The *pam* environment is set up when *pam* is run, and can be reset at any time by using the command *getenv*. The environment variables are read from a file and are not sorted or checked for syntax by *pam*. *Pam* passes the current environment to commands that it starts and uses the following environment variables in running commands:

ACTION — The ACTION variable specifies a command name (corresponding to an executable file), and is used whenever a data file is specified as a command. The ACTION command is run in this case and the data file is passed as the first argument. The default value for ACTION is "view".

HOME — The HOME variable specifies a folder and is used whenever the "cd" command is run without an argument. The HOME folder specifies the directory to change to in this case. The default value for HOME is "/".

LANG — The LANG variable is added by *pam* to its environment file if it is not already there. This happens when *pam* is initialized or when the getenv command is done. The value of the LANG variable is set to match the language that the system is localized for.

PATH — The PATH variable specifies a list of folders. When *pam* runs a command it looks for it in the folders in the PATH list.

SCRSHELL — The SCRSHELL variable specifies the shell to be used by *pam* in running scripts. If the specified name does not contain a "/" then *pam* searches for the shell using the PATH environment variable. If the SCRSHELL is undefined or the specified shell does not exist, the script

is processed by *pam*.

Menu

The *pam* menu displays the following softkey menu labels corresponding to the indicated function keys:

[function key 1] **open**, **view**, or **start** (a program), or **reread**
[f2] **echo**
[f3] **send** or **arrow** (toggle key)
[f4] **move**
[f5] **copy**
[f6] **rename**
[f7] **delete**
[f8] **close**

The command associated with a menu item is run whenever the item is selected by pressing the corresponding function key. The highlighted file name in the folder area of the display is used as a parameter for the command.

The menu item associated with **f3** is used to toggle the semantics of the arrow keys and is available only in non-windowed systems. **f3** is initially set to use the arrow keys for manipulating the position of the file highlight in the folder area. **f3** alternately controls movement of the cursor on the command line.

Built-In Commands

Several commands are executed directly by *pam*:

cd [name] — Make the named folder (directory) the open folder (current directory). If no folder is specified, the HOME environment variable is used to determine which folder to open.

close — Closes the open folder and displays the parent folder.

copy *name1* [*name2*]
copy *name1* [*name2* ...] *folder_name* — Copy *name1* to *name2*; if exists and is not a folder, it is overwritten. If only *name1* is specified, the copy is completed with the command *to*name2. If the last parameter specified is a folder, all the specified files are copied into that folder.

delete *name1* [*name2* ...] — The named files and folders (if empty) are deleted.

echo [**arg ...**] — Arguments are written to standard output. The echo menu item (menu item 2 and/or function key 2) writes the full pathname of the highlighted file in the folder area of the display to the command line.

getenv *name* — The named file is read in and used as the active environment.

makefolder *name1* [*name2* ...] — Folders are created and given the specified name(s).

move *name1* [*name2*]
move *name1* [*name2*] *folder_name* — Rename file or folder *name1* to *name2*. If *name2* exists and is a file, it is overwritten. If only *name1* is specified, use the command *to name2* to complete the move. If the

last parameter is a folder, all the specified files are moved into that folder.

netunam *pathname* [*string*]
Initiate a network connection to the specified system (as indicated by *pathname*) using the specified login (as indicated by *string*). If *string* is omitted, the network connection to the specified system, if currently active, is disconnected.

print *name1* [*name2* ...] Print the specified files on the designated system printer.

rename *name1* [*name2*]
rename *file_name1* [*file_name2* ...] *folder_name*
Same as **move.**

reread Reread the open folder and update the display. The keystroke CONTROL-L also does a reread.

send Send the full *pathname* of the highlighted *filename* in the folder area to an application as if it were typed from the keyboard (windowed systems only).

source *name* [*arg* ...] Read command lines from the named script file and execute them. A shell is NOT forked to execute the commands. Parameter substitution for the arguments (*arg* ...) is handled the same way as during regular script execution.

stopprint Stop current printing activity if it was started by the **print** command.

to [*name*] Complete a pending copy, move or rename. The file or folder *name* identifies the destination for a preceding **copy**, **move**, or **rename** command that had no destination specified. If *name* is omitted, the destination defaults to the current open folder.

view *name1* [*name2* ...] Copy the specified file(s) to standard output. If standard output is the screen (default), the file is displayed one page at a time.

Signals

Pam ignores INTERRUPT and QUIT signals if the command is followed by an "&"; otherwise *pam* uses default signal handling when running commands.

Invoking *Pam*

Pam can be invoked as a keyboard command or from a program. When *pam* is invoked without **-c** as the first argument, *pam* acts as an interactive, display-oriented command interpreter.

When *pam* is invoked with **-c** as the first argument (either from the keyboard or from a running program), the remaining arguments in the command are interpreted as command inputs intended for processing by *pam*. The list of arguments intended as commands for *pam* must not exceed a total of 160 characters. When the **-c** option is used, *pam* executes the list of command arguments (built-in commands, redirection, pipes, and most other *pam* facilities can be used), then exits.

Exiting from *Pam*

To terminate *pam* and return to normal HP-UX operation, press CONTROL-D or CONTROL-C.

HARDWARE DEPENDENCIES

With a non-windowed *pam* running on certain terminals the shifted right and left arrow keys cannot be used to move the cursor on the command line.

IPC: *pam* runs commands with the SIGHUP signal ignored.
The *netunam* built-in command is not supported.

Series 300/500: The built-in commands *print*, *stopprint*, and *send* are not supported.
The environment variable LANG is not set up by *pam*.

FILES

/rom/PAM
/rom/.environ
/tmp/Plock
Autost
/rom/PAMmsg
/usr/lib/nls/n-computer/pam.cat
/dev/null

NAME

passwd – change login password

SYNOPSIS

passwd [name]

DESCRIPTION

This command changes or installs a password associated with the login *name*. If *name* is omitted, it defaults to *getlogin*(3C) name.

Ordinary users may change only the password which corresponds to their login *name*.

Passwd prompts ordinary users for their old password, if any. It then prompts for the new password twice. The first time the new password is entered *passwd* checks to see if the old password has "aged" sufficiently. If "aging" is insufficient the new password is rejected and *passwd* terminates; see *passwd*(4).

Assuming "aging" is sufficient, a check is made to insure that the new password meets construction requirements. When the new password is entered a second time the two copies of the new password are compared. If the two copies are not identical the cycle of prompting for the new password is repeated for at most two more times.

Passwords must be constructed to meet the following requirements:

Each password must have at least six characters. Only the first eight characters are significant.

Each password must contain at least two alphabetic characters and at least one numeric or special character. In this case, "alphabetic" means upper and lower case letters.

Each password must differ from the user's login *name* and any reverse or circular shift of that login *name*. For comparison purposes, an upper case letter and its corresponding lower case letter are equivalent.

New passwords must differ from the old by at least three characters. For comparison purposes, an upper case letter and its corresponding lower case letter are equivalent.

One whose effective user ID is zero is called a super-user; see *id*(1), and *su*(1). Super-users may change any password; hence, *passwd* does not prompt super-users for the old password. Super-users are not forced to comply with password aging and password construction requirements. A super-user can create a null password by entering a carriage return in response to the prompt for a new password.

FILES

/etc/passwd

SEE ALSO

id(1), login(1), su(1), crypt(3C), passwd(4).

INTERNATIONAL SUPPORT

8- and 16-bit data, messages.

NAME

paste – merge same lines of several files or subsequent lines of one file

SYNOPSIS

paste file1 file2 ...
paste **–d** list file1 file2 ...
paste **–s** [**–d** list] file1 file2 ...

DESCRIPTION

In the first two forms, *paste* concatenates corresponding lines of the given input files *file1*, *file2*, etc. It treats each file as a column or columns of a table and pastes them together horizontally (parallel merging). If you will, it is the counterpart of *cat*(1) which concatenates vertically, i.e., one file after the other. In the last form above, *paste* replaces the function of an older command with the same name by combining subsequent lines of the input file (serial merging). In all cases, lines are glued together with the *tab* character, or with characters from an optionally specified *list*. Output is to the standard output, so it can be used as the start of a pipe, or as a filter, if – is used in place of a 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

cut(1), grep(1), pr(1).

NOTES

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.

INTERNATIONAL SUPPORT

8-bit data and filenames.

NAME

pathalias – electronic address router

SYNOPSIS

pathalias [**–icv**] [**–l** host] [**–d** arg] [**–t** arg] [files]

DESCRIPTION

Pathalias computes the shortest paths and corresponding routes from one host (computer system) to all other known, reachable hosts. *Pathalias* reads host-to-host connectivity information on standard input or in the named *files*, and writes a list of host-route pairs on the standard output.

Options are:

–i Ignore case: map all host names to lower case. By default, case is significant.

–c Print costs: print the path cost (see below) before each host-route pair.

–v Verbose: report some statistics on the standard error output.

–l *host*
Set local host name to *host*. By default, *pathalias* discovers the local host name in a system-dependent way.

–d *arg*
Declare a dead link, host, or network (see below). If *arg* is of the form "host1!host2," the link from host1 to host2 is treated as an extremely high cost (*i.e.*, DEAD) link. If *arg* is a single host name, that host is treated as dead and is used as an intermediate host of last resort on any path. If *arg* is a network name, the network requires a gateway.

–t *arg*
Trace input for link, host or network on the standard error output. The form of *arg* is as above.

The public domain version of *pathalias* includes two undocumented options, that are briefly described in the Special Options section below.

Input Format

A line beginning with white space continues the preceding line. Anything following '#' on an input line is ignored.

A list of host-to-host connections consists of a "from" host in column 1, followed by white space, followed by a comma-separated list of "to' hosts, called *links*. A link may be preceded or followed by a network character to use in the route. Valid network characters are '!' (default), '@', ':', and '%'. A link (and network character, if present) may be followed by a "cost" enclosed in parentheses. Costs may be arbitrary arithmetic expressions involving numbers, parentheses, '+', '–', '*', and '/'. The following symbolic costs are recognized:

LOCAL	25	(local-area network connection)
DEDICATED	95	(high speed dedicated link)
DIRECT	200	(toll-free call)
DEMAND	300	(long-distance call)
HOURLY	500	(hourly poll)
EVENING	1800	(time restricted call)
DAILY	5000	(daily poll, also called POLLED)
WEEKLY	30000	(irregular poll)

In addition, DEAD is a very large number (effectively infinite), and HIGH and LOW are –5 and +5 respectively, for baud-rate or quality bonuses/penalties. These symbolic costs represent an imperfect measure of bandwidth, monetary cost, and frequency of connections. For most mail traffic, it is important to minimize the number of intermediaries in a route, thus, *e.g.*, HOURLY is far greater than DAILY / 24. If no cost is given, a default of 4000 is used.

For the most part, arithmetic expressions that mix symbolic constants other than HIGH and LOW make no sense. For example, if a host calls a local neighbor whenever there is work, and additionally polls every evening, the cost is DIRECT, **not** DIRECT+EVENING.

Some examples:

down	princeton!(DEDICATED), tilt, %thrash(LOCAL)
princeton	topaz!(DEMAND+LOW)
topaz	@rutgers(LOCAL)

If a link is encountered more than once, the least-cost occurrence dictates the cost and network character. Links are treated as bidirectional, to the extent that a DEAD reverse link is assumed unless better information is available.

The set of names by which a host is known by its neighbors is called its *aliases*. Aliases are declared as follows:

name = alias, alias ...

The name used in the route to or through aliased hosts is the name by which the host is known to its predecessor in the route.

Fully connected networks, such as the ARPANET or a local-area network, are declared as follows:

net = {host, host, ...}

The host-list may be preceded or followed by a routing character, and may be followed by a cost:

princeton-ethernet = {down, up, princeton}!(LOCAL)
ARPA = @{sri-unix, mit-ai, su-score}(DEDICATED)

The routing character used in a route to a network member is the one encountered when "entering" the network. See also the sections on *gateways* and *domains*.

Connection data may be given while hiding host names by declaring

private {host, host, ...}

Pathalias will not generate routes for private hosts, but may produce routes through them. The scope of a private declaration extends from the declaration to the end of the input file in which it appears. It is best to put private declarations at the beginning of the appropriate input file.

Output Format

A list of host-route pairs is written to the standard output, where route is a string appropriate for use with *printf*(3S), *e.g.*,

rutgers princeton!topaz!%s@rutgers

The "%s" in the route string should be replaced by the user name at the destination host. (This task is normally performed by a mailer.)

Except for *domains* (see below), the name of a network is never used in expansions. Thus, in the earlier example, the path from down to up would be "up!%s," not "princeton-ethernet!up!%s."

Gateways

A network is represented by a pseudo-host and a set of network members. Links from the members to the network have the weight given in the input, while the cost from the network to the members is zero. If a network is declared dead on the command line (with the **-d** option), the member-to-network links are marked dead, which discourages paths to members by way of the network.

If the input also shows a link from a host to the network, then that host will be preferred as a gateway. Gateways need not be network members.

For example, suppose CSNET is declared dead on the command line and the input contains

```
CSNET = {...}
csnet-relay          CSNET
```

Then routes to CSNET hosts will use csnet-relay as a gateway.

Domains

A host or network whose name begins with '.' is called a domain. Domains are presumed to require gateways, *i.e.*, they are DEAD. The route given by a path through a domain is similar to that for a network, but here the domain name is tacked onto the end of the next host. Subdomains are permitted. For example,

```
harvard              .EDU
.EDU = {.BERKELEY}
.BERKELEY            ernie
```

yields

```
ernie                ...!harvard!ernie.BERKELEY.EDU!%s
```

Output is given for the nearest gateway to a domain, *e.g.*, the example above gives

```
.EDU                 ...!harvard!%s
```

Special Options

The public domain version of *pathalias* includes two undocumented options that rewrite named files with intermediate data of limited usage. Here are brief descriptions:

−g *file* Dump graph edges into *file* in the form "host>host" for simple connections and "host@<tab>host" for network connections (from hosts to networks only).

−s *file* Dump shortest path tree into *file* in the form "host<tab>[@]host[!](cost)", including both connections from hosts to networks and from networks to hosts. This data may be useful for generating lists of one-way connections.

BUGS

The **−i** option should be the default.

The order of arguments is significant. In particular, **−i** and **−t** should appear early.

Pathalias can generate hybrid (*i.e.* ambiguous) routes, which are abhorrent and most certainly should not be given as examples in the manual entry.

Multiple '@'s in routes are prohibited by many mailers, so *pathalias* resorts to the "magic %" rule when appropriate. This convention is not documented anywhere, including here.

AUTHOR

Pathalias was developed by Peter Honeyman and Steven M. Bellovin.

FILES

newsgroup mod.map Likely location of some input files.

NAME

pc – Pascal compiler

SYNOPSIS

pc [options] files

REMARKS

This manual page describes the generic HP Pascal compiler; implementation dependencies for different machines are noted as needed.

DESCRIPTION

Pc is the HP standard Pascal compiler. It accepts several types of file arguments:

(1) Arguments whose names end with **.p** are taken to be Pascal source files. They are each compiled, and each corresponding object program or module(s) 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 be immediately deleted (leaving only the linked executable file) if only a single source is compiled and linked, if the **–C** option is specified, or if the source fails to compile correctly.

(2) All other file arguments, including those whose names end with **.o** or **.a**, are passed on to the linker (*ld*(1)) to be linked into the final program.

Arguments can be passed to the compiler through the PCOPTS environment variable as well as on the command line. The compiler picks up the value of PCOPTS and places its contents before any arguments on the command line. For example (in *sh*(1) notation),

```
$ PCOPTS=-v
$ export PCOPTS
$ pc -L prog.p
```

is equivalent to

```
$ pc -v -L prog.p
```

Options

The following options are recognized:

–A Produce warnings for the use of non-ANSI-Pascal features. (Same as **$ANSI ON$**).

–C Suppress code generation. No **.o** files will be created and linking will be suppressed. This is effectively a request for syntax/semantic checking only (same as **$CODE OFF$**).

–c Suppress linking and only produce object (**.o**) files from source files.

–g Generate additional information needed by a symbolic debugger, and ensure that the program is linked as required. See the appropriate implementation reference manual for more information on symbolic debugging support.

–L Write a program listing to *stdout* (see the HARDWARE DEPENDENCIES section below for exceptions).

–l*x* Cause the linker to search first in the library named **/lib/lib***x***.a** and then in **/usr/lib/lib***x***.a** (see *ld*(1)).

–N Cause the output file from the linker to be marked as unshareable (see **–n**). For details and system defaults, refer to the linker documentation (*ld*(1)).

–n Cause the output file from the linker to be marked as shareable (see **–N**). For details and system defaults, refer to the linker documentation (*ld*(1)).

–o *outfile* Name the output file from the linker *outfile* instead of **a.out**.

-P *lines* Specifies the number of lines (including any header or trailer) which should be listed per page of generated listing (same as **$LINES n$**).

-Q Cause the output file from the linker to be marked as not demand loadable (see **-q**). For details and system defaults, refer to the linker documentation (*ld*(1)).

-q Cause the output file from the linker to be marked as demand loadable (see **-Q**). For details and system defaults, refer to the linker documentation (*ld*(1)).

-s Cause the output of the linker to be *stripped* of symbol table information (see *ld*(1) and *strip*(1)). (This option is incompatible with symbolic debugging.)

-t *c,name* Substitute or insert subprocess *c* with *name* where *c* is one or more of an implementation-defined set of identifiers indicating the subprocess(es). This option works in two modes: 1) if *c* is a single identifier, *name* represents the full pathname of the new subprocess; 2) if *c* is a set of (more than one) identifiers, *name* represents a prefix to which the standard suffixes are concatenated to construct the full path name of the new subprocesses.

The values *c* can assume are:

c compiler body (standard suffix is *pascomp*)
0 same as *c*
l linker (standard suffix is *ld*)

-v Enable verbose mode, producing a step-by-step description of the compilation process on *stderr*.

-w Suppress warning messages (same as **$WARN OFF$**).

-W *c,arg1*[,*arg2*,...,*argN*]

Cause *arg1* through *argN* to be handed off to subprocess *c*. The *argi* are of the form *-argoption*[,*argvalue*], where *argoption* is the name of an option recognized by the subprocess and *argvalue* is a separate argument to *argoption* where necessary. The values that *c* can assume are those listed under the **-t** option, as well as the value *d* (driver program), which has a special meaning explained below.

For example, the specification to pass the **-r** (preserve relocation information) option to the linker would be:

-W l,-r

The **-W d** option specification allows additional, implementation-specific options to be recognized and passed through the compiler driver to the appropriate compiler subprocesses. For example, on Series 500:

-W d,-U

will send the option **-U** to the driver and compiler. Furthermore, a shorthand notation for this mechanism can be used by prepending **+** to the option name; as in

+U

which is equivalent to the previous option expression. Note that for simplicity this shorthand is applied to each implementation-specific option individually, and that the *argvalue* is no longer separated from the *argoption* by a comma (see **-W**).

HARDWARE DEPENDENCIES

Series 200, 300, 500:

The following option is supported:

−Y Enable 16-bit Native Language Support when parsing string literals and comments (same as **NLS_SOURCE**). Note that 8-bit parsing is always supported.

Series 200, 300 and Integral PC:

The **−L** option writes a program listing to the file given in the **$LIST filename$** option in the source, instead of to *stdout*.

The following option is implemented only on the Series 200, 300 and the Integral PC:

+a Cause the compiler to generate archived object (**.a**) files instead of simple object (**.o**) files. This allows source files containing multiple HP Pascal modules to be compiled such that each module can be linked independently. Use of this option is discouraged for portability reasons. It is provided to facilitate migration from Series 200 HP-UX releases prior to Release 5.1 (where **.a** files were always generated), to 5.1 and subsequent releases. Otherwise, it is recommended that modules needing separate linkability be placed in separate source files. To facilitate use of previously existing makefiles and scripts that depended on the archive generation, the PCOPTS environment variable can be used (e.g., set PCOPTS="+a $PCOPTS").

Series 200, 300:

The following options are implemented only on the Series 200 and 300:

+A Cause the compiler to always use 2-byte data and stack alignment rules instead of default 4-byte alignment rules for stacks and data objects exceeding four bytes when generating object code for MC68020 systems (see reference manual and **+x** option below for more details).

+X Cause the compiler to generate "generic" MC68010 code. The code can also run on an MC68020 microprocessor, but cannot use its expanded capabilities.

+x Cause the compiler to generate code that utilizes the expanded capabilities of the MC68020 and generates in-line code for the MC68881.

+M Cause the compiler not to generate inline code for the MC68881 floating-point coprocessor. Library routines will be referenced for math and intrinsic operations. This option is meaningless on MC68010 based systems or in conjunction with **+X**.

Series 500:

The following options are not supported: **−b**, **−e**, and **−f**.

The following options must be specified with the **−W d,...** option or the **+** shorthand: **−E**, **−F**, **−H** [*bytes*], and **−W** [*bytes*].

The following options are implemented only on the Series 500:

+E Cause the program to be linked with the library **/lib/libpcesc.a**, which transforms all execution errors (HP-UX signals, Pascal run-time errors, Pascal I/O errors and HP-UX errors) into escapes. This differs from the default library **/lib/libpccat.a**, which prints the appropriate error message and aborts the program.

+F Cause the compiler to generate information for use by various program analyzers.

+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.

+Q *dfile* Cause *dfile* to be read before compilation of each source file. *Dfile* may only contain compiler options.

+U Cause the compiler to upshift externally visible names. Default is lowercase (same as **$UPSHIFT_LEVEL1 ON$**).

+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.

To use the **+H** or **+W** options on an executable file other than **a.out**, the file to be examined (modified) must be specified with the **−o** option. For example, to set the heap of program **foo** to **1000000**, use:

pc +H 1000000 -o foo

do not use:

pc +H 1000000 -W l,-o,foo

Series 800:

The following options are not supported: **−P**, **−t**, **−w**.

The following option is implemented only on the Series 800:

+O*opt* Invoke optimizations selected by *opt*. If *opt* is '1', then only level 1 optimizations are handled. If *opt* is '2', then all optimizations are performed. The option **+O2** is the same as **−O**.

FILES

file.p	input file (Pascal source file)
file.a	any archive file to be searched at link time (or for the Series 200, Series 300, and Integral PC only, optionally generated object archive file)
file.o	compiler-generated or other object file that is to be relocated at link time
a.out	linked executable output file
/usr/lib/pascomp	compiler
/usr/lib/paserrs	compiler error message file
/usr/lib/escerrs	Pascal escape codes (Series 200, Series 300, and Integral PC only)
/usr/lib/syserrs	HP-UX system messages (Series 200, Series 300, and Integral PC only)
/usr/lib/ioerrs	Pascal I/O results (Series 200, Series 300, and Integral PC only)
/lib/crt0.o	runtime startup (except Series 500)
/lib/prt0.o	runtime startup (Series 500 only)
/lib/libpc.a	Pascal run-time library (except Series 800)
/lib/libcl.a	Pascal run-time library (Series 800 only)
/lib/libm.a	HP–UX math library (Series 200, Series 300, and Integral PC only)
/lib/libpccat.a	Pascal run-time library, reports errors and aborts program (Series 500 only)
/lib/libpcesc.a	Pascal run-time library, translates errors into escapes (Series 500 only)
/lib/libc.a	HP-UX system library (C-language library)
/usr/tmp/*	temporary files used by the compiler; names are created by *tmpnam*(3S).

SEE ALSO

HP Pascal Language Reference for Series 200, Series 300 (also valid for Integral PC).

Pascal/9000 Language Reference Manual, for Series 500.

Programming in Pascal with Hewlett-Packard Pascal, by Peter Grogono.

DIAGNOSTICS

The diagnostics produced by *pc* are intended to be self-explanatory. Occasional messages may be produced by the linker.

A list of all compiler errors may be found in **/usr/lib/paserrs**.

If a listing is requested (–L option), errors are written to the listing file (*stdout*). If a listing is requested and either or both of *stdout/stderr* has been redirected to something other than a terminal, errors will also be written to *stderr*. If no listing is requested (no –L option), errors are written to *stderr*. This effectively guarantees that *stderr* will always receive error messages, unless that would result in duplication of error messages printed on the terminal.

INTERNATIONAL SUPPORT

8- and 16-bit data only in strings and comments, 8-bit filenames.

NAME

pg – file perusal filter for soft-copy terminals

SYNOPSIS

pg [*–number*] [**–p** *string*] [**–cefns**] [*+linenumber*] [*+/pattern/*] [files...]

REMARKS

The decryption facilities provided by this software are under control by the United States Government and cannot be exported without special licenses. These capabilities can be sold only to domestic customers at this time.

DESCRIPTION

The *pg* command is a filter which allows the examination of *files* one screenful at a time on a soft-copy terminal. (The file name – and/or NULL arguments indicate that *pg* should read from the standard input.) Each screenful is followed by a prompt. If the user types a carriage return, another page is displayed; other possibilities are enumerated below.

This command is different from previous paginators in that it allows you to back up and review something that has already passed. The method for doing this is explained below.

In order to determine terminal attributes, *pg* scans the *terminfo*(4) data base for the terminal type specified by the environment variable **TERM**. If **TERM** is not defined, the terminal type **dumb** is assumed.

The command line options are:

–number	An integer specifying the size (in lines) of the window that *pg* is to use instead of the default. (On a terminal containing 24 lines, the default window size is 23).
–p *string*	Causes *pg* to use *string* as the prompt. If the prompt string contains a "%d", the first occurrence of "%d" in the prompt will be replaced by the current page number when the prompt is issued. The default prompt string is ":".
–c	Home the cursor and clear the screen before displaying each page. This option is ignored if **clear_screen** is not defined for this terminal type in the *terminfo*(4) data base.
–e	Causes *pg* *not* to pause at the end of each file.
–f	Normally, *pg* splits lines longer than the screen width, but some sequences of characters in the text being displayed (e.g., escape sequences for underlining) generate undesirable results. The *–f* option inhibits *pg* from splitting lines.
–n	Normally, commands must be terminated by a <*newline*> character. This option causes an automatic end of command as soon as a command letter is entered.
–s	Causes *pg* to print all messages and prompts in standout mode (usually inverse video).
+linenumber	Start up at *linenumber*.
+/pattern/	Start up at the first line containing the regular expression pattern.

Pg looks in the environment variable *PG* to pre-set any flags desired. For example, if you prefer to view files using the **–c** mode of operation, the shell command sequence *PG='-c' ; export PG* or the *csh* command *setenv PG -c* would cause all invocations of *pg*, including invocations by programs such as *man* and *msgs*, to use this mode. Normally, the user will place the command sequence which sets up the *PG* environment variable in the *.profile* or *.cshrc* file.

The responses that may be typed when *pg* pauses can be divided into three categories: those causing further perusal, those that search, and those that modify the perusal environment.

Commands which cause further perusal normally take a preceding *address*, an optionally signed number indicating the point from which further text should be displayed. This *address* is interpreted in either pages or lines depending on the command. A signed *address* specifies a point relative to the current page or line, and an unsigned *address* specifies an address relative to the beginning of the file. Each command has a default address that is used if none is provided.

The perusal commands and their defaults are as follows:

(+1)<*newline*> or <*blank*>
: This causes one page to be displayed. The address is specified in pages.

(+1) **l**
: With a relative address this causes *pg* to simulate scrolling the screen, forward or backward, the number of lines specified. With an absolute address this command prints a screenful beginning at the specified line.

(+1) **d** or ^**D**
: Simulates scrolling half a screen forward or backward.

The following perusal commands take no *address*.

. or ^**L**
: Typing a single period causes the current page of text to be redisplayed.

$
: Displays the last windowful in the file. Use with caution when the input is a pipe.

The following commands are available for searching for text patterns in the text. The regular expressions described in *ed*(1) are available. They must always be terminated by a <*newline*>, even if the *–n* option is specified.

i/*pattern*/
: Search forward for the *i*th (default *i*=1) occurrence of *pattern*. Searching begins immediately after the current page and continues to the end of the current file, without wrap-around.

i^*pattern*^
i?*pattern*?
: Search backwards for the *i*th (default *i*=1) occurrence of *pattern*. Searching begins immediately before the current page and continues to the beginning of the current file, without wrap-around. The ^ notation is useful for Adds 100 terminals which will not properly handle the ?.

After searching, *pg* will normally display the line found at the top of the screen. This can be modified by appending **m** or **b** to the search command to leave the line found in the middle or at the bottom of the window from now on. The suffix **t** can be used to restore the original situation.

The user of *pg* can modify the environment of perusal with the following commands:

*i***n**
: Begin perusing the *i*th next file in the command line. The *i* is an unsigned number, default value is 1.

*i***p**
: Begin perusing the *i*th previous file in the command line. *i* is an unsigned number, default is 1.

*i***w**
: Display another window of text. If *i* is present, set the window size to *i*.

s *filename*
: Save the input in the named file. Only the current file being perused is saved. The white space between the **s** and *filename* is optional. This command must always be terminated by a <*newline*>, even if the *–n* option is specified.

h
: Help by displaying an abbreviated summary of available commands.

q or **Q** Quit *pg*.

!*command*
: *Command* is passed to the shell, whose name is taken from the **SHELL** environment

variable. If this is not available, the default shell is used. This command must always be terminated by a <*newline*>, even if the *-n* option is specified.

At any time when output is being sent to the terminal, the user can hit the quit key (normally control-\) or the interrupt (break) key. This causes *pg* to stop sending output, and display the prompt. The user may then enter one of the above commands in the normal manner. Unfortunately, some output is lost when this is done, due to the fact that any characters waiting in the terminal's output queue are flushed when the quit signal occurs.

If the standard output is not a terminal, then *pg* acts just like *cat*(1), except that a header is printed before each file (if there is more than one).

EXAMPLE

A sample usage of *pg* in reading system news would be

```
news | pg -p "(Page %d):"
```

NOTES

While waiting for terminal input, *pg* responds to **BREAK**, **DEL**, and ^ by terminating execution. Between prompts, however, these signals interrupt *pg*'s current task and place the user in prompt mode. These should be used with caution when input is being read from a pipe, since an interrupt is likely to terminate the other commands in the pipeline.

Users of Berkeley's *more* will find that the z and f commands are available, and that the terminal /, ^, or ? may be omitted from the searching commands.

FILES

/usr/lib/terminfo/?/*	Terminal information data base
/tmp/pg*	Temporary file when input is from a pipe

SEE ALSO

crypt(1), ed(1), grep(1), terminfo(4).

BUGS

If terminal tabs are not set every eight positions, undesirable results may occur.

When using *pg* as a filter with another command that changes the terminal I/O options (e.g., *crypt*(1)), terminal settings may not be restored correctly.

INTERNATIONAL SUPPORT

8-bit filenames.

NAME

pr – print files

SYNOPSIS

pr [options] [files]

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.

The below *options* may appear singly or be combined in any order:

+*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.
–m	Merge and print all files simultaneously, one per column (overrides the **–***k*, and **–a** options).
–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).
–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).
–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).
–w*k*	Set the width of a line to *k* character positions (default is 72 for equal-width multi-column output, no limit otherwise).
–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.
–l*k*	Set the length of a page to *k* lines (default is 66).
–h	Use the next argument as the header to be printed instead of the file name.
–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).

–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.

–r Print no diagnostic reports on failure to open files.

–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.

–s*c* Separate columns by the single character *c* instead of by the appropriate number of spaces (default for *c* is a tab).

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).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

prealloc – preallocate disk storage

SYNOPSIS

prealloc name size

DESCRIPTION

Prealloc will preallocate at least *size* bytes of disk space for an ordinary file *name* of zero length. It will create the file if it does not already exist. The space will be allocated in an implementation dependent fashion for fast sequential reads and writes for the file.

Prealloc will fail and no disk space will be allocated if *name* already exists and is not an ordinary file of zero length, if there is not enough space left on disk, or if *size* exceeds the maximum file size or the process' file size limit (see *ulimit*(2)). The EOF is left at the end of the preallocated area. The current file pointer is left at zero. The file is zero-filled.

EXAMPLES

The following example preallocates 50000 bytes for the file *myfile*:

```
prealloc myfile 50000
```

DIAGNOSTICS

Upon successful completion, *prealloc* exits with a 0 status. Exit status is 1 if *name* already exists and is not an ordinary file of zero length, 2 if there is not enough room on disk, or 3 if *size* exceeds file size limits.

AUTHOR

Prealloc was developed by the Hewlett-Packard Company.

SEE ALSO

prealloc(2), ulimit(2)

BUGS

The allocation of the file space is highly dependent on the current disk usage. A successful return does not tell you how fragmented the file actually might be if the disk is reaching its capacity.

NAME

prmail – print out mail in the post office

SYNOPSIS

prmail [user ...]

DESCRIPTION

Prmail prints the mail which waits for you, or the specified user, in the post office. The mail is not disturbed.

FILES

/usr/mail/* post office

AUTHOR

Prmail was developed by the University of California, Berkeley.

SEE ALSO

from(1), mail(1).

NAME

prof – display profile data

SYNOPSIS

prof [**–tcan**] [**–ox**] [**–g**] [**–z**] [**–h**] [**–s**] [**–m** mdata] [prog]

DESCRIPTION

Prof interprets a profile file produced by the *monitor*(3C) function. The symbol table in the object file *prog* (**a.out** by default) is read and correlated with a profile file (**mon.out** by default). For each external text symbol the percentage of time spent executing between the address of that symbol and the address of the next is printed, together with the number of times that function was called and the average number of milliseconds per call.

The mutually exclusive options **t, c, a,** and **n** determine the type of sorting of the output lines:

–t Sort by decreasing percentage of total time (default).

–c Sort by decreasing number of calls.

–a Sort by increasing symbol address.

–n Sort lexically by symbol name.

The mutually exclusive options **o** and **x** specify the printing of the address of each symbol monitored:

–o Print each symbol address (in octal) along with the symbol name.

–x Print each symbol address (in hexadecimal) along with the symbol name.

The following options may be used in any combination:

–g Include non-global symbols (static functions).

–z Include all symbols in the profile range (see *monitor*(3C)), even if associated with zero number of calls and zero time.

–h Suppress the heading normally printed on the report. (This is useful if the report is to be processed further.)

–s Print a summary of several of the monitoring parameters and statistics on the standard error output.

–m mdata
Use file *mdata* instead of **mon.out** as the input profile file.

A program creates a profile file if it has been loaded with the **–p** option of *cc*(1). This option to the *cc* command arranges for calls to *monitor*(3C) at the beginning and end of execution. It is the call to *monitor* at the end of execution that causes a profile file to be written. The number of calls to a function is tallied if the **–p** option was used when the file containing the function was compiled.

The name of the file created by a profiled program is controlled by the environment variable PROFDIR. If PROFDIR does not exist, "mon.out" is produced in the directory current when the program terminates. If PROFDIR = string, "string/pid.progname" is produced, where *progname* consists of argv[0] with any path prefix removed, and *pid* is the program's process id. If PROFDIR = nothing, no profiling output is produced.

A single function may be split into subfunctions for profiling by means of the MARK macro (see

prof(5)).

FILES

mon.out for profile
a.out for namelist

SEE ALSO

cc(1), tplot(1), exit(2), profil(2), monitor(3C), prof(5).

WARNING

The times reported in successive identical runs may show variances of 20% or more, because of varying cache-hit ratios due to sharing of the cache with other processes. Even if a program seems to be the only one using the machine, hidden background or asynchronous processes may blur the data. In rare cases, the clock ticks initiating recording of the program counter may "beat" with loops in a program, grossly distorting measurements.

Call counts are always recorded precisely, however.

BUGS

Only programs that call *exit*(2) or return from *main* will cause a profile file to be produced, unless a final call to monitor is explicitly coded.

The use of the **-p** option *cc*(1) to invoke profiling imposes a limit of 600 (300 on the PDP-11) functions that may have call counters established during program execution. For more counters you must call *monitor*(3C) directly. If this limit is exceeded, other data will be overwritten and the **mon.out** file will be corrupted. The number of call counters used will be reported automatically by the *prof* command whenever the number exceeds 5/6 of the maximum.

NAME

prs – print and summarize an SCCS file

SYNOPSIS

prs [**–d**[dataspec]] [**–r**[SID]] [**–e**] [**–l**] [**–c**] [**–a**] files

DESCRIPTION

Prs prints, on the standard output, parts or all of an SCCS file (see *sccsfile*(4)) in a user-supplied format. If a directory is named, *prs* behaves as though each file in the directory were specified as a named file, except that 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 or directory 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** below) 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 or the date given by the **–c** option.

–l Requests information for all deltas created *later* than and including the delta designated via the **–r** keyletter or the date given by the **–c** option.

–c[cutoff] Cutoff date-time, in the form

YY[MM[DD[HH[MM[SS]]]]]

Units omitted from the date-time default to their maximum possible values. Thus, **–c7502** is equivalent to **–c750228235959**. One or more non-numeric characters can be used to separate the various 2-digit segments of the cutoff date (for example **-c77/2/2 9:22:25**).

–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.

If no option letters (or only **–a**) are given, *prs* prints the file name using the default *dataspec* and the **–e** option. This produces information on all deltas.

Data Keywords

Data keywords specify which parts of an SCCS file are to be retrieved and output. All parts of an SCCS file (see *sccsfile*(4)) have an associated data keyword. There is no limit on the number of times a data keyword may appear in a *dataspec*.

The information 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.

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:\nMRs:\n:MR:COMMENTS:\n:C:"

TABLE 1. SCCS Files Data Keywords

Keyword	Data Item	File Sect.	Value	Fmt
:Dt:	Delta information	Delta Tbl	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 # of deltas incl, excl, ign		: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 Nm	text	M
:FL:	Flag list	Flags	text	M
:Y:	Module type flag		text	S
:MF:	MR validation flag		*yes* or *no*	S
:MP:	MR validation pgm name		text	S
:KF:	Keyword error/warning flag		*yes* or *no*	S
:KV:	Keyword validation string		text	S
:BF:	Branch flag		*yes* or *no*	S
:J:	Joint edit flag		*yes* or *no*	S
:LK:	Locked releases		:R:...	S

:Q:	User defined keyword		text	S
:M:	Module name		text	S
:FB:	Floor boundary		:R:	S
:CB:	Ceiling boundary		:R:	S
:Ds:	Default SID		:I:	S
:ND:	Null delta flag		*yes* or *no*	S
:FD:	File descriptive text	Comments	text	M
:BD:	Body	Body	text	M
:GB:	Gotten body		text	M
:W:	A form of *what*(1) string	N/A	:Z::M:\t:I:	S
:A:	A form of *what*(1) string	N/A	:Z::Y: :M: :I::Z:	S
:Z:	*what*(1) string delimiter	N/A	@(#)	S
:F:	SCCS file name	N/A	text	S
:PN:	SCCS file path name	N/A	text	S

* :Dt: = :DT: :I: :D: :T: :P: :DS: :DP:

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 specifications for selecting or printing are 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. The only keyletter argument allowed to be used with the *special case* is the –**a** keyletter.

FILES

/tmp/pr?????

SEE ALSO

admin(1), delta(1), get(1), help(1), sccsfile(4).
Source Code Control System User's Guide in *HP-UX Selected Articles.*

DIAGNOSTICS

Use *help*(1) for explanations.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

ps – report process status

SYNOPSIS

ps [**–edafl**] [**–c** corefile] [**–s** swapdev] [**–n** namelist] [**–t** termlist] [**–p** proclist] [**–u** uidlist] [**–g** grplist]

DESCRIPTION

Ps prints certain information about active processes. Without *options*, information is printed about processes associated with the current terminal. The output consists of a short listing containing only the process ID, terminal identifier, cumulative execution time, and the command name. Otherwise, the information that is displayed is controlled by the selection of *options*.

Options using lists as arguments can have the list specified in one of two forms: a list of identifiers separated from one another by a comma, or a list of identifiers enclosed in double quotes and separated from one another by a comma and/or one or more spaces.

The *options* are:

–e Print information about all processes.

–d Print information about all processes, except process group leaders.

–a Print information about all processes, except process group leaders and processes not associated with a terminal.

–f Generate a *full* listing. (See below for meaning of columns in a full listing.)

–l Generate a *long* listing. See below.

–c *corefile* Use the file *corefile* in place of **/dev/mem**.

–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.

–n *namelist* The argument will be taken as the name of an alternate system *namelist* file in place of **/hp-ux**.

–t *termlist* Restrict listing to data about the processes associated with the terminals given in *termlist*. Terminal identifiers may be specified in one of two forms: the device's file name (e.g., **tty04**) or if the device's file name starts with **tty**, just the digit identifier (e.g., **04**).

–p *proclist* Restrict listing to data about processes whose process ID numbers are given in *proclist*.

–u *uidlist* Restrict listing to data about processes whose user ID numbers or login names are given in *uidlist*. In the listing, the numerical user ID will be printed unless the **–f** option is used, in which case the login name will be printed.

–g *grplist* Restrict listing to data about processes whose process group leaders are given in *grplist*.

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 determine only 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:

0	swapped;
1	in core;
2	system process;
4	locked in core (e.g., for physical I/O);
10	being swapped;
20	being traced by another process;
40	another tracing flag;

S (l) The state of the process:

0 non-existent;
S sleeping;
W waiting;
R running;
I intermediate;
Z terminated;
T stopped;
X growing.

UID (f,l) The user ID number of the process owner; the login name is printed under the **–f** option.

PID (all) The process ID of the process; it is possible to kill a process if you know this datum.

PPID (f,l) The process ID of the parent process.

C (f,l) Processor utilization for scheduling.

PRI (l) The priority of the process; higher numbers mean lower priority.

NI (l) Nice value; used in priority computation.

ADDR (l) The memory address of the 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.

STIME (f) Starting time of the process.

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.

To make *ps* output safer to display and easier to read, all control characters in the CMD field are mapped to visible equivalents. These are of the form ^*C* where the original character was in the range 0 - 037 and **c** is that value plus 040.

HARDWARE DEPENDENCIES

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.
In the **S** field, the **L** means waiting on a file lock via *lockf*(2) .
In the **S** field, the **B** (blocked) state means blocked via an IPC system call such as *semop*(2), *msgrcv*(2), or *msgsnd*(2) .
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**, **n**, and **c** options are not currently supported. A diagnostic is printed if they are used.

FILES

/hp-ux	system namelist
/dev/mem	memory
/dev/swap	the default swap device
/etc/passwd	supplies UID information
/etc/ps_data	internal data structure
/dev	searched to find terminal ("tty") names

SEE ALSO

acctcom(1), 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.

INTERNATIONAL SUPPORT

8- and 16-bit data.

NAME

psqlc, psqlpas, psqlfor - ALLBASE/HP-UX preprocessors for C, Pascal and FORTRAN

SYNOPSIS

psqlc **-s** [**-i** sourcefilename.sql] [**-p** sqloutfilename.f]

psqlc DBEnvirnomentName [**-o** ownername] [**-m** modulename] [**-d** [**-r**]] [**-i** sourcefilename.sql] [**-p** sqloutfilename.f]

psqlpas **-s** [**-i** sourcefilename.sql] [**-p** sqloutfilename.f]

psqlpas DBEnvirnomentName [**-o** ownername] [**-m** modulename] [**-d** [**-r**]] [**-i** sourcefilename.sql] [**-p** sqloutfilename.f]

psqlfor **-s** [**-i** sourcefilename.sql] [**-p** sqloutfilename.f]

psqlfor DBEnvirnomentName [**-o** ownername] [**-m** modulename] [**-d** [**-r**]] [**-i** sourcefilename.sql] [**-p** sqloutfilename.f]

REMARKS

The ALLBASE/HP-UX product must be previously installed on the system for *psqlc*, *psqlpas*, or *psqlfor* to function.

DESCRIPTION

Psqlc, *psqlpas* and *psqlfor* invoke the C, Pascal and FORTRAN preprocessors, respectively, for programmatically accessing an ALLBASE/HP-UX relational DataBase Environment (DBEnvironment). *Psqlc*, *psqlpas* and *psqlfor* can be executed by all system users.

Options

-s Specifies that the preprocessor is only to check the syntax of embedded SQL commands.

DBEnvironmentName
Identifies the DBEnvironment in which a module is to be stored.

-o *ownername* Associates the stored module with a user's login name, a classname, or a groupname. You can specify an *ownername* for the module only if you have DBA authority in the DBEnvironment where the module is to be stored. If not specified, the default *ownername* is your login name.

-m *modulename*
Assigns a name to the stored module. *Modulenames* must follow the rules governing HPSQL basic names as described in the ALLBASE/HP-UX SQL Reference Manual. If a *modulename* is not specified, the preprocessor uses the PROGRAM Statement name as the *modulename*.

-d (DROP) Deletes any module currently stored in the DBEnvironment by the *modulename* and *ownername* specified in the options list. If not specified, NODROP is assumed and all existing RUN authorities for that module are preserved.

-r (REVOKE) Is specified when the program being preprocessed already has a stored module and you want to revoke all existing RUN authorities for that module. REVOKE cannot be specified unless DROP is also specified. If the **-r** (REVOKE) option is not specified, it is assumed that all existing RUN authorities for that module are to be PRESERVED.

-i *sourcefilename.sql*
Identifies the name of the input file containing the source code to be preprocessed. If *sourcefilename.sql* is not specified, a file by the name of **sqlin** is

assumed. It is recommended that the source file name have a file extension of ".sql"; however, this is not required.

-p *sqloutfilename.f*

Identifies the name of the output file containing the preprocessor generated code. If *sqloutfilename.f* is not specified, the preprocessor generated code is written to a file with the same name as the source file name but with an appended file extension of ".f".

AUTHOR

Psqlc, *psqlpas*, and *psqlfor* were developed by Hewlett-Packard.

FILES

/usr/bin/hpdbdaemon	cleanup daemon program file
/usr/bin/psqlc	C preprocessor program file
/usr/bin/psqlpas	Pascal preprocessor program file
/usr/bin/psqlfor	FORTRAN preprocessor program file
/usr/lib/hpsqlproc	HP SQL program file
/usr/bin/isql	Interactive SQL program file
/usr/bin/sqlutil	SQLUTIL program file
/usr/lib/hpsqlcat	HP SQL message catalog file
/usr/lib/isqlwel	Interactive SQL welcome banner file
/usr/lib/libsql.a	run time routine library file

SEE ALSO

ALLBASE/HP-UX HPSQL C Application Programming Guide.

ALLBASE/HP-UX HPSQL Pascal Application Programming Guide.

ALLBASE/HP-UX HPSQL FORTRAN Application Programming Guide.

NAME

ptx – permuted index

SYNOPSIS

ptx [options] [input [output]]

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* macro provided by the user, or provided by the *mptx* 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 by using a line length of 100.

–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. Punctuation characters are treated as part of the word in the absence of this option.

–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

/usr/lib/eign
/bin/sort
/usr/lib/tmac/tmac.ptx

SEE ALSO

nroff(1), mm(5).

BUGS

Line length counts do not account for overstriking or proportional spacing.
Lines that contain tildes ( ) are botched, because *ptx* uses that character internally.

INTERNATIONAL SUPPORT

8-bit data and filenames.

NAME

pwd – working directory name

SYNOPSIS

pwd

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 system manager.

INTERNATIONAL SUPPORT

8-bit filenames, messages.

NAME

query – interactive IMAGE database access

SYNOPSIS

query

Remarks:

Query is implemented on the Series 500 only, and is not included in the standard HP-UX operating system. Optional IMAGE software must be installed on the system before *query* can be used.

DESCRIPTION

Query is an interactive, command driven program to simplify IMAGE database access. It can be used to generate reports from the database, add information to the database, change information in the database, and aid in developing programs that access databases using IMAGE library subroutines.

Consistent with the HP-UX environment in which it operates, *query* is initiated by simply typing its name. There are no options or parameters. Input and output redirection can be done at the shell level (< >) although more convenient methods are available via *query* commands.

A list of the available commands:

data-base=	help	exit	!
list	form	find	xeq
update a	update d	update r	report

Query accepts these commands in upper- or lower-case. Special care must be taken in the case of set names, item names, and item values since these are case sensitive. That is, Setname, setname, and SETNAME are three unique sets.

All *query* commands must be followed by a semicolon. *Query* waits silently for a semicolon or a zero-length record before processing a command. A zero-length record is entered as a solitary carriage return. This method of signaling the end of a command line enables you to enter commands which are several lines long. Line length is limited to 256 characters. At any point in a line, you may type a carriage return and continue the command line, thus improving the readability of long command lines.

Once initiated, *query* identifies itself and gives the prompt:

NEXT?

Whenever this prompt appears, you may enter any of the *query* commands, which are described below.

DATA-BASE=

The **data-base=** command opens a database. You can type:

data-base=data_base_name;

where *data_base_name* is the name of a database. If you are presently in the directory where the database exists, you need only give the database name. If the database is in another directory, you need to supply a partial or complete path name.

Some examples are:

data-base=/users/fred/inventory;

specifies a database called "inventory" in the directory /users/fred.

data-base=equipment;

specifies a database called "equipment" in the current directory.

As part of the **data-base=** command, *query* asks for a password for that database with the prompt:

PASSWORD?

The password is not echoed on the terminal as you type it. As usual, the password must be followed by a semicolon. If no password is required, simply press RETURN.

Provided the database name and password are valid, the database is opened with "modify shared" access (DBOPEN mode 1). The command prompt "NEXT?" appears.

HELP

The **help** command provides a syntax model, a brief description, and examples of itself or any other *query* command. It can be invoked in the following ways:

```
help [ command ];
   or
? [ command ];
```

where *command* is any *query* command. If no command is supplied, **help** describes itself and gives a list of the commands for which help is available.

EXIT The **exit** command provides you with a graceful way to terminate the *query* program. It is entered thus:

```
exit;
```

Query can also be terminated by hitting the BREAK key in response to a command prompt.

!

At any time, in response to a "NEXT?" prompt, you may wish to execute a shell command without leaving the *query* program. This is useful when debugging report procedure files from within *query*, or routing output files to a printer during a *query* session. For example:

```
!pr filename | lpr
```

runs the formatter/printer *pr* on a file called *filename*, and pipes the output into the *lpr* program.

A shell command following an exclamation mark is executed, and *query* is suspended until that command is completed. *Query* processing can be continued when the "[Hit return to continue QUERY]" message appears.

LIST The **list** command is a convenient method of redirecting output from within the *query* program. *Query* sends output to stdout (the input device) by default. To send output elsewhere, type:

```
list=filename;
```

Output is sent to *filename* in the current directory. It may be sent to a file in another directory by specifying the desired pathname.

An example is:

```
list=/usr/spool/uucppublic/report;
```

which specifies that the output file is */usr/spool/uucppublic/report.*

If *query* finds that the named file already exists, it prints the message:

```
FILE ALREADY EXISTS. (O)VERWRITE IT, (A)PPEND TO IT,
OR (N)EITHER ?
```

You type **o**, **a**, or **n** to select an option. If you choose **n**, *query* prompts:

```
NEW FILE NAME=
```

in response to which you provide a (presumably) different file name. Otherwise, *query* overwrites or appends to the selected file, as instructed. Output directed to a file is properly formatted for direct submission to *lpr*(1). At any time during a *query* session, you may

return output to the terminal by typing:

list;

This can be repeated as often as necessary, using the same file or many different files for output. When the **list** command appears in an XEQ file, no choices are offered. The specified file is silently opened and any output is appended to it.

FORM

The **form** command outputs a *schema* description for the open database. It lists each data set name, its type (automatic master, manual master, or detail), the set capacity, and the current number of entries in the set. With each data set, each item is listed including its name, type (alphanumeric, integer, or real), length in bytes, and number of elements in the item. *Query* also identifies key items, sort items (the sort item name may be truncated), and indicates whether you have write access for the item. It is initiated by typing:

form;

Form's output is directed to the file specified by the **list** command, or stdout by default. Its output can be terminated by hitting the BREAK key. Within a few seconds, output stops and a command prompt (NEXT?) appears.

FIND

A major use of any *query* program is to search a database for an arbitrary group of entries meeting some criteria. **Find** is used in conjunction with the **update** or **report** commands, providing "victims" for the update or report. It is entered by typing:

find retrieve_procedure end;

where *retrieve_procedure* is a group of data item names, data item values, and relational operators joined together by logical connectors.

A retrieve procedure defines a relationship between a data item and a data item value, and the **find** command collects entries which satisfy that relationship for later use by **update** or **report**. A typical retrieve procedure looks like this:

[setname.]itemname operator "value"

where *setname* is the name of a data set which contains the data item. It is always accepted, but is not necessary when the item name exists in only one data set, or when the set name has been previously established in the retrieve procedure. *Itemname* is simply the name of a data item. For compound items, only the first element is used. *Operator* is one of the following relational operators:

is, ie	equal to
isnot, ine	not equal to
ilt	less than
inlt	not less than
igt	greater than
ingt	not greater than

Value is enclosed in quotation marks (" ") and is compared to the value of the named item for each entry in the specified (or implied) set. It should be appropriate for the data item type.

Two or more retrieve procedures can be joined by the logical operators **and** and **or** to make a more complex procedure. Parentheses are not allowed in **find** procedures, so care should be taken in ordering statements in a compound retrieve procedure.

Some examples are:

find inventory.quantity is "324" end;

searches all entries in the "inventory" set for a "quantity" equal to "324". This would be appropriate for a quantity of any type (alphanumeric, integer, or real).

find part_description ie "widgit" end;

searches all entries in the set which contains the item "part_description" for a value of "widgit". The value would obviously be inappropriate for an item type of integer or real. This example generates an error if "part_description" exists in more than one set in the database.

find inventory.quantity igt "324" and part_description isnot "widgit" end;

searches all entries in the set "inventory", collecting those that show a quantity greater than 324, excluding widgits. The items "quantity" and "part_description" must both be items contained in the set "inventory".

XEQ The **xeq** command allows any number of commands to be read from a file created by any of the HP-UX editors. Commands must appear in the file exactly as they would be entered interactively, one command per line. The **xeq** command is entered:

xeq=filename;

The *filename* may be an absolute pathname, if necessary. *Query* reads commands from that file until it encounters either an end-of-file or another **xeq** command. When end-of-file is reached, *query* returns to an interactive state. When an **xeq** command is encountered within an **xeq** file, *query* closes the current **xeq** file and begins reading commands from the new one. The old one is not re-opened. **Xeq** files can be nested up to 10 deep.

A few commands behave differently when they occur in an **xeq** file. The "**list**=file" command silently opens the specified file and appends data to it. The update mode of the **update r** command can be terminated only by a lone semicolon. (In interactive use, **update r** can be terminated by a semicolon or a zero-length record.)

UPDATE A

Update a (add) adds entries to a data set. It is the only update which does not require a preceding **find**. The **update a**dd command is entered:

update a, setname;
or
update add, setname;

Query checks the validity of the set name, and then prompts for item values one at a time. The item name is displayed followed by an "=". The value to be assigned to that item should then be entered, enclosed in quotes, and followed by a semicolon. *Query* then prompts for the next item value. Only one prompt is given for compound data items; the item values should be entered each in quotes, separated by commas. Null values may be entered by a lone carriage return in response to a prompt, but *query* insists on valid values for key items. Addition of detail entries requires that values for key items already exist in the corresponding master set(s).

The BREAK key can be used to abort an **update a** command. No update takes place, and a command prompt appears.

UPDATE D

Update d (delete) deletes entries from a data set. It requires a preceding **find**, and complains if not satisfied. The command is entered:

update d;
or
update delete;

As a safety check, *query* asks "OK TO DELETE?(YES/NO)". Upon receipt of a **y** or **n**, *query* proceeds as directed. It refuses to delete master set entries which contain chain heads with non-empty chains (i.e. connected detail sets), and displays a message to that effect.

UPDATE R

Update r (replace) also must be preceded by a **find**. It is a means of changing item values in an existing entry. It is invoked by typing:

```
update r;
   or
update replace;
```

Query responds with the prompt "ITEM =". You then enter:

```
item_name="value";
```

where *item_name* is an item name which exists in the entries in the select file. *Value* is a value appropriate to the item type (alpha, integer, or real) enclosed in double quotes.

When you have finished entering the changes desired, a lone carriage return or a lone semicolon exits this update mode, and *query* executes the changes and returns to the command level. (In an **xeq** file, only the semicolon suffices.) The new value(s) are inserted into all entries collected in the select file. Updates are refused for key items and sort items in master sets. Updating key or sort items in detail sets causes that record to be deleted and reentered with the new values. A report following such an update may give an "EMPTY RECORD" error message. Don't panic. The record may be found at its new location by a **find** command.

The BREAK key can be used to abort an **update r** command. No update takes place, and a command prompt appears.

REPORT

The **report** command provides many features to display information about the entries in the select file. The information is sent to the list device (the input device, by default). **Report**'s output can be terminated by using the BREAK key, which yields a command prompt (NEXT?). You can request the name and value of each data item for all the data entries specified in the select file, or request the data item values for all of the data entries without printing the data item name. Also you may create output formats complete with page headings, page numbers, column headings, space and page control, and selectivity in item value display. **Report** can be invoked in one of three ways:

```
report all [,character];
   or
report name=procedure_name;
   or
report;
body
end;
```

where *character* is any ASCII printing character which determines the printing of certain optional information.

Procedure_name is the name of a file (specified as a relative or absolute pathname) which contains **report** commands stored via a system editor, such as *ed* or *vi*.

Body consists of header, detail, edit, and sort commands as outlined below.

The three forms of the report command are described below.

REPORT ALL [,CHARACTER];
prints the entire data item and all elements of a compound item. This report form prints the item name, followed by "=", followed by the item value. The optional *character* causes *query* to print only the item value, without the item name and "=". All item values appear left justified, and numbers are stripped of insignificant zeros. Real numbers may appear in decimal form or scientific notation, as necessary. *This is the only report form which shows all values for compound data items.*

REPORT NAME=PROCEDURE_NAME;
gets header, detail, edit, and sort commands from a file, reading commands until an "end;" or an error is encountered. The contents of a procedure file are identical to the "body" in the next form of the **report** command. It should be noted that the use of the shell escape (!) is a valuable aid in the development of procedure files. It enables you to invoke an editor, modify a file, exit it, and return to the same point in *query* to test the file, without having to re-define the database or re-establish a select file.

REPORT;
BODY
END; accepts **report** commands from the user, scanning each line as it is entered for syntax errors. The entry of an "end;" command initiates the execution of the commands, producing a report. The body is a collection of the following commands:

Header	Prints title, column headings, and page numbers at the top of each report page.
Detail	Prints data item values in the column position specified.
Edit	Describes the number of decimal places to be displayed for real numbers.
Sort	Sorts data entries based upon the value of a specified data item.

Report Formatting

The above commands can be formatted using the following parameters. (Note: these are *parameters* to the **report** commands, not commands themselves.)

print position
Specifies the ending column for an item value or heading.

space and space control
Causes line skips between item values or heading lines.

skip and skip control
Causes page skips between item values or heading lines.

edit
Specifies edit commands to which output item values should conform.

These parameters are described below.

print position
This parameter is an integer between 1 and 132 which indicates the column number in which the last character of an item value should appear in a header or detail line. It is your responsibility to avoid overlap between fields on the same line. However, in most cases *query* replaces an overlapping value with asterisks to indicate an error.

space and space control
This parameter outputs blank lines either before or after the printing of a header string or detail line value. The keyword **space** should be followed by either an **a** or **b**, indicating where the blank line should appear – after or before the line to be

printed. The **a** or **b** may be followed by an integer in the range 1 - 5, to skip multiple lines. Absence of the integer causes *query* to skip 1 line. These may appear more than once in a command, as in spacing before and after a line:

```
h1,"page",35,space b2,space a3;
```

This generates two blank lines before printing "page", and three blank lines afterward.

skip and skip control

Similar to **space**, **skip** yields page feeds either before or after the printing of a line. Unlike **space**, **skip** can only be used with a "detail" command. The keyword **skip** is followed by an **a** or **b** to direct where the page feed should be placed (see "space and space control" above). Normally, *query* prints 54 lines per page before skipping to a new page.

edit

The **edit** parameter is the letter **e** followed by an integer in the range 0 - 9. This number corresponds to a numbered edit command which specifies the number of decimal places (for real numbers) or the number of characters (for alphanumeric strings) to be printed.

Report Commands

h (header command)

The **header** command is used to print heading information of your choice at the top of each page of the report. A maximum of five lines of header information can appear at the top of each report page. The format of the header command is:

hnumber,data_type,print_position[,space space-control];

where *number* is an integer from 1 to 5 specifying on which header line (out of five possible lines) the information is to appear. Header information in a header command labeled "h1" appears in the first line, "h2" appears in the second line, etc.

Data_type is either an ASCII character string enclosed in double quotes, or the word **pageno** (without quotes). If **pageno** appears in the header command, *query* prints the page number of the report in the position specified by *print_position*. *Query* increments the page number automatically for each page printed.

Print_position, *space*, and *space-control* are parameters defined in the section on report formatting.

An example is:

```
h1,"PAGE",70,space b2;
h1,pageno,76;
h2,"DAILY REPORT",50,space a3;
```

which prints the word "PAGE" with the letter "E" in column 70, on the second line from the top of a page (via the "space b2" parameter). On the same line, the page number is printed ending in column 76. The next line contains "DAILY REPORT" ending in column 50, followed by three blank lines.

d (detail command)

The **detail** command indicates which data items of a data entry specified in the select file are to be printed in the report. Data items can be printed on up to 10 different lines. *Query* prints only the values of data items which appear in a detail command.

If an ASCII value length exceeds the distance between a preceding value on the same line or the left margin, it is silently truncated on the left. If a numeric

value overlaps in the same manner, it is replaced by a series of asterisks, indicating the error.

Detail commands without an edit parameter print numeric values in whatever format necessary to give maximum accuracy.

The format of a detail command is:

d[n],data_type,print_position[,space space-control][,skip skip-control][,edit];

where *n* is an integer from 1 to 9. Each number specifies a different line on which the data items are printed. If the number is omitted, the unnumbered detail item is printed on a separate line above any numbered detail item lines. The lowest numbered command is printed first and all others follow in numeric order. Detail commands with the same number are printed on the same line.

Data_type is either an ASCII character string enclosed in double quotes, or the name of a data item contained in the data entries specified in the select file. NOTE: the **report** command processor expects the data item name *by itself.* Preceding the item name with a set name generates an error.

Print_position, *space*, *space-control*, *skip*, *skip-control*, and *edit* are parameters defined in the section on report formatting.

e (edit command)

The **edit** command is used to format the printing of real and/or alphanumeric item values. Up to ten edit commands, labeled from e0 to e9, can be used in a report. To edit output from a detail command, you include the label of the desired edit command. *Query* refers to the labeled edit command to edit the value printed by the detail command. The same edit command can be referenced by more than one detail command in the same report, and each edit command must be referenced at least once in the report body. The format of the edit command is:

enumber,"places,format";

where *number* is an integer from 0 to 9, identifying the edit command. An integer cannot be used to identify more than one edit command.

Places is an integer indicating the number of digits to follow the decimal point (for real numbers) or the number of characters to be printed (for alphanumeric strings).

Format is one of the following single letters:

- **f** indicating that the number should be formatted in decimal form, with the specified number of digits following the decimal point. Numbers accurate to more than the specified number of places are rounded.
- **e** indicating that the number should be formatted in scientific notation with the specified number of digits following the decimal point.
- **s** indicating that the data item is an alphanumeric string. The number of characters printed is specified by the accompanying integer. If you specify 10 characters for a data item 40 characters in length, the leftmost 10 characters are printed. If you specify 100 characters for the same data item, only 40 characters are printed.

Here are some examples:

```
e1,"6f";
d1,real_number,40,e1;
```

might yield such numbers as:

```
2.340000
25487.123456
1.000000
```

and

```
e1,"4e";
d1,realnum,40,e1;
```

might yield

```
2.3400e+0
3.2549e+5
1.0000e+0
```

and

```
e1,"15s";
d1,String,40,e1;
```

might yield

```
Smith, Jame
truncate ri
Walker, Mau
Doe, John
```

Finally, the difference between the edit *command* and the edit *parameter* should be emphasized. For example,

```
e1,"7f";
d1,Any_real,30,e1;
end;
```

The first line is an edit command, specifying a format for real numbers. In the second line, the "e1" is a parameter, indicating that the real number(s) "Any_real" should be printed according to the format shown in the "e1" command.

s (sort command)

The **sort** command specifies an item upon which you want the entries in the select file sorted. The format of the sort command is:

```
s,itemname;
```

where *itemname* is the name of a data item which appears in entries currently stored in the select file.

A sort item value may not exceed 80 bytes in length. In the case of a compound data item, sort uses only the first value in that item. After a "find", the entries appear in the select file in the order the find command encounters them. The sort command will rearrange those entries in ascending alphabetic or numeric order, depending on the sort item.

Report Example

Assuming that *Emp_name* is a 20-byte alphanumeric item, *Emp_age* is a 2-byte integer, and *Emp_wage* is a 4-byte real:

```
H1,"EMPLOYEE REPORT",34,space b3;
h1,"PAGE",52;
h1,pageno,56,space a2;
h2,"NAME",7;
H2,"AGE",27;
```

```
H2,"HOURLY WAGE",52;
e1,"2f";
s,Emp_name;
d1,Emp_name,20;
d1,Emp_age,27;
d1,"$",44;
d1,Emp_wage,50,e1,space a;
end;
```

This report might yield:

EMPLOYEE REPORT PAGE 1

NAME	AGE	HOURLY WAGE
Anderson,Richard	32	$ 14.75
Carr,Elaine	21	$ 11.50
Wilson,Kathy	42	$ 17.25

.
.
.

Summary

Although the commands appear throughout this document in lower-case, *query* accepts them in upper-case also. This is helpful when working with databases ported from Series 500 BASIC, in which database names, data set names, and data item names are frequently in upper-case.

It should be stressed that all commands must end with a semicolon or zero-length entry. If *query* seems to have "gone away", be sure that a semicolon followed tha last command entered. If this is not the case, an extra carriage return serves to terminate the command and prompt *query* into action.

Query sometimes appears to be "eating" report commands and doing nothing about them, other than supplying the "NEXT?" prompt. This is usually the result of having used the "list" command to re-route output earlier in the session, and having forgotten about it. Typing

```
list;
```

re-routes output to the terminal again.

Abnormal termination of *query* leaves files in */usr/tmp*. It is your responsibility to remove these files or they may accumulate and use up large amounts of memory. The files can be identified by the owner id, shown by typing "ll /usr/tmp". Do not attempt to remove files belonging to anyone else.

FILES

/usr/bin/query	
/usr/bin/querysort	*query*'s own sort routine
/usr/lib/query.help	help file
/usr/tmp/*	temporary files

NAME

ratfor – rational Fortran dialect

SYNOPSIS

ratfor [options] [files]

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 per line and automatic continuation of lines

comments:

this is a comment.

compiler directives:

directives beginning with a dollar sign (**$**) in column one are passed through to the compiler unchanged.

translation of relationals:

>, >=, etc., become **.GT.**, **.GE.**, etc.

return expression to caller from function:

return (expression)

define:

define *name replacement*

include:

include *file*

Options are as follows:

-h causes quoted strings to be turned into Hollerith constructs such as, for example, **27H**.

-C copies comments to the output and attempts to format it neatly.

-6*c* normally, continuation lines are marked with an **&** in column 1. The option **-6*c*** makes the continuation character *c* and places it in column 6.

Ratfor is best used with *f77*(1).

HARDWARE DEPENDENCIES

Series 200, 300:

Options may be passed to *ratfor* through *f77*(1) by using the **-W** option specifier.

Series 500:

Fc (on *f77*(1)) does not recognize **ratfor.r** files. Therefore, *ratfor* must be called directly.

The **-h** option should not be used.

The **-6x** option must be used for successful automatic continuation.

SEE ALSO

f77(1).

B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, 1976.

NAME

rev – reverse lines of a file

SYNOPSIS

rev [file] ...

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.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

Series 500 Only

NAME

revision – get HP-UX revision information

SYNOPSIS

/usr/bin/revision

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:    05.11
Version:    B
Machine:    9050X
Identity:   HP-UX  NNNNANNNNN
Nodename:   hpfcla
```

SEE ALSO

uname(2).

NAME

rm, rmdir – remove files or directories

SYNOPSIS

rm [**–fri**] file ...
rmdir dir ...

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 when the **–f** option is given or 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. In that case, *rm* recursively deletes the entire contents of the specified directory, and the directory itself. (Note that *rm* can recursively remove a maximum of 17 directory levels.)

If the **–i** (interactive) option is in effect, *rm* asks whether to delete each file, and, under **–r**, whether to examine each directory.

Rmdir removes entries for the named directories, which must be empty.

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 .*
```

INTERNATIONAL SUPPORT

rm: 8- and 16-bit data, 8-bit filenames
rmdir: 8- and 16-bit data, 8-bit filenames, messages.

NAME

rmdel – remove a delta from an SCCS file

SYNOPSIS

rmdel –rSID files

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 *Source Code Control System 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(4).
Source Code Control System User's Guide in *HP-UX Selected Articles.*

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

rmnl – remove extra new-line characters from file

SYNOPSIS

rmnl

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, effectively removing all blank lines. This is used by the *man* command.

Ssp(1) can be used to remove redundant blank lines, rather than all blank lines.

SEE ALSO

man(1), ssp(1).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

rtprio – execute process with realtime priority

SYNOPSIS

rtprio priority command [arguments]
rtprio priority -pid
rtprio -t command [arguments]
rtprio -t -pid

DESCRIPTION

Rtprio executes *command* with a realtime priority, or changes the realtime priority of currently executing process *pid*. Realtime priorities range from zero (highest) to 127 (lowest). Realtime processes are not subject to priority degradation and are all of greater (scheduling) importance than non–realtime processes. See *rtprio*(2) for more details.

If **-t** is specified instead of a realtime *priority* then *rtprio* executes *command* with a timeshare (non-realtime) priority, or changes the currently executing process *pid* from a possibly realtime priority to a timeshare priority. The former is useful to spawn a timeshare priority command from a realtime priority shell.

If -t is not specified, *command* will not be scheduled, or *pid*'s realtime priority will not be changed, if the user is not a member of a group having PRIV_RTPRIO access and is not the super–user. When changing the realtime priority of a currently executing process, the effective user ID of the calling process must be superuser, or the real or effective user ID must match the real or saved user ID of the process to be modified.

EXAMPLES

The following example executes the a.out file at a real-time priority of 100:

```
rtprio 100 a.out
```

The following example sets the currently running process with pid 24217 to a real-time priority of 40:

```
rtprio 40 -24217
```

AUTHOR

Rtprio was developed by the Hewlett-Packard Company.

SEE ALSO

getprivgrp(2), rtprio(2).

RETURNS

Rtprio returns exit status 0 if *command* is successfully scheduled or if *pid*'s realtime priority is successfully changed, 1 if *command* is not executable or *pid* does not exist, and 2 if *command* (*pid*) lacks realtime capability, or the invoker's effective user ID is not superuser, or the real or effective user ID does not match the real or saved user ID of the process to be changed.

NAME

sact – print current SCCS file editing activity

SYNOPSIS

sact files

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.

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* for editing).
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).

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

sccsdiff – compare two versions of an SCCS file

SYNOPSIS

sccsdiff –rSID1 –rSID2 [–**p**] [–**s***n*] files

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.

FILES

/tmp/get????? Temporary files

SEE ALSO

bdiff(1), diff(1), get(1), help(1), pr(1).
Source Code Control System User's Guide in *HP-UX: Selected Articles.*

DIAGNOSTICS

"*file*: No differences" if the two versions are the same.

Use *help*(1) for explanations.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

sdb – symbolic debugger

SYNOPSIS

sdb [–w] [–W] [objfil [corfil [directory-list]]]

DESCRIPTION

Sdb is a symbolic debugger that can be used with C and F77 programs. It may be used to examine their object files and core files and to provide a controlled environment for their execution.

Objfil is normally an executable program file which has been compiled with the **–g** (debug) option; if it has not been compiled with the **–g** option, or if it is not an executable file, the symbolic capabilities of *sdb* will be limited, but the file can still be examined and the program debugged. 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**. The core file need not be present. A – in place of *corfil* will force *sdb* to ignore any core image file. The colon separated list of directories (*directory-list*) is used to locate the source files used to build *objfil*.

It is useful to know that at any time there is a *current line* and *current file*. If *corfil* exists then they are initially set to the line and file containing the source statement at which the process terminated. Otherwise, they are set to the first line in *main()*. The current line and file may be changed with the source file examination commands.

By default, warnings are provided if the source files used in producing *objfil* cannot be found, or are newer than *objfil*. This checking feature and the accompanying warnings may be disabled by the use of the **–W** flag.

Names of variables are written just as they are in C or F77. Note that names in C are now of arbitrary length, *sdb* will no longer truncate names. Variables local to a procedure may be accessed using the form *procedure:variable*. If no procedure name is given, the procedure containing the current line is used by default.

It is also possible to refer to structure members as *variable.member*, pointers to structure members as *variable–>member* and array elements as *variable*[*number*]. Pointers may be dereferenced by using the form *pointer*[0]. Combinations of these forms may also be used. F77 common variables may be referenced by using the name of the common block instead of the structure name. Blank common variables may be named by the form *.variable*. A number may be used in place of a structure variable name, in which case the number is viewed as the address of the structure, and the template used for the structure is that of the last structure referenced by *sdb*. An unqualified structure variable may also be used with various commands. Generally, *sdb* will interpret a structure as a set of variables. Thus, *sdb* will display the values of all the elements of a structure when it is requested to display a structure. An exception to this interpretation occurs when displaying variable addresses. An entire structure does have an address, and it is this value *sdb* displays, not the addresses of individual elements.

Elements of a multidimensional array may be referenced as *variable*[*number*][*number*]..., or as *variable*[*number,number,...*]. In place of *number*, the form *number;number* may be used to indicate a range of values, * may be used to indicate all legitimate values for that subscript, or subscripts may be omitted entirely if they are the last subscripts and the full range of values is desired. As with structures, *sdb* displays all the values of an array or of the section of an array if trailing subscripts are omitted. It displays only the address of the array itself or of the section specified by the user if subscripts are omitted. A multidimensional parameter in an F77 program cannot be displayed as an array, but it is actually a pointer, whose value is the location of the array. The array itself can be accessed symbolically from the calling function.

A particular instance of a variable on the stack may be referenced by using the form *procedure:variable,number*. All the variations mentioned in naming variables may be used. *Number* is the occurrence of the specified procedure on the stack, counting the top, or most

current, as the first. If no procedure is specified, the procedure currently executing is used by default.

It is also possible to specify a variable by its address. All forms of integer constants which are valid in C may be used, so that addresses may be input in decimal, octal or hexadecimal.

Line numbers in the source program are referred to as *file-name:number* or *procedure:number*. In either case the number is relative to the beginning of the file. If no procedure or file name is given, the current file is used by default. If no number is given, the first line of the named procedure or file is used.

While a process is running under *sdb*, all addresses refer to the executing program; otherwise they refer to *objfil* or *corfil*. An initial argument of **-w** permits overwriting locations in *objfil*.

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. In some cases (e.g., for programs with separated I and D space) the two segments for a file may overlap.

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 *sdb* to be used on large files, all appropriate values are kept as signed 32-bit integers.

Commands

The commands for examining data in the program are:

t Print a stack trace of the terminated or halted program.

T Print the top line of the stack trace.

variable/*clm*

Print the value of *variable* according to length *l* and format *m*. A numeric count *c* indicates that a region of memory, beginning at the address implied by *variable*, is to be displayed. The length specifiers are:

b	one byte
h	two bytes (half word)
l	four bytes (long word)

Legal values for *m* are:

c	character
d	decimal
u	decimal, unsigned
o	octal
x	hexadecimal
f	32-bit single precision floating point
g	64-bit double precision floating point

s Assume *variable* is a string pointer and print characters starting at the address pointed to by the variable.

a Print characters starting at the variable's address. This format may not be used with register variables.

p pointer to procedure

i disassemble machine-language instruction with addresses printed numerically and symbolically.

I disassemble machine-language instruction with addresses just printed numerically.

The length specifiers are only effective with the formats **c**, **d**, **u**, **o** and **x**. Any of the specifiers, *c*, *l*, and *m*, may be omitted. If all are omitted, *sdb* choses a length and a format suitable for the variable's type as declared in the program. If *m* is specified, then this format is used for displaying the variable. A length specifier determines the output length of the value to be displayed, sometimes resulting in truncation. A count specifier *c* tells *sdb* to display that many units of memory, beginning at the address of *variable*. The number of bytes in one such unit of memory is determined by the length specifier *l*, or if no length is given, by the size associated with the *variable*. If a count specifier is used for the **s** or **a** command, then that many characters are printed. Otherwise successive characters are printed until either a null byte is reached or 128 characters are printed. The last variable may be redisplayed with the command **./**.

The *sh*(1) metacharacters * and ? may be used within procedure and variable names, providing a limited form of pattern matching. If no procedure name is given, variables local to the current procedure and global variables are matched; if a procedure name is specified then only variables local to that procedure are matched. To match only global variables, the form *:pattern* is used.

linenumber?*lm*
variable:?*lm*

Print the value at the address from **a.out** or I space given by *linenumber* or *variable* (procedure name), according to the format *lm*. The default format is 'i'.

variable=*lm*
linenumber=*lm*
number=*lm*

Print the address of *variable* or *linenumber*, or the value of *number*, in the format specified by *lm*. If no format is given, then **lx** is used. The last variant of this command provides a convenient way to convert between decimal, octal and hexadecimal.

variable!*value*

Set *variable* to the given *value*. The value may be a number, a character constant or a variable. The value must be well defined; expressions which produce more than one value, such as structures, are not allowed. Character constants are denoted *'character*. Numbers are viewed as integers unless a decimal point or exponent is used. In this case, they are treated as having the type double. Registers are viewed as integers. The *variable* may be an expression which indicates more than one variable, such as an array or structure name. If the address of a variable is given, it is regarded as the address of a variable of type *int*. C conventions are used in any type conversions necessary to perform the indicated assignment.

x Print the machine registers and the current machine-language instruction.

X Print the current machine-language instruction.

The commands for examining source files are:

e *procedure*

e *file-name*
e *directory/*
e *directory file-name*
The first two forms set the current file to the file containing *procedure* or to *file-name*. The current line is set to the first line in the named procedure or file. Source files are assumed to be in *directory*. The default is the current working directory. The latter two forms change the value of *directory*. If no procedure, file name, or directory is given, the current procedure name and file name are reported.

/regular expression/
Search forward from the current line for a line containing a string matching *regular expression* as in *ed*(1). The trailing / may be deleted.

?regular expression?
Search backward from the current line for a line containing a string matching *regular expression* as in *ed*(1). The trailing ? may be deleted.

p Print the current line.

z Print the current line followed by the next 9 lines. Set the current line to the last line printed.

w Window. Print the 10 lines around the current line.

number
Set the current line to the given line number. Print the new current line.

count+
Advance the current line by *count* lines. Print the new current line.

count−
Retreat the current line by *count* lines. Print the new current line.

The commands for controlling the execution of the source program are:

count **r** *args*
count **R**
Run the program with the given arguments. The **r** command with no arguments reuses the previous arguments to the program while the **R** command runs the program with no arguments. An argument beginning with < or > causes redirection for the standard input or output, respectively. If *count* is given, it specifies the number of breakpoints to be ignored.

linenumber **c** *count*
linenumber **C** *count*
Continue after a breakpoint or interrupt. If *count* is given, it specifies the breakpoint at which to stop after ignoring *count* - 1 breakpoints. **C** continues with the signal which caused the program to stop reactivated and **c** ignores it. If a line number is specified then a temporary breakpoint is placed at the line and execution is continued. The breakpoint is deleted when the command finishes.

linenumber **g** *count*
Continue after a breakpoint with execution resumed at the given line. If *count* is given, it specifies the number of breakpoints to be ignored.

s *count*
S *count*
Single step the program through *count* lines. If no count is given then the program is run for one line. **S** is equivalent to **s** except it steps through procedure calls.

i

I Single step by one machine-language instruction. **I** steps with the signal which caused the program to stop reactivated and **i** ignores it.

variable£**m** *count*
address:**m** *count*
Single step (as with **s**) until the specified location is modified with a new value. If *count* is omitted, it is effectively infinity. *Variable* must be accessible from the current procedure. Since this command is done by software, it can be very slow.

level **v**
Toggle verbose mode, for use when single stepping with **S**, **s** or **m**. If *level* is omitted, then just the current source file and/or subroutine name is printed when either changes. If *level* is 1 or greater, each C source line is printed before it is executed; if *level* is 2 or greater, each assembler statement is also printed. A **v** turns verbose mode off if it is on for any level.

k Kill the program being debugged.

procedure(arg1,arg2,...)
procedure(arg1,arg2,...)/*m*
Execute the named procedure with the given arguments. Arguments can be integer, character or string constants or names of variables accessible from the current procedure. The second form causes the value returned by the procedure to be printed according to format *m*. If no format is given, it defaults to **d**.

linenumber **b** *commands*
Set a breakpoint at the given line. If a procedure name without a line number is given (e.g., "proc:"), a breakpoint is placed at the first line in the procedure even if it was not compiled with the **-g** option. If no *linenumber* is given, a breakpoint is placed at the current line. If no *commands* are given, execution stops just before the breakpoint and control is returned to *sdb*. Otherwise the *commands* are executed when the breakpoint is encountered and execution continues. Multiple commands are specified by separating them with semicolons. If **k** is used as a command to execute at a breakpoint, control returns to *sdb*, instead of continuing execution.

B Print a list of the currently active breakpoints.

linenumber **d**
Delete a breakpoint at the given line. If no *linenumber* is given then the breakpoints are deleted interactively. Each breakpoint location is printed and a line is read from the standard input. If the line begins with a **y** or **d** then the breakpoint is deleted.

D Delete all breakpoints.

l Print the last executed line.

linenumber **a**
Announce. If *linenumber* is of the form *proc*:*number*, the command effectively does a *linenumber* **b l**. If *linenumber* is of the form *proc*:, the command effectively does a *proc*: **b T**.

Miscellaneous commands:

!*command*
The command is interpreted by *sh*(1).

new-line
If the previous command printed a source line then advance the current line by one line and print the new current line. If the previous command displayed a memory location, then display the next memory location.

control-D
Scroll. Print the next 10 lines of instructions, source or data depending on which was printed last.

< *filename*
Read commands from *filename* until the end of file is reached, and then continue to accept commands from standard input. When *sdb* is told to display a variable by a command in such a file, the variable name is displayed along with the value. This command may not be nested; < may not appear as a command in a file.

M Print the address maps.

M [?/][*] *b e f*
Record new values for the address map. The arguments **?** and **/** specify the text and data maps, respectively. The first segment, (*b1, e1, f1*), is changed unless * is specified, in which case the second segment (*b1, e1, f1*), of the mapping is changed. If fewer than three values are given, the remaining map parameters are left unchanged.

" *string*
Print the given string. The C escape sequences of the form *character* are recognized, where *character* is a nonnumeric character.

q Exit the debugger.

The following commands also exist and are intended only for debugging the debugger:

V Print the version number.
Q Print a list of procedures and files being debugged.
Y Toggle debug output.

FILES

a.out
core

SEE ALSO

cc(1), f77(1), sh(1), a.out(4), core(4).

WARNINGS

When *sdb* prints the value of an external variable for which there is no debugging information, a warning is printed before the value. The value is assumed to be **int** (integer).

Data which are stored in text sections are indistinguishable from functions.

Line number information in optimized functions is unreliable, and some information may be missing.

BUGS

If a procedure is called when the program is *not* stopped at a breakpoint (such as when a core image is being debugged), all variables are initialized before the procedure is started. This makes it impossible to use a procedure which formats data from a core image.

The default type for printing F77 parameters is incorrect. Their address is printed instead of their value.

Tracebacks containing F77 subprograms with multiple entry points may print too many arguments in the wrong order, but their values are correct.

The range of an F77 array subscript is assumed to be *1* to *n*, where *n* is the dimension corresponding to that subscript. This is only significant when the user omits a subscript, or uses * to indicate the full range. There is no problem in general with arrays having subscripts whose lower bounds are not 1.

NAME

sdfchmod – change mode of an SDF file

SYNOPSIS

sdfchmod mode device:file ...

DESCRIPTION

Sdfchmod is intended to mimic *chmod*(1).

An SDF file name is recognized by the embedded colon (:) delimiter (see *sdf*(4) for SDF file naming conventions).

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**, which is 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**; **t** only works with **u**.

EXAMPLES

The examples that follow assume that an SDF directory structure exists on the HP-UX device file **/dev/rdsk/c1d0s3**.

The first example denies write permission to others for the SDF directory **/bin**:

sdfchmod o–w /dev/rdsk/c1d0s3:/bin

The second example makes the SDF file **/users/fred/a.out** executable and readable by everyone:

sdfchmod a=rx /dev/rdsk/c1d0s3:/users/fred/a.out

The third example adds read permission for the group associated with the SDF file **/last.boot.rev**:

sdfchmod g+r /dev/rdsk/c1d0s3:/last.boot.rev

The fourth example assigns read and execute permission to everybody, and sets the set-user-id bit for the SDF file **/usr/local/hoo**:

sdfchmod 4555 /dev/rdsk/c1d0s3:/usr/local/hoo

In the fifth example, the two commands perform the same function, namely to give read, write, and execute permission to the owner and read and execute permissions to everybody else for the

SDF file **/users/debbie/script**:

sdfchmod a=rx,u+w /dev/rdsk/c1d0s3:/users/debbie/script

sdfchmod 755 /dev/rdsk/c1d0s3:/users/debbie/script

AUTHOR

Sdfchmod was developed by the Hewlett-Packard Company.

SEE ALSO

sdf(4), chmod(1), chmod(2).

NAME

sdfchown, sdfchgrp – change owner or group of an SDF file

SYNOPSIS

sdfchown owner device:file ...

sdfchgrp group device:file ...

DESCRIPTION

Sdfchown and *sdfchgrp* are intended to mimic *chown*(1) and *chgrp*(1).

An SDF file name is recognized by the embedded colon (:) delimiter (see *sdf*(4) for SDF file naming conventions).

Sdfchown 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.

Sdfchgrp 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.

EXAMPLES

The examples that follow assume that an SDF directory structure exists on the HP-UX device file **/dev/rdsk/c9d1d5**.

The first example sets the owner of the SDF file **/users/abc/phone.num** to **adm**:

sdfchown adm /dev/rdsk/c9d1d5:/users/abc/phone.num

The second example sets the group ID of the SDF file **/tmp/b.date** to the decimal number **105**:

sdfchgrp 105 /dev/rdsk/c9d1d5:/tmp/b.date

AUTHOR

Sdfchown was developed by the Hewlett-Packard Company.

FILES

/etc/passwd
/etc/group

SEE ALSO

sdf(4), chown(1), chgrp(1), group(5), passwd(5).

NAME

sdfcp, sdfln, sdfmv – copy, link, or move files to/from an SDF volume

SYNOPSIS

sdfcp file1 [file2 ...] target
sdfln file1 [file2 ...] target
sdfmv file1 [file2 ...] target

DESCRIPTION

Sdfcp, *sdfln*, *sdfmv* are intended to mimic *cp*(1).

An SDF file name is recognized by the embedded colon (:) delimiter (see *sdf*(4) for SDF file naming conventions).

Sdfcp copies an HP-UX file to an SDF file, or an SDF file to either an SDF or HP-UX file. It also copies a list of HP-UX files to an SDF directory, or copies a list of SDF files to either an SDF or HP-UX directory.

Sdfln creates links to *target* if, and only if, all *files* referenced on the command line are on the same SDF volume.

Sdfmv behaves the same way as *sdfcp*, except that it moves files instead of copying them.

The last name on the argument list is the target file or directory. If two or more files are specified in the command line, not counting *target*, then *target* must be a directory. Under no circumstances may any argument other than *target* be a directory.

The file name "-" (dash) is interpreted to mean standard input or standard output, depending on the position in the argument list. The use of the file name "-" makes no sense for *sdfln* and *sdfmv*.

EXAMPLES

The examples that follow assume that an SDF directory structure exists on the HP-UX device file **/dev/rdsk/c2d0s2**.

The first example copies the HP-UX file **mydata** to the SDF file **/users/old/mike/olddata**:

```
sdfcp mydata /dev/rdsk/c2d0s2:/users/old/mike/olddata
```

The second example copies the SDF file **/users/gary/.cshrc** to the SDF directory **/tmp** (on the same SDF volume):

```
sdfcp /dev/rdsk/c2d0s2:/users/gary/.cshrc /dev/rdsk/c2d0s2:/tmp
```

The third example copies the SDF files **/a/b** and **/a/c** to the HP-UX directory **/users/dave**:

```
sdfcp /dev/rdsk/c2d0s2:/a/b /dev/rdsk/c2d0s2:/a/c /users/dave
```

The fourth example copies standard input to the SDF file **/users/craig/memo**:

```
sdfcp - /dev/rdsk/c2d0s2:/users/craig/memo
```

The fifth example copies the SDF file **/etc/rc** to the SDF file **/etc/rc.old** on another SDF volume residing in the HP-UX device file **/dev/rdsk/c2d1s0**:

```
sdfcp /dev/rdsk/c2d0s2:/etc/rc /dev/rdsk/c2d1s0:/etc/rc.old
```

The sixth example shows how you can implement a *cat*(1) program for concatenating SDF files using *sdfcp* in a shell script:

```
if [ $# -lt 1 ]
then
        echo "Usage: sdfcat file ..."
        exit 1
fi
```

```
for i in $*
do
        sdfcp $i -
done
```

The seventh example links the SDF file **/tmp/x** to **/users/gary/x1**:

```
sdfln /dev/rdsk/c2d0s2:/tmp/x /dev/rdsk/c2d0s2:/users/gary/x1
```

The eighth example moves the HP-UX file **/etc/rc.backup** to the SDF file **/etc/rc**:

```
sdfmv /etc/rc.backup /dev/rdsk/c2d0s2:/etc/rc
```

Assuming that the current HP-UX directory contains only regular files, the ninth example shows how to move all files in an HP-UX directory to the SDF directory **/savestuff**:

```
sdfmv * /dev/rdsk/c2d0s2:/savestuff
```

AUTHOR

Sdfcp was developed by the Hewlett-Packard Company.

SEE ALSO

sdf(4), cp(1).

NAME

sdffind – find files in an SDF system

SYNOPSIS

sdffind path-name-list expression

DESCRIPTION

Sdffind is intended to mimic *find*(1).

An SDF file name is recognized by the embedded colon (:) delimiter (see *sdf*(4) for SDF file naming conventions).

Sdffind 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.

–name *pattern* True if *pattern* matches the current file name.

–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.

–type *n* True if the current file being examined by *sdffind* is a network special 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 (on the local system, not the SDF file system), 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 (on the local system, not the SDF file system), it is taken as a group ID.

–size *n* True if the file is *n* blocks long.

–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 *sdffind* command line anytime you want *sdffind* to print the path names it has found on the standard output. If **–print** is not specified, *sdffind* locates the files, but fails to tell you about them!

When **–print** is specified as the only *expression*, *sdffind* 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*, *sdffind* prints the absolute path names of only those files which satisfy the other primaries in the *expression*.

–inum *n* True if the file has inode number *n*.

EXAMPLES

The examples that follow assume that an SDF directory structure exists on the HP-UX device file **/dev/rdsk/c3d0s0**.

The first example prints the names of all files on the SDF volume **/dev/rdsk/c3d0s0**:

sdffind /dev/rdsk/c3d0s0: -print

The second example prints the name of all the subdirectories under **/usr/lib** on the SDF file system:

sdffind /dev/rdsk/c3d0s0:/usr/lib -type d -print

The third example gives a long listing of every ordinary file under **/users** on the SDF file system:

sdffind /dev/rdsk/c3d0s0:/users -type f -exec sdfls -l {} ';'

The fourth example finds all the files on the SDF volume by the name of "core" and asks whether they should be removed:

sdffind /dev/rdsk/c3d0s0: -name core -ok sdfrm {} ';'

AUTHOR

Sdffind was developed by the Hewlett-Packard Company.

FILES

/etc/passwd
/etc/group

SEE ALSO

sdf(4), find(1), stat(2), chmod(1).

NAME

sdfls, sdfll – list contents of SDF directories

SYNOPSIS

sdfls [**−AadlpFi**] [names]
sdfll [**sdfls options**] [names]

DESCRIPTION

Sdfls is intended to mimic *ls*(1). *Sdfll* is equivalent to **sdfls -l**.

An SDF file name is recognized by the embedded colon (:) delimiter (see *sdf*(4) for SDF file naming conventions).

For each SDF directory named, *sdfls* lists the contents of that SDF directory; for each SDF file named, *sdfls* repeats its name and the information requested.

If you are the super-user, *sdfls* defaults to listing all files except . (current directory) and .. (parent directory).

There are several options to *sdfls*:

−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**. Due to the internal data representation of the SDF directory format, the **−A** and **−a** options perform the same function.

−d If argument is a directory, list only its name; often used with **−l** to get the status of a directory.

−l List in long format giving mode, number of links, owner, group, size in bytes, and time of last modification for each file.

−p Do not use **/etc/passwd** and **/etc/group** to interpret user and group ownership, but rather print out the numeric form.

−F If the entry is a directory or SRM special file, print a '/' character after the entry, or if the entry is executable, print a '*' character after the entry.

−i Print the inode number of each entry before the listing the entry names.

EXAMPLES

The examples that follow assume that an SDF directory structure exists on the HP-UX device file **/dev/rdsk/c7s0s1**.

The first example will list all the files in the root directory of the SDF directory structure:

sdfls −a /dev/rdsk/c7s0s1:

The second example gives (in long format) all the information about the SDF directory **/users/root** itself (but not the files in the directory):

sdfls −ld /dev/rdsk/c7s0s1:/users/root

The third example will print (in long form) all the information about every file in the SDF directory **/etc**, printing numbers instead of names for user and group IDs.

sdfls −ap /dev/rdsk/c7s0s1:/etc

The previous example is useful if the SDF directory structure was not created on your system but brought in from another series 500 system.

HARDWARE DEPENDENCIES

On the Series 500, network special files are supported. With the **−F** option, *sdfls* will print a '/'

character after the entry for a network special file.

AUTHOR

Sdfls was developed by the Hewlett-Packard Company.

FILES

/etc/passwd	to get user ids.
/etc/group	to get group ids.

SEE ALSO

sdf(4), ls(1).

NAME

sdfmkdir – make an SDF directory

SYNOPSIS

sdfmkdir device:dirname ...

DESCRIPTION

Sdfmkdir is intended to mimic *mkdir*(1).

An SDF file name is recognized by the embedded colon (:) delimiter (see *sdf*(4) for SDF file naming conventions).

Sdfmkdir creates specified directories in mode 777, masked with the current value of *umask*.

RETURNS

Sdfmkdir returns exit code **0** if all directories were successfully made; otherwise, it prints a diagnostic and returns non-zero.

EXAMPLES

The following example assumes that an SDF directory structure exists on the HP-UX device file **/dev/rdsk/c0d1s5**.

This example will create an empty subdirectory named **sysmods** under the directory **/usr/lib**:

sdfmkdir /dev/rdsk/c0d1s5:/usr/lib/sysmods

AUTHOR

Sdfmkdir was developed by the Hewlett-Packard Company.

SEE ALSO

sdf(4), mkdir(1).

NAME

sdfrm, sdfrmdir – remove SDF files or directories

SYNOPSIS

sdfrm [**–fri**] device:file ...

sdfrmdir device:dir ...

DESCRIPTION

Sdfrm and *sdfrmdir* are intended to mimic *rm*(1) and *rmdir*(1).

An SDF file name is recognized by the embedded colon (:) delimiter (see *sdf*(4) for SDF file naming conventions).

Sdfrm 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.

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 Remove a file with no questions asked, even if the file has no write permission.

–r Cause *sdfrm* to recursively delete the entire contents of a directory, and then the directory itself. *Sdfrm* can recursively delete up to 17 levels of directories.

–i Cause *sdfrm* to ask whether or not to delete each file. If **–r** is also specified, *sdfrm* asks whether to examine each directory encountered.

Sdfrmdir removes entries for the named directories, which must be empty.

EXAMPLES

The following examples assume that an SDF directory structure exists on the HP-UX device file **/dev/rdsk/c6d0s1**.

The first example recursively combs through the SDF directory **/tmp** and asks if each SDF file should be removed (forced, with no file mode checks):

sdfrm –irf /dev/rdsk/c6d0s1:/tmp

The second example removes the SDF directory **/users/doug**:

sdfrmdir /dev/rdsk/c6d0s1:/users/doug

AUTHOR

Sdfrm was developed by the Hewlett-Packard Company.

SEE ALSO

sdf(4), rm(1), rmdir(1).

NAME

sdiff – side-by-side difference program

SYNOPSIS

sdiff [options ...] file1 file2

DESCRIPTION

Sdiff uses the output of *diff*(1) to produce a side-by-side listing of two files indicating those lines that are different. Each line of the two files is printed with a blank gutter between them if the lines are identical, a < in the gutter if the line only exists in *file1*, a > in the gutter if the line only exists in *file2*, and a | for lines that are different.

For example:

```
x       |       y
a               a
b       <
c       <
d               d
        >       c
```

The following options exist:

–w *n* Use the next argument, *n*, as the width of the output line. The default line length is 130 characters.

–l Only print the left side of any lines that are identical.

–s Do not print identical lines.

–o *output* Use the next argument, *output*, as the name of a third file that is created as a user-controlled merging of *file1* and *file2*. Identical lines of *file1* and *file2* are copied to *output*. Sets of differences, as produced by *diff*(1), are printed; where a set of differences share a common gutter character. After printing each set of differences, *sdiff* prompts the user with a **%** and waits for one of the following user-typed commands:

l	append the left column to the output file
r	append the right column to the output file
s	turn on silent mode; do not print identical lines
v	turn off silent mode
e l	call the editor with the left column
e r	call the editor with the right column
e b	call the editor with the concatenation of left and right
e	call the editor with a zero length file
q	exit from the program

On exit from the editor, the resulting file is concatenated on the end of the *output* file.

SEE ALSO

diff(1), ed(1).

NAME

sed – stream text editor

SYNOPSIS

sed [**–f** sfile] [**–e** script] [**–n**] [files]

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:

n n= 1 - 512. Substitute for just the n th occurrence of the *regular expression*.

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.
(0) #
If a # appears as the first character on the first line of a script file, then that entire line is treated as a comment, with one exception. If the character after the # is an 'n', then the default output will be suppressed. The rest of the line after #n is also ignored. A script file must contain at least one non-comment line.

SEE ALSO

awk(1), ed(1), grep(1).

BUGS

There is a limit of 100 commands in the script.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

send – submit RJE jobs

SYNOPSIS

send argument ...

DESCRIPTION

Send

Send is a command-level interface to the RJE subsystems. It allows the user to collect input from various sources in order to create a run stream consisting of card images, and submit this run stream for transmission to an IBM host computer. Output from the IBM system may be returned to the user in either ASCII text form or EBCDIC punch format (see *pnch*(4)). How output is to be disposed of once it returns from the host is determined by a "usr=" specification which should be embedded in each job that a user submits for transmission. A detailed description of RJE operation and the "usr=" specification is given in *HP-UX Remote Job Entry User Guide*.

Possible sources of input to *send* are: ordinary files, standard input, the terminal, and the output of a command or shell file. Each source of input is treated as a virtual file, and no distinction is made based upon its origin. Typical input is an ASCII text file of the sort that is created by the editor *ed*(1). An optional format specification appearing in the first line of a file (see *fspec*(4)) determines the settings according to which tabs are expanded into spaces. In addition, lines that begin with ~ are normally interpreted as commands controlling the execution of *send*. They may be used to set or reset flags, to define keyword substitutions, and to open new sources of input in the midst of the current source. Other text lines are translated one-for-one into card images of the run stream.

The run stream that results from this collection is treated as one job by the RJE subsystems. *Send* prints the card count of the run stream, and the queuer that is invoked prints the name of the temporary file that holds the job while it is awaiting transmission. The initial card of a job submitted to a host must have a // in the first column. Any cards preceding this card will be excised. If a host computer is not specified before the first card of the runstream is ready to be sent, *send* will select a reasonable default. All cards beginning with **/*$** will be excised from the runstream, because they are HASP command cards.

The arguments that *send* accepts are described below. An argument is interpreted according to the first pattern that it matches. Preceding a character with \ causes it to loose any special meaning it might otherwise have when matching against an argument pattern.

.	Close the current source.
–	Open standard input as a new source.
+	Open the terminal as a new source.
:*spec*:	Establish a default format specification for included sources, e.g., **:m6t–12:**
:*message*	Print message on the terminal.
–:*prompt*	Open standard input and, if it is a terminal, print *prompt*.
+:*prompt*	Open the terminal and print *prompt*.
–*flags*	Set the specified flags, which are described below.
+*flags*	Reset the specified flags.
=*flags*	Restore the specified flags to their state at the previous level.
!*command*	Execute the specified HP-UX system *command* via the one-line shell, with input redirected to **/dev/null** as a default. Open the standard output of the command as a new source.

$*line*	Collect contiguous arguments of this form and write them as consecutive lines to a temporary file; then have the file executed by the shell. Open the standard output of the shell as a new source.
	The current directory for the send process is changed to *directory*. The original directory will be restored at the end of the current source.
~*comment*	Ignore this argument.
?:*keyword*	Prompt for a definition of *keyword* from the terminal unless *keyword* has an existing definition.
?*keyword*=^*xx*	Define the *keyword* as a two-digit hexadecimal character code unless it already has a non-null replacement.
?*keyword*=*string*	Define the *keyword* in terms of a replacement string unless it already has a non-null replacement.
=:*keyword*	Prompt for a definition of *keyword* from the terminal.
keyword=^*xx*	Define *keyword* as a two-digit hexadecimal character code.
keyword=*string*	Define *keyword* in terms of a replacement string.
host	The host machine that the job should be submitted to. It can be any name that corresponds to one in the first column of the RJE configuration file (**/usr/rje/lines**).
file-name	Open the specified file as a new source of input.

When commands are executed via **$** or **!** the shell environment (see *environ*(5)) will contain the values of all send keywords that begin with **$** and have the syntax of a shell variable.

The flags recognized by *send* are described in terms of the special processing that occurs when they are set:

-l	List card images on standard output. EBCDIC characters are translated back to ASCII.
-q	Do not output card images.
-f	Do not fold lower case to upper.
-t	Trace progress on diagnostic output, by announcing the opening of input sources.
-k	Ignore the keywords that are active at the previous level and erase any keyword definitions that have been made at the current level.
-r	Process included sources in raw mode; pack arbitrary 8-bit bytes one per column (80 columns per card) until an EOF.
-i	Do not interpret control lines in included sources; treat them as text.
-s	Make keyword substitutions before detecting and interpreting control lines.
-y	Suppress error diagnostics and submit job anyway.
-g	Gather mode, qualifying **-l** flag; list text lines before converting them to card images.
-h	Write listing with standard tabs.
-p	Prompt with * when taking input from the terminal.
-m	When input returns to the terminal from a lower level, repeat the prompt, if any.

–a Make **–k** flag propagate to included sources, thereby protecting them from keyword substitutions.

–c List control lines on diagnostic output.

–d Extend the current set of keyword definitions by adding those active at the end of included sources.

–x This flag guarantees that the job will be transmitted in the order of submission (relative to other jobs sent with this flag).

Control lines are input lines that begin with ~. In the default mode **+ir**, they are interpreted as commands to *send*. Normally they are detected immediately and read literally. The **–s** flag forces keyword substitutions to be made before control lines are intercepted and interpreted. This can lead to unexpected results if a control line uses a keyword which is defined within an immediately preceding ~$ sequence. Arguments appearing in control lines are handled exactly like the command arguments to *send*, except that they are processed at a nested level of input.

The two possible formats for a control line are: "~argument" and "~##argument#...". In the first case, where the ~ is not followed by a space, the remainder of the line is taken as a single argument to *send*. In the second case, the line is parsed to obtain a sequence of arguments delimited by spaces. In this case the quotes ' and " may be employed to pass embedded spaces.

The interpretation of the argument . is chosen so that an input line consisting of ~. is treated as a logical EOF. The following example illustrates some of the above conventions:

```
send## –
~##argument ...
~.
```

This sequence of three lines is equivalent to the command synopsis at the beginning of this description. In fact, the – is not even required. By convention, the *send* command reads standard input if no other input source is specified. *Send* may therefore be employed as a filter with side-effects.

The execution of the *send* command is controlled at each instant by a current environment, which includes the format specification for the input source, a default format specification for included sources, the settings of the mode flags, and the active set of keyword definitions. This environment can be altered dynamically. When a control line opens a new source of input, the current environment is pushed onto a stack, to be restored when input resumes from the old source. The initial format specification for the new source is taken from the first line of the file. If none is provided, the established default is used or, in its absence, standard tabs. The initial mode settings and active keywords are copied from the old environment. Changes made while processing the new source will not affect the environment of the old source, with one exception: if **–d** mode is set in the old environment, the old keyword context will be augmented by those definitions that are active at the end of the new source.

When *send* first begins execution, all mode flags are reset, and the values of the shell environment variables become the initial values for keywords of the same name with a **$** prefixed.

The initial reset state for all mode flags is the + state. In general, special processing associated with a mode *N* is invoked by flag *–N* and is revoked by flag *+N*. Most mode settings have an immediate effect on the processing of the current source. Exceptions to this are the **–r** and **–i** flags, which apply only to included source, causing it to be processed in an uninterpreted manner.

A keyword is an arbitrary 8-bit ASCII string for which a replacement has been defined. The replacement may be another string or the hexadecimal code for a single 8-bit byte. At any instant, a given set of keyword definitions is active. Input text lines are scanned, in one pass from left to right, and longest matches are attempted between substrings of the line and the

active set of keywords. Characters that do not match are output, subject to folding and the standard translation. Keywords are replaced by the specified hexadecimal code or replacement string, which is then output character by character. The expansion of tabs and length checking, according to the format specification of an input source, are delayed until substitutions have been made in a line.

All of the keywords definitions made in the current source may be deleted by setting the **−k** flag. It then becomes possible to reuse them. Setting the **−k** flag also causes keyword definitions active at the previous source level to be ignored. Setting the **+k** flag causes keywords at the previous level to be ignored but does not delete the definitions made at the current level. The **=k** argument reactivates the definitions of the previous level.

When keywords are redefined, the previous definition at the same level of source input is lost, however the definition at the previous level is only hidden, to be reactivated upon return to that level unless a **−d** flag causes the current definition to be retained.

Conditional prompts for keywords, **?:A**,/p which have already been defined at some higher level to be null or have a replacement will simply cause the definitions to be copied down to the current level; new definitions will not be solicited.

Keyword substitution is an elementary macro facility that is easily explained and that appears useful enough to warrant its inclusion in the *send* command. More complex replacements are the function of a general macro processor such as *m4*(1). To reduce the overhead of string comparison, it is recommended that keywords be chosen so that their initial characters are unusual. For example, let them all be upper case.

Send performs two types of error checking on input text lines. Primarily, only ASCII graphics and tabs are permitted in input text. Secondly, the length of a text line, after substitutions have been made, may not exceed 80 bytes. The length of each line may be additionally constrained by a size parameter in the format specification for an input source. Diagnostic output provides the location of each erroneous line, by line number and input source, a description of the error, and the card image that results. Other routine errors that are announced are the inability to open or write files, and abnormal exits from the shell. Normally, the occurrence of any error causes *send*, before invoking the queuer, to prompt for positive affirmation that the suspect run stream should be submitted.

Before submitting a job to a host, *send* translates 8-bit ASCII characters into their EBCDIC equivalents. The conversion for 8-bit ASCII characters in the octal range 040-176 is based on the character set described in Appendix H of *IBM System/370 Principles of Operation* (IBM SRL GA22-7000). Each 8-bit ASCII character in the range 040-377 possesses an EBCDIC equivalent into which it is mapped, with five exceptions: ~ into EBCDIC not, 0345 into ~, 0325 into EBCDIC cent, 0313 into EBCDIC split-bar; 0177 (DEL) is illegal. In listings requested from *send* and in printed output returned by the subsystem, the reverse translation is made with the qualification that EBCDIC characters that do not have valid 8-bit ASCII equivalents are translated into ^.

Additional control over the translation process is afforded by the **−f** flag and hexadecimal character codes. As a default, *send* folds lower-case letters into upper case. Setting the **−f** flag inhibits any folding. Non-standard character codes are obtained as a special case of keyword substitution. The users should check with the remote IBM system to be sure the special processing will be accepted.

SEE ALSO

m4(1), sh(1), lseek(2), ascii(5), fspec(4), pnch(4), environ(5).

BUGS

Standard input is read in blocks, and unused bytes are returned via *lseek*(2). If standard input is a pipe, multiple arguments of the form **−** and **−:*prompt*** should not be used, nor should the logical

EOF (~.).

NAME

sh, rsh – shell, the standard/restricted command programming language

SYNOPSIS

sh [**–acefhiknrstuvx**] [args]
rsh [**–acefhiknrstuvx**] [args]

DESCRIPTION

Sh is a command programming language that executes commands read from a terminal or a file. *Rsh* is a restricted version of the standard command interpreter *sh*; it is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. See **Invocation** below for the meaning of arguments to the shell.

Definitions

A *blank* is a tab or a space. A *name* is a sequence of letters, digits, or underscores beginning with a letter or underscore. A *parameter* is a name, a digit, or any of the characters *, #, ?, –, $, and !.

Commands

A *simple-command* is a sequence of non-blank *words* separated by *blanks*. 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 **|** (or, for historical compatibility, by ^). The standard output of each command but the last is connected by a *pipe*(2) to the standard input of the next command. Each command is run as a separate process; the shell waits for the last command to terminate. The exit status of a pipeline is the exit status of the last command.

A *list* is a sequence of one or more pipelines separated by **;**, **&**, **&&**, or **||**, and optionally terminated by **;** or **&**. Of these 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 filename generation (see **File Name Generation**) except that a slash, a leading dot, or a dot immediately following a slash need not be matched explicitly.

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.

name () { *list*;} Define a function which is referenced by *name*. The body of the function is the *list* of commands between { and }. Execution of functions is described below (see **Execution**).

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*. There are two types of parameters, positional and keyword. If *parameter* is a digit, it is a positional parameter. Positional parameters may be assigned values by **set**. Keyword parameters (also known as variables) may be assigned values by writing:

name=*value* [*name*=*value*] ...

Pattern-matching is not performed on *value*. There cannot be a function and a variable with the same *name*.

${***parameter***}** 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. If *parameter* is * or 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, substitute its value; otherwise substitute *word*.

${*parameter***:=***word***}**
If *parameter* is not set or is null 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, 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.
$	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). The user may not change **PATH** if executing under *rsh*.
CDPATH	The search path for the *cd* command.
MAIL	If this parameter is set to the name of a mail file *and* the **MAILPATH** parameter is not set, the shell informs the user of the arrival of mail in the specified file.
MAILCHECK	This parameter specifies how often (in seconds) the shell will check for the arrival of mail in the files specified by the **MAILPATH** or **MAIL** parameters. The default value is 600 seconds (10 minutes). If set to 0, the shell will check before each prompt.
MAILPATH	A colon (:) separated list of file names. If this parameter is set, the shell informs the user of the arrival of mail in any of the specified files. Each file name can be followed by **%** and a message that will be printed when the modification time changes. The default message is *you have mail*.
PS1	Primary prompt string, by default "**$** ".
PS2	Secondary prompt string, by default "> ".
IFS	Internal field separators, normally **space**, **tab**, and **new-line**.
SHACCT	If this parameter is set to the name of a file writable by the user, the shell will write an accounting record in the file for each shell procedure executed. Accounting routines such as *acctcom*(1) and *acctcms*(1M) can be used to analyze the data collected.
SHELL	When the shell is invoked, it scans the environment (see **Environment** below) for this name. If it is found and there is an 'r' in the file name part of its value, the shell becomes a restricted shell. **SHELL** is also used by some processors to determine which command interpreter to run.

The shell gives default values to **PATH**, **PS1**, **PS2**, **MAILCHECK** and **IFS**. **HOME** and **MAIL** are 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. If the first character following the opening ``['' is a **"!"** any character not enclosed is matched.

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	Use the file associated with file descriptor *digit* as standard input. Similarly for the standard output using **>&digit**. (See *dup*(2)).
<&–	The standard input is closed. Similarly for the standard output using **>&–**.

If any of the above is preceded by a digit, then the file descriptor which will be associated with the file is that specified by the digit (instead of the default 0 or 1). For example:

... 2>&1

associates file descriptor 2 with the file currently associated with 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).

The order in which redirections are specified is significant. The shell evaluates redirections left-to-right. For example:

... 1>*xxx* 2>&1

first associates file descriptor 1 with file *xxx*. It associates file descriptor 2 with the file associated with file descriptor 1 (i.e. *xxx*). If the order of redirections were reversed, file descriptor 2 would be associated with the terminal (assuming file descriptor 1 had been) and file descriptor 1 would be associated with file *xxx*.

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.

Redirection of output is not allowed in the restricted shell.

Environment

The *environment* (see *environ*(5)) 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 value of any of these parameters or creates new parameters, none of these affects the environment unless the **export** command is used to bind the shell's parameter to the environment (see also **set -a**). A parameter may be removed from the environment with the **unset** command. The environment seen by any executed command is thus composed of any unmodified name-value pairs originally inherited by the shell, minus any pairs removed by **unset**, 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                    and
(export TERM; TERM=450; cmd)
```

are equivalent (as far as the 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. If the command name matches one of the **Special Commands** listed below, it is executed in the shell process. If the command name does not match a *Special Command*, but matches the name of a defined function, the function is executed in the shell process (note how this differs from the execution of shell procedures). The positional parameters **$1**, **$2**, ... are set to the arguments of the function. If the command name matches neither a *Special Command* nor the name of a defined function, 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; such commands will not be executed by the restricted shell. 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.

The location in the search path where a command was found is remembered by the shell (to help avoid unnecessary *execs* later). If the command was found in a relative directory, its location must be re-determined whenever the current directory changes. The shell forgets all remembered locations whenever the **PATH** variable is changed or the **hash -r** command is executed (see below).

Special Commands

The following commands are executed in the shell process. Input/output redirection is permitted for these commands. File descriptor 1 is the default output location.

: 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*. The shell parameter **CDPATH** defines the search path for the directory containing *arg*. Alternative directory names are separated by a colon (:). The default path is **<null>** (specifying the current directory). Note that the current directory is specified by a null path name, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If *arg* begins with a / the search path is not used. Otherwise, each directory in the path is searched for *arg*. The *cd* command may not be executed by *rsh*.

echo [*arg* ...] Echo arguments. See *echo*(1) for usage and description.

eval [*arg* ...] The arguments are read as input to the shell and the resulting command(s) executed.

exec [*arg* ...] 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 *names* 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. Function names may *not* be exported.

hash [**-r**] [*name* ...]
For each *name*, the location in the search path of the command specified by *name* is determined and remembered by the shell. The **-r** option causes the shell to forget all remembered locations. If no arguments are given, information about remembered commands is presented. *Hits* is the number of times a command has been invoked by the shell process. *Cost* is a measure of the work required to locate a command in the search path. There are certain situations which require that the stored location of a command be recalculated. Commands for which this will be done are indicated by an asterisk (*) adjacent to the *hits* information. *Cost* will be incremented when the recalculation is done.

newgrp [*arg* ...]
Equivalent to **exec newgrp** *arg* .... See *newgrp*(1) for usage and description.

pwd Print the current working directory. See *pwd*(1) for usage and description.

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 *names* are marked *readonly* and the values of the these *names* may not be changed by subsequent assignment. If no arguments are given, then a list of all *readonly* names is printed.

return [*n*] Causes a function to exit with the return value specified by *n*. If *n* is omitted, the return status is that of the last command executed.

set [**—aefhkntuvx** [*arg* ...]]

-a Mark variables which are modified or created for export.

-e Exit immediately if a command exits with a non-zero exit status.

-f Disable file name generation

-h Locate and remember function commands as functions are defined (function commands are normally located when the function is executed).

-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 [*n*] The positional parameters from **$n+1** ... are renamed **$1** .... If *n* is not given, it is assumed to be 1.

test Evaluate conditional expressions. See *test*(1) for usage and description. Note that "[...]" in an **if** *list* is interpreted the same as "**test** ...". There must be blanks around the brackets.

times Print the accumulated user and system times for processes run from the shell.

trap [*arg*] [*n*] ...
The command *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) or signal 18 (death of child) will produce 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.

type [*name* ...]
For each *name*, indicate how it would be interpreted if used as a command name.

ulimit [**–f** [*n*]]
If the **–f** *n* option is used, a size limit of *n* blocks is imposed on files written by child processes (files of any size may be read). With no argument, the current limit is printed. If no option is given, **–f** is assumed.

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.

unset [*name* ...]
For each *name*, remove the corresponding variable or function. The variables **PATH**, **PS1**, **PS2**, **MAILCHECK** and **IFS** cannot be unset.

wait [*n*] Wait for the specified process and report its termination status. If *n* is not given all currently active child processes are waited for and the return code is 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 (except for *Special Commands*) 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.

The remaining flags and arguments are described under the **set** command above.

Rsh Only

Rsh 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:

changing directory (see *cd*(1)),
setting the value of **$PATH,**
specifying path or command names containing **/**,
redirecting output (> and >>).

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 imposing a limited menu of commands; this scheme assumes that the end-user does not have write and execute permissions in the same directory.

The net effect of these rules is that the writer of the **.profile** has complete control over user actions, by performing guaranteed setup actions and leaving the user in an appropriate directory (probably *not* the login directory).

The system administrator often sets up a directory of commands (i.e., **/usr/rbin**) that can be safely invoked by *rsh*. Some systems also provide a restricted editor *red*.

FILES

$HOME/.profile
/dev/null
/etc/profile
/tmp/sh*

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

acctcms(1M), acctcom(1), cd(1), echo(1), env(1), login(1), newgrp(1), pwd(1), test(1), umask(1), dup(2), exec(2), fork(2), pipe(2), signal(2), ulimit(2), umask(2), wait(2), a.out(4), profile(4), environ(5).

CAVEATS

If a command is executed, and a command with the same name is installed in a directory in the search path before the directory where the original command was found, the shell will continue to *exec* the original command. Use the **hash** command to correct this situation.

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. If you move the current directory or one above it, **pwd** may not give the correct response. Use the **cd** command with a full path name to correct this situation.

The command **readonly** (without arguments) produces the same output as the command **export**.

INTERNATIONAL SUPPORT

sh: 8- and 16-bit data, 8-bit filenames, messages.

Series 300 and 500 Only

NAME

shl – shell layer manager

SYNOPSIS

shl

DESCRIPTION

Shl enables a user to access and interact with two or more shells from a single terminal. These shells, known as *layers*, are controlled by using the commands described below.

The *current layer* is the layer which can receive input from the keyboard. Other layers attempting to read from the keyboard are blocked. Output from multiple layers is multiplexed onto the terminal. To have the output of a layer blocked when it is not current, the *stty* option **loblk** may be set within the layer.

The *stty* character **swtch** (set to ^Z if NUL) is used to switch control to *shl* from a layer. *Shl* has its own prompt, >>>, to help distinguish it from a layer.

A *layer* is a shell which has been bound to a pseudo tty device (**/dev/pty/tty??**). The pseudo device can be manipulated like a real tty device using *stty*(1) and *ioctl*(2). Each layer has its own process group id.

Definitions

A *name* is a sequence of characters delimited by a blank, tab or new-line. Only the first eight characters are significant. The *names* **(1)**, through **(2)**, ... are used. Names can be abbreviated to just the digit.

Commands

The following commands may be issued from the *shl* prompt level. Any unique prefix is accepted.

create *name*
: Create a layer called *name* and make it the current layer. If no argument is given, a layer is created with a name of the form *(#)* where # is the number of the next available slot in an internal table. The shell prompt variable **PS1** is set to the name of the layer followed by a space. The number of layers that can be created is dependent on the number of available pseudo-terminals.

block *name* [*name* ...]
: For each *name*, block the output of the corresponding layer when it is not the current layer. This is equivalent to setting the *stty* option **loblk** within the layer.

delete *name name* ...
: For each *name*, delete the corresponding layer. All processes in the process group of the layer are sent the SIGHUP signal (see *signal*(2)).

help or **?**	Print the syntax of the *shl* commands.
layers –l *name* ...	For each *name*, list the layer name and its process group. The –l option produces a *ps*(1)-like listing. If no arguments are given, information is presented for all existing layers.
resume *name*	Make the layer referenced by *name* the current layer. If no argument is given, the last existing current layer will be resumed.
toggle	Resume the layer that was current before the last current layer.
unblock *name* [*name* ...]	For each *name*, do not block the output of the corresponding layer when it is not the current layer. This is equivalent to setting the *stty* option –**loblk** within the layer.
quit	Exit *shl*. Sends the SIGHUP signal to all layers.
name	Make the layer referenced by *name* the current layer.

FILES

/dev/pty/tty??	Pseudo tty devices
$SHELL	Variable containing path name of the shell to use (default is */bin/sh*).

SEE ALSO

sh(1), stty(1), ioctl(2), signal(2), pty(7).

NAME

size – print section sizes of object files

SYNOPSIS

size [**–d**] [**–o**] [**–x**] [**–V**] files

DESCRIPTION

The *size* command produces section size information for each section in the 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.

Numbers will be printed in decimal unless either the **–o** or the **–x** option is used, in which case they will be printed in octal or in hexadecimal, respectively.

The **–V** flag will supply the version information on the *size* command.

HARDWARE DEPENDENCIES

Series 200, 300

The **–V** option is not supported.

Series 500

The **–V** option is not supported.

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.

Series 800

Size information is printed for each subspace in the object file. Each subspace size is followed by the subspace name in parentheses.

SEE ALSO

as(1), cc(1), ld(1). a.out(4), ar(4).

DIAGNOSTICS

size: name: cannot open	if *name* cannot be read.
size: name: bad magic	if *name* is not an appropriate object file.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

sleep – suspend execution for an interval

SYNOPSIS

sleep time

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

slp – set the options for a printer

SYNOPSIS

slp [**–a**] [**–b**] [**–c** cols] [**–d**] [**–i** indent] [**–l** lines] [**–n**] [**–C** pages] [**–O** pages]

DESCRIPTION

Slp sets printer status information such as the number of lines per page, the number of characters per line, and the indentation. These characteristics are controlled by the printer drive as described in *lp*(7). *Slp* acts on the current standard output.

The meanings of the options are:

–a Reports all of the option settings.

–b Indicates that this is a character printer; back spaces are to pass through the driver unchanged. The absence of this option indicates a line printer. The driver takes the necessary action to accommodate a backspace character.

–c *cols* *Cols* selects the number of columns to be printed. Characters beyond the last specified column will be truncated.

–d Resets options to the defaults for the device. (This action is not taken until the next open occurs on the device.)

–i *indent* *Indent* selects the number of columns to indent before the first printed column.

–l *lines* *Lines* selects the number of lines per page. The last new-line character of each page will be changed to a form-feed.

–n Set the page size to infinity. (Since the last new-line of the page is never encountered, no new-line characters will be changed to form-feeds.)

–C *pages* Zero or more *pages* may be ejected after the final close of the device.

–O *pages* Zero or more *pages* may be ejected as the device is opened.

EXAMPLE

A typical case is to set the printer to 80 columns, no indentation, and no form-feeds between pages:

```
slp -c80 -i0 -n >/dev/lp
```

HARDWARE DEPENDENCIES

Integral PC

This command is not available. Refer to the *Integral Personal Computer Programmer's Guide* for more information about the *lp* implementation on the Integral PC.

Series 200, 300

The value of *cols* will be forced into the range of 1 to 227, the value of *indent* from 0 to 227, and the value of *lines* from 1 to MAXSHORT. The **-b**, **-C**, and **-O** options are not supported.

The uppercase-only flag, the no-overprint flag, the raw-mode flag, and no-page-eject-on-open-or-close flag can be selected (enabled) by appropriate use of the *minor* number in the *mknod*(1M) command. See the *HP-UX System Administrator Manual* for details.

Series 500

This command is not available. However the number of characters per line (80 or 132) and wrap-around can be selected (enabled) via the *minor* number in the *mknod*(1M) command. See the *HP-UX System Administrator Manual* for details.

AUTHOR

Slp was developed by the Hewlett-Packard Company.

SEE ALSO

ioctl(2), lp(7).

NAME

sort – sort and/or merge files

SYNOPSIS

sort [[**–cmu**] [**–o**output] [**–y**kmem] [**–z**recsz] [**–dfiMnrl**] [**–tbx**] [+pos1 [–pos2]]] [files]

DESCRIPTION

Sort sorts lines of all the named files together and writes the result on the standard output. The standard input is read if – is used as a file name or no input files are named.

Comparisons are based on one or more sort keys extracted from each line of input. By default, there is one sort key, the entire input line, and ordering is lexicographic by bytes in machine collating sequence.

International Support: Specifying the **–l** option causes sorting to be performed using the collation sequence associated with the specified language. If the language is not specified or is set to **n_computer,** the ordering is lexicographic by bytes in machine-collating sequence.

If the user's language includes two-byte characters, one-byte characters are machine-collated before two-byte characters.

The following options alter the default behavior:

–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 Unique: suppress all but one in each set of lines having equal keys.

–o*output* The argument given 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. There may be optional blanks between **–o** and *output.*

–y*kmem* The amount of main memory used by the sort has a large impact on its performance. Sorting a small file in a large amount of memory is a waste. If this option is omitted, *sort* begins using a system default memory size, and continues to use more space as needed. If this option is presented with a value, *kmem*, *sort* will start using that number of kilobytes of memory, unless the administrative minimum or maximum is violated, in which case the corresponding extremum will be used. Thus, **–y**0 is guaranteed to start with minimum memory. By convention, **–y** (with no argument) starts with maximum memory.

–z*recsz* The size of the longest line read is recorded in the sort phase so buffers can be allocated during the merge phase. If the sort phase is omitted via the **–c** or **–m** options, a popular system default size will be used. Lines longer than the buffer size will cause *sort* to terminate abnormally. Supplying the actual number of bytes in the longest line to be merged (or some larger value) will prevent abnormal termination.

The following options override the default ordering rules.

–d "Dictionary" order: only letters, digits and blanks (spaces and tabs) are significant in comparisons.

–f Fold lowercase letters into uppercase. The **-f** option is ignored if a language other than **n_computer** is specified.

–i Ignore characters outside the ASCII range 040-0176 in non-numeric comparisons. The **-i** option will be ignored if a language other than **n_computer** is specified.

-M Compare as months. The first three non-blank characters of the field are folded to uppercase and compared so that "JAN" < "FEB" < ... < "DEC". Invalid field compare low to "JAN". The **-M** option implies the **-b** option (see below).

-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. The **-n** option implies the **-b** option (see below). Note that the **-b** option is only effective when restricted sort key specifications are in effect.

-r Reverse the sense of comparisons.

The following option applies to International Support (see above).

-l Collate characters using the collation rules associated with the user's LANG variable, see *environ*(5).

When ordering options appear before restricted sort key specifications, the requested ordering rules are applied globally to all sort keys. When attached to a specific sort key (described below), the specified ordering options override all global ordering options for that key.

The notation *+pos1 -pos2* restricts a sort key to one beginning at *pos1* and ending at *pos2*. The characters at positions *pos1* and *pos2* are included in the sort key (provided that *pos2* does not precede *pos1*). A missing *-pos2* means the end of the line.

Specifying *pos1* and *pos2* involves the notion of a field, a minimal sequence of characters followed by a field separator or a new-line. By default, the first blank (space or tab) of a sequence of blanks acts as the field separator. All blanks in a sequence of blanks are considered to be part of the next field; for example, all blanks at the beginning of a line are considered to be part of the first field. The treatment of field separators can be altered using the options:

-t*x* Use *x* as the field separator character; *x* is not considered to be part of a field (although it may be included in a sort key). Each occurrence of *x* is significant (e.g., *xx* delimits an empty field).

-b Ignore leading blanks when determining the starting and ending positions of a restricted sort key. If the **-b** option is specified before the first *+pos1* argument, it will be applied to all *+pos1* arguments. Otherwise, the **b** flag may be attached independently to each *+pos1* or *-pos2* argument (see below).

Pos1 and *pos2* each have the form *m.n* optionally followed by one or more of the flags **bdfinr**. A starting position specified by *+m.n* is interpreted to mean the *n*+1st character in the *m*+1st field. A missing *.n* means .0, indicating the first character of the *m*+1st field. If the **b** flag is in effect *n* is counted from the first non-blank in the *m*+1st field; *+m.0***b** refers to the first non-blank character in the *m*+1st field.

A last position specified by *-m.n* is interpreted to mean the *n*th character (including separators) after the last character of the *m th* field. A missing *.n* means .0, indicating the last character of the *m*th field. If the **b** flag is in effect *n* is counted from the last leading blank in the *m*+1st field; *-m.1***b** refers to the first non-blank in the *m*+1st field.

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.

EXAMPLES

Sort the contents of *infile* with the second field as the sort key:

```
sort +1 -2 infile
```

Sort, in reverse order, the contents of *infile1* and *infile2*, placing the output in *outfile* and using the first character of the second field as the sort key:

sort –r –o outfile +1.0 –1.2 infile1 infile2

Sort, in reverse order, the contents of *infile1* and *infile2* using the first non-blank character of the second field as the sort key:

sort –r +1.0b –1.1b infile1 infile2

Print the password file (*passwd*(4)) sorted by the numeric user ID (the third colon-separated field):

sort –t: +2n –3 /etc/passwd

Print the lines of the already sorted file *infile*, suppressing all but the first occurrence of lines having the same third field (the options **–um** with just one input file make the choice of a unique representative from a set of equal lines predictable):

sort –um +2 –3 infile

FILES

/usr/tmp/stm???

SEE ALSO

comm(1), join(1), uniq(1), col_seq_8(4), environ(5), hpnls(5), langid(5).

DIAGNOSTICS

Comments and exits with non-zero status for various trouble conditions (e.g., when input lines are too long), and for disorder discovered under the **–c** option. When the last line of an input file is missing a **new-line** character, *sort* appends one, prints a warning message, and continues.

If there is an error in accessing the tables containing the collation rules for the specified language, *sort* prints a warning message and defaults to **n_computer.**

The **–d** option recognizes ASCII characters only. If a language other than **n_computer** is specified with the **-d** option, *sort* prints a warning message and defaults to **n_computer.**

The **–M** option compares American month names only. If a language other than **n_computer** is specified with the **-M** option, *sort* prints a warning message and defaults to **n_computer.**

The **–n** option only recognizes the English radix character (decimal point) in numeric comparisons. If a language other than **n_computer** is specified with the **-n** option, *sort* prints a warning message and defaults to **n_computer.**

BUGS

When using the specified ordering option(s) with two-byte characters, *pos1* and *pos2* must specify byte position, not character position.

The **–t** option only recognizes a character encoded in one byte as a field separator character.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

spell, hashmake, spellin, hashcheck – find spelling errors

SYNOPSIS

spell [**–v**] [**–b**] [**–x**] [**–l**] [**–i**] [+local_file] [files]

/usr/lib/spell/hashmake

/usr/lib/spell/spellin n

/usr/lib/spell/hashcheck spelling_list

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 *troff*, *tbl*(1), and *eqn* constructions.

Options

–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*, *programme*, *speciality*, *travelled*, etc., this option insists upon *-ise* in words like *standardise*.

–x Every plausible stem is printed with = for each word.

By default, *spell* (like *deroff*(1)) follows chains of included files (**.so** and **.nx** *troff* requests), *unless* the names of such included files begin with **/usr/lib**. Under the **–l** option, *spell* will follow the chains of *all* included files. Under the **–i** option, *spell* will ignore all chains of included files.

Under the *+local_file* option, words found in *local_file* are removed from *spell*'s output. *Local_file* is the name of a user-provided file that contains a sorted list of words, one per line. With this option, the user can specify a set of words that are correct spellings (in addition to *spell*'s own spelling list) for each job.

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 (see **FILES**). 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.

Three routines help maintain and check the hash lists used by *spell*:

hashmake Reads a list of words from the standard input and writes the corresponding nine-digit hash code on the standard output.

spellin n Reads *n* hash codes from the standard input and writes a compressed spelling list on the standard output. Information about the hash coding is printed on standard error.

hashcheck Reads a compressed *spelling_list* and recreates the nine-digit hash codes for all the words in it; it writes these codes on the standard output.

EXAMPLES

The following example creates the hashed spell list **hlist** and checks the result by comparing the two temporary files; they should be equal.

```
cat goodwds | /usr/lib/spell/hashmake | sort -u >tmp1
cat tmp1 | /usr/lib/spell/spellin `cat tmp1 | wc -l` >hlist
cat hlist | /usr/lib/spell/hashcheck >tmp2
diff tmp1 tmp2
```

WARNINGS

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 file that is added to the hashed *spelling_list* via *spellin*.

The British spelling feature was done by an American.

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
/usr/lib/spell/spellprog	program

VARIABLES

D_SPELL	Your hashed spelling list. (Default as above.)
H_SPELL	Spelling history. (Default as above.)
S_SPELL	Your hashed stop list. (Default as above.)

SEE ALSO

deroff(1), sed(1), sort(1), tbl(1), tee(1).

NAME

split – split a file into pieces

SYNOPSIS

split [*−n*] [file [name]]

DESCRIPTION

Split reads *file* and writes it in *n*-line pieces (default 1000 lines) onto a set of output files. The name of the first output file is *name* with **aa** appended, and so on lexicographically, up to **zz** (a maximum of 676 files). *Name* cannot be longer than 12 characters. 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.

SEE ALSO

bfs(1), csplit(1).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

Series 800 Only

NAME

sqlutil – ALLBASE/HP-UX DBCore utilities

SYNOPSIS

sqlutil

REMARKS

The ALLBASE/HP-UX product must be previously installed on the system for *sqlutil* to function.

DESCRIPTION

Sqlutil invokes the DBCore utility program for maintaining and reconfiguring an ALLBASE/HP-UX relational DataBase Environment (DBEnvironment). There are no options available with this command. Sqlutil can be executed by all system users on all DataBase Environment Configuration (DBECon) files created by them.

AUTHOR

Sqlutil was developed by Hewlett-Packard.

FILES

/usr/bin/hpdbdaemon	cleanup daemon program file
/usr/lib/hpsqlproc	HP SQL program file
/usr/bin/sqlutil	SQLUTIL program file
/usr/lib/hpsqlcat	HP SQL message catalog file

SEE ALSO

ALLBASE/HP-UX SQL Reference Manual.

NAME

ssp – remove multiple line-feeds from output

SYNOPSIS

ssp

DESCRIPTION

Ssp (single-space) removes redundant blank lines from the standard input and sends the result to the standard output. It is typically used in pipelines like

```
nroff -ms file1 | ssp
```

Ssp is equivalent to the 4.2BSD **cat -s** command.

SEE ALSO

cat(1), rmnl(1).

NAME

strings – find the printable strings in an object, or other binary, file

SYNOPSIS

strings [**–a**] [**–o**] [*–number*] [file] ...

DESCRIPTION

Strings looks for ascii strings in a 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.

The following flags are defined.

a By default, *strings* only looks in the initialized data space of object files (as recognised by their magic numbers). If this flag is used, the whole file is inspected. This flag is always set if stdin is being read or the file is not recognized as an object file. For backward compatibility, – is taken as a synonym for **–a**.

o Each string is preceded by its offset in the file (in octal). *number number* is used as the minimum string length rather than 4.

Strings is useful for identifying random object files and many other things.

AUTHOR

Strings was developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

od(1)

BUGS

The algorithm for identifying strings is extremely primitive.

Series 200, 300, 500 Only

NAME

strip – remove symbols and debug information

SYNOPSIS

strip name ...

DESCRIPTION

Strip removes the symbol table and debug information from an executable object file. Once this is done, no symbolic debugging access will be available for that file; therefore this command is normally run only on production modules that have been debugged and tested. The effect is the same as use of the **–s** option of *ld*.

If *name* is a relocatable file, *strip* removes the debug information.

If the *strip* command is executed on an archive file (see *ar* (5)) the archive symbol table will be removed. The archive symbol table must be restored by executing the *ar* (1) command with the **s** option before the archive can be link-edited by the *ld* (1) command. Also, strip removes the debug information from any *a.out* file it finds in the archive.

The purpose of this command is to reduce the file storage overhead taken by the object file.

HARDWARE DEPENDENCIES

Series 200:

If *name* is a relocatable file, *strip* will remove the local symbols from it. 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/s* temporary files

SEE ALSO

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

NAME

strip – strip symbol and line number information from an object file

SYNOPSIS

strip [–l] [–x] [–r] [–V] filename

DESCRIPTION

The *strip* command strips the symbol table and line number information from object files, including archives. Once this has been done, no symbolic debugging access will be available for that file; therefore, this command is normally run only on production modules that have been debugged and tested.

The amount of information stripped from the symbol table can be controlled by using any of the following options:

–l	Strip line number information only; do not strip any symbol table information.
–x	Do not strip static or external symbol information.
–r	Reset the relocation indexes into the symbol table.
–V	Print the version of the strip command executing on the standard error output.

If there are any relocation entries in the object file and any symbol table information is to be stripped, *strip* will complain and terminate without stripping *file-name* unless the **–r** flag is used.

If the *strip* command is executed on an archive file (see *ar*(4)) the archive symbol table will be removed. The archive symbol table must be restored by executing the *ar*(1) command with the **s** option before the archive can be link-edited by the *ld*(1) command. *Strip* will instruct the user with appropriate warning messages when this situation arises.

The purpose of this command is to reduce the file storage overhead taken by the object file.

HARDWARE DEPENDENCIES

Series 800

The **–l** and **–x** options are synonymous, since the symbol table contains only static and external symbols. Either option causes only symbolic debugging information to be stripped. The **–r** option allows strip to be run on relocatable files, in which case the effect is also to strip only symbolic debugging information.

FILES

/usr/tmp/strp??????

SEE ALSO

ar(1), as(1), cc(1), ld(1), a.out(4), ar(4).

DIAGNOSTICS

strip: name: cannot open	if *name* cannot be read.
strip: name: bad magic	if *name* is not an appropriate object file.
strip: name: relocation entries present; cannot strip	if *name* contains relocation entries and the **–r** flag is not used, the symbol table information cannot be stripped.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

stty – set the options for a terminal port

SYNOPSIS

stty [**–a** | **–g** | options]

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 *termio*(7) for asynchronous lines. Options in the last group are implemented using options in the previous groups. Note that many combinations of options make no sense, but no sanity 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 *termio*(7)).

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 possible. (Some speeds are not supported by all hardware interfaces.)

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 *termio*(7)).

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 *termio*(7)).

nl0 nl1 select style of delay for line-feeds (see *termio*(7)).

tab0 tab1 tab2 tab3 select style of delay for horizontal tabs (see *termio*(7).

bs0 bs1 select style of delay for backspaces (see *termio*(7)).

ff0 ff1 select style of delay for form-feeds (see *termio*(7)).

vt0 vt1 select style of delay for vertical tabs (see *termio*(7)).

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.

tostop (**–tostop**) enable (disable) generation of SIGTTOU signals when background jobs attempt output. This flag is ignored on those systems which do not support job control.

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 *termio*(7)). For those systems which support job control, **susp** and **dsusp** characters may also be set. 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 *termio*(7)).

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**,

HARDWARE DEPENDENCIES

Series 200, 300, 500:

Job control is not supported. Refer to the HARDWARE DEPENDENCIES section of *termio*(7) for a further description of the capabilities that are not supported.

SEE ALSO

tabs(1), ioctl(2), termio(7).

INTERNATIONAL SUPPORT

8-bit data.

NAME

stty – set the options for a terminal port

SYNOPSIS

stty [**–a** | **–g** | options]

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 *termio*(7) for asynchronous lines. Options in the last group are implemented using options in the previous groups. Note that many combinations of options make no sense, but no sanity 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 *termio*(7)).

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 possible. (Some speeds are not supported by all hardware interfaces.)

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.

loblk (**–loblk**) enable (disable) layer output blocking.

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 *termio*(7)).

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 *termio*(7)).

nl0 nl1 select style of delay for line-feeds (see *termio*(7)).

tab0 tab1 tab2 tab3 select style of delay for horizontal tabs (see *termio*(7).

bs0 bs1 select style of delay for backspaces (see *termio*(7)).

ff0 ff1 select style of delay for form-feeds (see *termio*(7)).

vt0 vt1 select style of delay for vertical tabs (see *termio*(7)).

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.

tostop (–**tostop**) enable (disable) generation of SIGTTOU signals when background jobs attempt output. This flag is ignored on those systems which do not support job control.

Control Assignments

control-character c set *control-character* to *c*, where *control-character* is **erase**, **kill**, **intr**, **quit**, **eof**, **eol**, **swtch** (used with shell layers), **min**, or **time** (**min** and **time** are used with –**icanon**; see *termio*(7)). For those systems which support job control, **susp** and **dsusp** characters may also be set. 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 *termio*(7)).

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**,

HARDWARE DEPENDENCIES

Series 200, 300, 500:

Job control is not supported. Refer to the HARDWARE DEPENDENCIES section of *termio*(7) for a further description of the capabilities that are not supported.

SEE ALSO

tabs(1), ioctl(2), termio(7).

INTERNATIONAL SUPPORT

8-bit data.

NAME

su – become super-user or another user

SYNOPSIS

su [–] [name [arg ...]]

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 **root**). If the password is correct, *su* will execute a new shell with the real and effective user ID, real and effective group ID, and group access list set to that of the specified user. The new shell will be the optional program named in the shell field of the specified user's password file entry (see *passwd*(4)), or **/bin/sh** if none is specified (see *sh*(1)). To restore normal user ID privileges, type an **EOF** to the new shell.

Any additional arguments given on the command line are passed to the program invoked as the shell, permitting the super-user to run shell procedures with restricted privileges. When using programs like *sh*(1), an *arg* of the form **–c** *string* executes *string* via the shell and an arg of **–r** will give the user a restricted shell.

The following statements are true only if the optional program named in the shell field of the specified user's password file entry is like *sh*(1). If the first argument to *su* is a –, the environment will be changed to what would be expected if the user actually logged in as the specified user. This is done by invoking the program used as the shell with an *arg0* value whose first character is –, thus causing first the system's profile (**/etc/profile**) and then the specified user's profile (**.profile** in the new HOME directory) to be executed. Otherwise, the environment is passed along unchanged, except that **$PATH**, is unconditionally set to **/bin:/etc:/usr/bin** for **root**. Note that if the optional program used as the shell is **/bin/sh**, the user's **.profile** can check *arg0* for **–sh** or **–su** to determine if it was invoked by *login*(1) or *su*(1), respectively. If the user's program is other than **/bin/sh**, then **.profile** is invoked with an *arg0* of *-program* by both *login*(1) and *su*(1).

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**.

All attempts to become another user are logged in /usr/adm/sulog, including failures. Successful attempts are flagged with "+", failures with "-".

EXAMPLES

To become user **bin** while retaining your previously exported environment, execute:

```
su bin
```

To become user **bin** but change the environment to what would be expected if **bin** had originally logged in, execute:

```
su - bin
```

To execute *command* with the temporary environment and permissions of user **bin**, type:

```
su - bin -c "command args"
```

FILES

$HOME/.profile	user's profile

/etc/logingroup	system's default group access list file
/etc/passwd	system's password file
/etc/profile	system's profile
/usr/adm/sulog	log of all attempts

VARIABLES

HOME	the user's home directory
LOGNAME	the user's login name
PATH	the command name search path
PS1	the default prompt
SHELL	the name of the user's shell

SEE ALSO

env(1), login(1), sh(1), initgroups(3C), group(4), passwd(4), profile(4), environ(5).

NAME

sum – print checksum and block count of a file

SYNOPSIS

sum [**–r**] [file]

DESCRIPTION

Sum calculates and prints a 16-bit checksum for the named file, and also prints the number of blocks in the file. **Stdin** is used if no file 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.

DIAGNOSTICS

"Read error" is indistinguishable from end of file on most devices; check the block count.

SEE ALSO

wc(1).

NAME

tabs – set tabs on a terminal

SYNOPSIS

tabs [tabspec] [**+mn**] [**–T**type]

DESCRIPTION

Tabs sets the tab stops on the user's terminal according to the tab specification *tabspec*, after clearing any previous settings. The user's terminal must have 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*(5). If no **–T** flag is supplied, *tabs* searches for the **$TERM** value in the *environment* (see *environ*(5)). 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.

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.

SEE ALSO

nroff(1), pr(1), tset(1), environ(5), term(5).

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 64.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

tail – deliver the last part of a file

SYNOPSIS

tail [±[number][**lbc**[**f**]]] [file]

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 shown in this example:

```
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.

As another example, the command:

```
tail –15cf fred
```

will print the last 15 characters of the file **fred**, followed by any lines that are appended to **fred** between the time *tail* is initiated and killed.

The + option starts at the number indicated from the beginning of the file (rather than skipping the number of units indicated and then starting.) For example:

```
tail +1b fred
```

prints the entire contents of the file *fred*.

SEE ALSO

dd(1), head(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.

INTERNATIONAL SUPPORT

8-bit data and filenames.

NAME

tar – tape file archiver

SYNOPSIS

tar [*key*] [[*file* | -C *directory*] ...]

DESCRIPTION

Tar saves and restores files on magnetic tape or flexible disk. 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:

r The named *files* are added to the end of the archive. The **c** function implies this function.

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 a named file on tape does not exist on the system, the file is created as follows:

The user, group, and other protections are restored from the tape.
The modification time is restored from the tape unless the **m** option is specified.
The file owner and group owner are normally that of the restoring process.
The set-user-ID, set-group-ID and sticky bits are normally not set. The **o** and **p** options control the restoration of protection; see below for more details.

If the files exist, their modes are not changed except that the set-user-ID, set-group-ID and sticky bits are cleared. 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.

t The names of all the files on the archive are listed. Adding the **v** option will expand this listing to include the file modes and owner numbers. The names of all files are listed each time that they occur on the tape.

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.

c Create a new archive; writing begins at the beginning of the archive, instead of after the last file. This command implies the **r** function.

The following function modifiers may be used in addition to the function letters listed above:

#s Where **#** is a tape drive number (**0,...,7**), and **s** is the density (**l** - low (800 bpi), **m** - medium (1600 bpi), or **h** - high (6250 bpi)). This modifier selects the drive on which the 9 track tape is mounted. The default is **0m**.

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. With the **t** function, **v** gives more information about the tape entries than just the name.

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/0m.** If the name of the file is –, *tar* writes to the standard output or reads from the standard input, whichever is appropriate, and the default blocking factor becomes 1. 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. If both **f** and **b** modifiers are specified, their arguments must match the order in which they are specified. The default is 20; the maximum is at least 20. However, if the **f -** modifier is used, the default blocking factor is 1. The block size is determined automatically when reading 9 track tapes (key letters **x** and **t**). The blocking factor must be specified when reading flexible disks and cartridge tapes if they were written with a blocking factor different than the default.

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 For writing:

This option suppresses writing certain directory information that older versions of tar cannot handle on input. Tar normally writes information specifying owners and modes of directories in the archive. Former versions of tar, when encountering this information, will give error message of the form

"<name>/: cannot create".

This option will suppress writing that information.

For reading:

Causes extracted files to take on the user and group identifier of the user running the program rather than those on the tape. This is the default for the ordinary user, and may be overridden, to the extent the system protections allow, by the **p** option.

p This option causes files to be restored to the original modes and ownerships written on the archive, if possible. This is the default for the super-user, and may be overridden by the **o** option. For the ordinary user, if the system protections forbid the *chown*(2) operation needed to do this, the error will be ignored, and the ownership left with the restoring process. Set-user-ID, set-group-ID and sticky information will be restored as allowed by the protections defined by *chmod*(2), if the *chown* operation above succeeded.

The following option may be included in the file list:

-C*directory* *tar* will perform a *chdir*(2) to *directory*. This allows multiple directories not related by a close common parent to be archived using short relative path names.

If a 9 track tape drive is used as the output device, it must be configured in Berkeley compatibility mode; see *mt*(7).

EXAMPLES

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**).

tar cv -C /usr include -C / etc

This example archives files from /usr/include and from /etc.

ERRORS

Tar complains about bad key characters and tape read/write errors.

Tar complains if enough memory is not available to hold the link tables.

WARNINGS

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 disk or cartridge tape, and if the blocking factor specified on output was not the default, the same blocking factor must be specified on input. This is because the blocking factor is not explicitly stored on the archive. Not following this rule and updating the archive can destroy it.

The current limit on filename 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.

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 may be larger if the **b** option was used.

Note that **tar c0m** is not the same as **tar cm0**.

Archives should never be created on block special devices.

AUTHOR

Tar was developed by AT&T and the University of California, Berkeley.

FILES

/dev/rmt/*
/dev/rfd.*
/tmp/tar*

SEE ALSO

ar(1), cpio(1), mt(7).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

tbl – format tables for nroff

SYNOPSIS

tbl [**–TX**] [files]

DESCRIPTION

Tbl is a preprocessor that formats tables for *nroff*(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 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.

Full details of these and other features of *tbl* are given in the *tbl* reference guide.

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 (or if – is specified as the last argument), *tbl* reads the standard input, so it may be used as a filter. When it is used with *neqn*(1), *neqn*, *tbl* should come first to minimize the volume of data passed through pipes.

EXAMPLE

If we let <tab> represent a tab (which should be typed as a genuine tab), then the input:

```
.TS
center box ;
cB s s
cI | cI s
^ | c c
l | n n .
Household Population
_
Town<tab>Households
<tab>Number<tab>Size
=
Bedminster<tab>789<tab>3.26
Bernards Twp.<tab>3087<tab>3.74
Bernardsville<tab>2018<tab>3.30
Bound Brook<tab>3425<tab>3.04
Bridgewater<tab>7897<tab>3.81
Far Hills<tab>240<tab>3.19
.TE
```

yields:

Household Population		
Town	*Households*	
	Number	Size
Bedminster	789	3.26
Bernards Twp.	3087	3.74
Bernardsville	2018	3.30
Bound Brook	3425	3.04
Bridgewater	7897	3.81
Far Hills	240	3.19

SEE ALSO

cw(1), mm(1), neqn(1), nroff(1), mm(4).

BUGS

See **BUGS** under *nroff*(1).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

tcio – Command Set 80 Cartridge Tape Utility

SYNOPSIS

tcio –o[dervVZ] [**–S** buffersize] [**–l** number [**–n** limit]] filename
tcio –i[drvZ] [**–S** buffersize] [**–l** number [**–n** limit]] filename
tcio –u[rvV] [**–m** blocknumber] [**–l** number] filename

DESCRIPTION

Tcio is designed to optimize the data transfer rate between certain cartridge tape units and the host processor. When used in conjunction with other utilities (such as *cpio*(1)) a significant improvement in throughput can be obtained, in addition to reducing the wear and tear on the tape cartridges and drives. With autochanger mechanisms, *tcio* provides the capability of loading a specified cartridge, or automatically switching to successive cartridges as needed. With the utility operation, *tcio* provides functions that are unique to cartridge tapes.

Tcio –o (copy out) reads the standard input and writes the data to the Command Set 80 Cartridge Tape Unit specified by *filename*.

Tcio –i (copy in) reads the Command Set 80 Cartridge Tape Unit specified by *filename* and writes the data to the standard output.

Tcio –u (utility) performs utility functions on the cartridge tape, such as unload, mark, and/or verify the cartridge.

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* enables immediate report mode on cartridge tape units that support this mode (see HARDWARE DEPENDENCIES). During writing, this mode enables the drive to complete a write transaction with the host before the data has actually been written to the tape from the drive's buffer. This allows the host to start gathering data for the next write request while the data for the previous request is still in the process of being written. During reading, this mode enables the drive to read ahead after completing a host read request. This allows the drive to gather data for future read requests while the host is still processing data from the previous read request. Under favorable conditions, immediate report mode allows the drive to stream the tape continuously across multiple read/write requests, as opposed to having to reposition the tape between each read/write request. See ct(7) for further details.

By default, *tcio* puts a tape mark in the first block on each tape to prevent the tape from being image restored over a disk. It also utilizes the last block on each tape to flag whether or not the tape is the last tape in a multi-tape sequence.

The following command options are recognized. One of the options **–o**, **–i**, or **–u** must be specified. Additional options can be specified in any order, but all must precede the file name. Options without parameters can be listed individually or grouped together. Options with parameters require the parameter and must be listed individually. 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 that provides 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, the user must manually record and check it. The checksum is valid only 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 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.

–V This option turns off tape verification. Some cartridge tape units (see HARDWARE DEPENDENCIES) provide hardware for verifying the data output to the tape (called **read-while-write**). For these units software-driven verification is redundant, and this option is suggested.

For those drives that do not have the read-while-write hardware, a separate verification operation is suggested. Thus, it is recommended that this option not be used with drives that do not support read-while-write.

–r Unloads the tape from the drive. On autochanger units, the tape is returned to the magazine.

–S *buffersize* 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 4 will be silently increased to 4; a *buffersize* greater than 64 will be silently decreased to 64. If *buffersize* is not specified, *tcio* will allocate a 64 Kbyte buffer.

On tape units that support immediate report, a significant preformance increase can often be obtained by using a smaller buffer -- 8 Kbytes is the recommended buffer size for these units. On tape units that do not support the immediate report mode, or on tape units that are on a shared controller with a disk (see HARDWARE DEPENDENCIES) that is simultaneously being accessed, an increase in performance can usually be obtained with a larger buffer -- 64 Kbytes, the default, is the recommended buffer size for these units.

–m *blocknumber*
This option writes a tape mark on a tape at the specified block. A tape mark in block zero of the tape will prevent it from being image restored to a disk.

–Z This option prevents *tcio* from writing a file mark in the first and last blocks. This option should be used with care, as a tape without a tape mark in block zero can be image restored to a disk.

–l *number* This option is intended solely for autochanger-type tape units. With the input or output operations (**–i** or **–o**) the autochanger option selects the cartridge from the magazine with which the transfer will begin. When used with the utility function (**–u** option), *tcio* will load the specified cartridge into the drive. (Note: the autochanger must be in selective mode for the autochanger options to work properly.)

–n *limit* This option specifies the maximum number of cartridges to be used in a multi-tape transfer. It applies only to autochanger type units, and must be preceded by the **–l** option. Thus, **–l** starts the transfer by loading cartridge *number* and will use at most *limit* cartridges. If **–l** is specified without **–n**, *tcio* quietly assumes the remaining cartridges (in ascending order) from the magazine.

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/c0d1
tcio –i /dev/rct/c0d1 | cpio –i
```

To unload the cartridge from the drive (without verifying the tape) execute:

tcio -urV /dev/rct/c0d1

The next example copies all files in the current directory to the tape specified by the device file **/dev/rct/c1d0s2**. The device has a read-while-write head, so verify is turned off; a buffer size (option **-S**) of 8 blocks (i.e. 8 Kbytes) is specified:

ls | cpio -o | tcio -oV -S 8 /dev/rct/c1d0s2

The next example assumes that the cartridge tape unit is an autochanger, on controller 2, with 8 tapes in the magazine. The *tcio* operation will start writing with cartridge 3, and will use at most 4 cartridges before prompting the user for additional media:

find usr -cpio | tcio -oV -S 8 -l 3 -n 4 /dev/rct/c2

HARDWARE DEPENDENCIES

HP7941CT, HP9144A, and HP35401
: These cartridge tape devices support the immediate report mode.

HP7942, HP7946
: These cartridge tape devices support the immediate report mode. The use of a small buffer size is not recommended with these shared controller devices when there is simultaneous access to the disk, because the disk accesses will prevent proper tape streaming.

HP7908, HP7911, HP7912, and HP7914
: These cartridge tape devices do not support the immediate report mode.

AUTHOR

Tcio was developed by HP.

SEE ALSO

ct(7).

HP-UX System Administrator Manual.

INTERNATIONAL SUPPORT

8- and 16-bit data.

NAME

tcio – Command Set 80 Cartridge Tape Utility

SYNOPSIS

/etc/tcio –o [**dervSVC**] [buffersize] filename
/etc/tcio –i [**drvS**] [buffersize] filename
/etc/tcio –u [**cmrvV**] [blocknumber] [**save** | **restore**] filename [disc_filename]

Remarks:

This manual page describes the Series 500 implementation only. See other manual page for Series 200/300 implementation. Not supported on the Integral Personal Computer.

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.

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

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
```

The next example copies all files in the current directory (via executing *find*)*to* the tape specified by the device file */dev/rct*; a checksum (option **-d**) is performed to verify that the write to tape was performed correctly; verbose mode (**-v**) is used so that you can see the file names of files being copied; in addition, a buffer size (option **-S**) is specified at 128 memory blocks:

```
find . -print | cpio -o | tcio -odvS 128 /dev/rct
```

The following example copies all the files and directories from the tape (specified by */dev/rct*) to the current directory; the data is transfered through a 128-block buffer. Note that with the *cpio* command, the wildcard character * is inclosed in double quotes "*"; this must be done so that the shell doesn't expand the * to all the files in the current directory--i.e., you want the * to be interpreted as all the files on the tape, not your current directory. Here is the command:

```
tcio -ivS 128 /dev/rct |cpio -icdvu "*"
```

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] ...

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.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

test – condition evaluation command

SYNOPSIS

test expr
[expr]

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.
–p *file*	true if *file* exists and is a named pipe (fifo).
–u *file*	true if *file* exists and its set-user-ID bit is set.
–g *file*	true if *file* exists and its set-group-ID bit is set.
–k *file*	true if *file* exists and its sticky bit is set.
–s *file*	true if *file* exists and has a size greater than zero.
–t [*fildes*]	true if the open file whose file descriptor number is *fildes* (1 by default) is associated with a terminal device.
–z *s1*	true if the length of string *s1* is zero.
–n *s1*	true if the length of the string *s1* is 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.

WARNINGS

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.

SEE ALSO

find(1), sh(1).

NAME

time – time a command

SYNOPSIS

time command

DESCRIPTION

The *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.

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), timex(1), times(2).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

touch – update access, modification, and/or change times of file

SYNOPSIS

touch [**–amc**] [mmddhhmm[yy]] files

DESCRIPTION

Touch causes the access, modification, and last change times of each argument to be updated. The file name is created if it does not exist. 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).

INTERNATIONAL SUPPORT

8-bit filenames.

NAME

tput – query terminfo database

SYNOPSIS

tput [**–T** type] capname

DESCRIPTION

Tput uses the *terminfo* (4) database to make terminal-dependent capabilities and information available to the shell. *Tput* outputs a string if the attribute (**capa**bility **name**) is of type string, or an integer if the attribute is of type integer. If the attribute is of type boolean, tput simply sets the exit code (0 for TRUE, 1 for FALSE), and does no output.

-T*type* indicates the type of terminal. Normally this flag is unnecessary, as the default is taken from the environment variable **$TERM.**

Capname indicates the attribute from the *terminfo* database. See *terminfo* (4).

EXAMPLES

tput clear Echo clear-screen sequence for the current terminal.

tput cols Print the number of columns for the current terminal.

tput -Thp2623 cols
Print the number of columns for the hp2623 terminal.

bold=‘tput smso‘
Set shell variable "bold" to stand-out mode sequence for current terminal. This might be followed by a prompt:
echo "${bold}Please type in your name: \c"

tput hc Set exit code to indicate if current terminal is a hardcopy terminal.

FILES

/usr/lib/terminfo/?/* Terminfo data base
/usr/include/curses.h
/usr/include/term.h Definition files

DIAGNOSTICS

Tput prints error messages and returns the following error codes on error:

-1 Usage error.
-2 Bad terminal type.
-3 Bad capname.

In addition, if a capname is requested for a terminal that has no value for that capname (e.g., **tput -Thp2623 vt**), **-1** is printed.

SEE ALSO

stty(1), terminfo(4).

NAME

tr – translate characters

SYNOPSIS

tr [**–cds**] [string1 [string2]]

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), ascii(5).

BUGS

Will not handle ASCII **NUL** in *string1* or *string2*; always deletes **NUL** from input.

INTERNATIONAL SUPPORT

8- and 16-bit data.

NAME

true, false – provide truth values

SYNOPSIS

true

false

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

machid(1), 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 ...

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 files under the */usr/lib/terminfo* directory (see *terminfo*(4)). 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* for *sh*(1) users, or *.login* for *csh*(1) users) 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 **!**; **is a synonym for** **=** 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; users of *csh*(1) must also put a "\" before any "!" used.

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

```
eval `tset –s options...`
```

or using the C shell, *csh*(1):

set noglob; eval `tset –s *options...*

These commands cause *tset* to generate as output a sequence of shell commands which place the variable TERM in the environment; see *environ*(5).

Once the terminal type is known, *tset* engages in terminal 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 newline 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; in *sh*(1), the ^ character should be escaped (\^)).

–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 (@).

– report terminal type. Whatever type is decided on is reported. If no other flags are given, the only effect is to write the terminal type on the standard output.

–s generates appropriate commands (depending on your SHELL environment variable) to set TERM.

–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 the strings that would be assigned to TERM in the environment rather than generating commands for a shell. In *sh*(1), the following is an alternate way of setting TERM.

```
set -- `tset -S ...`
TERM=$1
```

–h forces a read of */etc/ttytype*. When **–h** is not specified, the terminal type is determined by reading the environment, unless some mapping is specified.

For compatibility with earlier versions of *tset*, the following flags are accepted, but their use is discouraged:

–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)). 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.

Assume, for the moment, that 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`
```

Now, 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;
/usr/lib/terminfo/?/*	terminal information data base.

VARIABLES

SHELL	if "csh" then generate *csh*(1) commands, otherwise generate *sh*(1) commands.
TERM	the (canonical) terminal name.

AUTHOR

Tset was developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

csh(1), sh(1), stty(1), ttytype(4), environ(5).

NAME

tsort – topological sort

SYNOPSIS

tsort [file]

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.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

tty – get the name of the terminal

SYNOPSIS

tty [**–s**]

DESCRIPTION

Tty prints the path name of the user's terminal. The **–s** option inhibits printing of the terminal path name, allowing one to test just the exit code.

RETURN VALUE

0 if standard input is a terminal,
1 otherwise.
2 if invalid options were specified,

DIAGNOSTICS

"not a tty" if the standard input is not a terminal and **–s** is not specified.

NAME

tty, pty - get the name of the terminal

SYNOPSIS

tty [**-s**]

pty [**-s**]

DESCRIPTION

Tty and *pty* print the path name of the user's terminal. The **-s** option inhibits printing of the terminal path name, and is used to test just the exit code.

RETURN VALUE

0	Standard input is a terminal if *tty* was invoked or a pseudo-terminal if *pty* was invoked.
1	Standard input is not a terminal or pseudo-terminal.
2	Invalid options were specified.

DIAGNOSTICS

not a tty	Standard input is not a terminal and **-s** was not specified in the tty command.
not a pty	Standard input is not a pseudo-terminal and **-s** was not specified in the pty command.

NAME

ul – do underlining

SYNOPSIS

ul [**–t** terminal] [**–i**] [name ...]

DESCRIPTION

Ul reads the named files (or standard input if none are given) and translates occurrences 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 *terminfo*(4) file corresponding to **TERM** 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.

FILES

/usr/lib/terminfo/?/* terminal capability files

AUTHOR

Ul was developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

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.

INTERNATIONAL SUPPORT

8-bit data and filenames.

NAME

umask – set file-creation mode mask

SYNOPSIS

umask [ooo]

DESCRIPTION

The user file-creation mode mask is set to *ooo*. The three octal 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

umodem – XMODEM-protocol file transfer program

SYNOPSIS

umodem [–options] files
umodem –c

DESCRIPTION

Umodem is a file transfer program that incorporates the well-known XMODEM protocol used on CP/M systems and on the HP110 portable computer.

Options

–1	Employ TERM II FTP 1.
–3	Enable TERM FTP 3 (CP/M UG).
–7	Enable 7-bit transfer mask.
–a	Turn on ARPA Net flag.
–c	Enter command mode.
–d	Do not delete *umodem.log* before starting.
–l	Turn on entry logging.
–m	Allow overwriting of files.
–p	Print all messages.
–r[t b]	Receive file. Specify **t** for text, or **b** for binary.
–s[t b]	Send file. Specify **t** for text, or **b** for binary.
–y	Display file status only.

EXAMPLES

To receive a text file:

umodem -rt7*file*

To receive a binary file:

umodem -rb*file*

To send a text file:

umodem -st7*file*

To send a binary file:

umodem -sb*file*

AUTHOR

Umodem is in the public domain.

SEE ALSO

kermit(1), cu(1), uucp(1).

NAME

uname – print name of current HP-UX version

SYNOPSIS

uname [**–snrvmia**]

DESCRIPTION

Uname prints the current system name of the HP-UX system on the standard output file. It is mainly useful to determine which system one is 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 may be a name that the system is known by open 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 nodename if the machine identification number cannot be ascertained.

–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 [–rSID] [–**s**] [–**n**] files

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 or the super-user).

FILES

p-file see *delta*(1).

g-file see *delta*(1).

SEE ALSO

delta(1), get(1), help(1), sact(1).

DIAGNOSTICS

Use *help*(1) for explanations.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

uniq – report repeated lines in a file

SYNOPSIS

uniq [**–udc** [+n] [–n]] [input [output]]

DESCRIPTION

Uniq reads the input file comparing 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). If the **–u** flag is used, just the lines that are not repeated in the original file are output. The **–d** option specifies that one copy of just the repeated lines is to be written. The normal mode output is the union of the **–u** and **–d** mode outputs.

The **–c** option 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.

The *n* arguments specify skipping an initial portion of each line in the comparison:

–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.

SEE ALSO

comm(1), sort(1).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

units – conversion program

SYNOPSIS

units [- file]

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
```

An alternate unit database file can be specified for use with the '- file' option. Units will look in this file rather than the default /usr/lib/unittab for the table of conversions. This must be in the same format as /usr/lib/unittab. This is useful in defining your own units and conversions.

WARNINGS

The monetary exchange rates are out of date.

FILES

/usr/lib/unittab

Series 500 Only

NAME

upm – unpack cpio archives from HP media

SYNOPSIS

upm –**E** [**cdmtuvx**] pathname chardevice [patterns]

upm –**iM** [**cdmtuvx**] [patterns] </dev/rmf?

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

uucp, uulog, uuname – UNIX system to UNIX system copy

SYNOPSIS

uucp [options] source-files destination-file

uulog [options]

uuname [**–l**] [**–v**]

DESCRIPTION

Uucp

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 which *uucp* knows about. The *system-name* may also be a list of names such as

system-name!system-name!...!system-name!path-name

in which case an attempt is made to send the file via the specified route, and only to a destination in PUBDIR (see below). Care should be taken to insure that intermediate nodes in the route are willing to foward information.

The shell metacharacters **?**, ***** and **[...]** 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*file* Report status of the transfer in *file*. If *file* is omitted, send mail to the requester when the copy is completed.

–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 Queue job but do not start the file transfer process. By default a file transfer process is started each time uucp is evoked.

–j Control writing of the *uucp* job number to standard output (see below).

Uucp associates a job number with each request. This job number can be used by *uustat* to obtain status or terminate the job.

The environment variable **JOBNO** and the **-j** option are used to control the listing of the *uucp* job number on standard output. If the environment variable **JOBNO** is undefined or set to **OFF**, the job number will not be listed (default). If *uucp* is then invoked with the **-j** option, the job number will be listed. If the environment variable **JOBNO** is set to **ON** and is exported, a job number will be written to standard output each time uucp is invoked. In this case, the **-j** option will supress output of the job number.

Uulog

Uulog queries a summary log of *uucp* and *uux*(1) transactions in the file **/usr/spool/uucp/LOGFILE.**

The options cause *uulog* to print logging information:

–s*sys* Print information about work involving system *sys*. If *sys* is not specified, then logging information for all systems will be printed.

–u*user* Print information about work done for the specified *user*. If *user* is not specified then logging information for all users will be printed.

Uuname

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. The **–v** option will print additional information about each system. A description will be printed for each system that has a line of information in **/usr/lib/uucp/ADMIN**. The format of ADMIN is: *sysname* tab *description* tab.

FILES

/usr/lib/uucp/*	other data and program files
/usr/spool/uucp	spool directory
/usr/spool/uucppublic	public directory for receiving and sending (PUBDIR)

SEE ALSO

mail(1), uux(1), chmod(2).

WARNING

The domain of remotely accessible files can (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.

NOTES

In order to send files that begin with a dot (e.g., .profile) the files must by qualified with a dot. For example: .profile, .prof*, .profil? are correct; whereas *prof*, ?profile are incorrect.

Uucp will not generate a job number for a strictly local transaction.

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.

The **-m** option will not work if all transactions are local or if **uucp** is executed remotely via the **-e** option.

The **-n** option will function only when the source and destination are not on the same machine.

Only the first seven characters of a *system-name* are significant. Any excess characters are ignored.

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.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames.

NAME

uustat – uucp status inquiry and job control

SYNOPSIS

uustat [options]

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:

–j*jobn* Report the status of the *uucp* request *jobn*. If **all** is used for *jobn*, the status of all *uucp* requests is reported. An argument must be supplied otherwise the usage message will be printed and the request will fail.

–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 that person is the super-user.

–r*jobn* Rejuvenate *jobn*. That is, *jobn* is touched so that its modification time is set to the current time. This prevents *uuclean* from deleting the job until the jobs modification time reaches the limit imposed by *uuclean*.

–c*hour* Remove the status entries which are older than *hour* hours. This administrative option can only be initiated by the user **uucp** or the super-user.

–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.

–M*mch* This is the same as the *–m* option except that two times are printed. The time that the last status was obtained and the time that the last successful transfer to that system occurred.

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.

–O Report the *uucp* status using the octal status codes listed below. If this option is not specified, the verbose description is printed with each *uucp* request.

–q List the number of jobs and other control files queued for each machine and the time of the oldest and youngest file queued for each machine. If a lock file exists for that system, its date of creation is listed.

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
```

will print the 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
000001	the copy failed, but the reason cannot be determined
000002	permission to access local file is denied

000004	permission to access remote file is denied
000010	bad *uucp* command is generated
000020	remote system cannot create temporary file
000040	cannot copy to remote directory
000100	cannot copy to local directory
000200	local system cannot create temporary file
000400	cannot execute *uucp*
001000	copy (partially) succeeded
002000	copy finished, job deleted
004000	job is queued
010000	job killed (incomplete)
020000	job killed (complete)

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/lib/uucp/L_stat	system status file
/usr/lib/uucp/R_stat	request status file
/usr/spool/uucp	spool directory

SEE ALSO

uucp(1).

NAME

uuto, uupick - public UNIX system to UNIX system file copy

SYNOPSIS

uuto [options] source-files destination
uupick [**-s** system]

DESCRIPTION

Uuto sends *source-files* to *destination*. *Uuto* uses the *uucp*(1) 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 *uuname* on *uucp*(1)). *Logname* 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 the PUBDIR for files sent from *system*.

FILES

PUBDIR/usr/spool/uucppublic public directory

NOTES

In order to send files that begin with a dot (e.g., .profile) the files must by qualified with a dot. For example: .profile, .prof*, .profil? are correct; whereas *prof*, ?profile are incorrect.

SEE ALSO

mail(1), uuclean(1M), uucp(1), uustat(1), uux(1).

NAME

uux – UNIX system to UNIX system command execution

SYNOPSIS

uux [options] command-string

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.

As an 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 will notify you if the requested command on the remote system was disallowed. The response comes by remote mail from the remote machine. The amount of mail notification can be reduced with the **–z** option, which notifies the remote system only if the command failed. Notification can be disabled totally with the **–n** option. Executable commands are listed in **/usr/lib/uucp/L.cmds** on the remote system. The format of the **L.cmds** file is:

cmd,machine1,machine2,...

If no machines are specified, then any machine can execute **cmd**. If machines are specified, only the listed machines can execute **cmd**. If the desired command is not listed in **L.sys** then no machine can execute that command.

Redirection of standard input and output is usually restricted to files in PUBDIR. Directories into which redirection is allowed must be specified in **/usr/lib/uucp/USERFILE** by the system administrator.

The following *options* are interpreted by *uux*:

– The standard input to *uux* is made the standard input to the *command-string*.

–n Send no notification to user.

-z Send notification only of failures to user.

-m*file*
Report status of the transfer in *file*. If *file* is omitted, send mail to the requester when the copy is completed.

-j Control writing of the *uucp* job number to standard output.

-r Queue job but do not start the file transfer process. By default a file transfer process is started each time uux is evoked.

Uux associates a job number with each request. This job number can be used by *uustat* to obtain status or terminate the job.

The environment variable **JOBNO** and the **-j** option are used to control the listing of the *uux* job number on standard output. If the environment variable **JOBNO** is undefined of set to **OFF**, the job number will not be listed (default). If *uuco* is then invoked with the **-j** option, the job number will be listed. If the environment variable **JOBNO** is set to **ON** and is exported, a job number will be written to standard output each time *uux* is invoked. In this case, the **-j** option will suppress output of the job number.

FILES

/usr/spool/uucp	spool directory
/usr/spool/uucppublic	public directory (PUBDIR)
/usr/lib/uucp/*	other data and programs

SEE ALSO

mail(1), uuclean(1M), uucp(1).

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.
Only the first seven characters of the *system-name* are significant. Any excess characters are ignored.

NAME

uxgen - generate an hp-ux system

SYNOPSIS

uxgen [-s] infile

DESCRIPTION

Uxgen is used to build an hp-ux system. The user supplies a set of instructions in *infile* which select optional parts of the kernel (eg. I/O drivers, pseudo-drivers) and specify values for system parameters such as the location of the swap area.

The files output by *uxgen* are placed in the directory *../infile*. This directory is created if it does not exist. Three files (conf.c, config.h, devices) are created by *uxgen*. The file "devices" contains a list of I/O devices, pseudo-drivers and major numbers assigned to them. This file is used by the commands *insf*(1), *mksf*(1) and *lssf*(1) for making and listing special files. In addition to *infile*, the file named "Makefile" must exist in the current directory. It will be copied to *../infile*. Makefile is supplied with the system and contains targets for compiling conf.c and linking the kernel (hp-ux).

After copying Makefile, *uxgen* changes the current directory to *../infile* and executes a "make". If the **-s** option is entered, then the make will not be executed. The make compiles conf.c and links the kernel (hp-ux). The file hp-ux may then be booted. See the System Administrators manual for information on how to boot the system.

Many header files are needed to compile conf.c. Also, an archive library file containing the kernel objects is needed to link the kernel. These files are supplied with the system and are contained in the directories found under /etc/conf. The directories in /etc/conf may be moved to any location in the filesystem; however, all the directories must exist to build the kernel. By convention, *infile* is placed in the directory named "gen" (eg. /etc/conf/gen).

The procedure for building a kernel follows:

1. switch current directory to "gen" (cd /etc/conf/gen)
2. edit or create *infile*
3. execute *uxgen* (*uxgen infile*)
4. New kernel is made and named *../infile*/hp-ux
5. Save old /hp-ux file.
6. Move *../infile*/hp-ux to /hp-ux.
7. Follow reboot instructions.

It is beyond the scope of this man page to give a complete description of the statements which may be used in *infile*. A "loose" syntax description for most statements is given below. See the Systems Administrators Manual for more information.

The statements used in *infile* form a simple "C-like" language. Before being read, *infile* is passed through the "C preprocessor". This allows features such as comments, macros, file inclusion and conditional statements. See *cpp*(1) for more information about the C preprocessor.

Generally, the first statement in *infile* is "#include /etc/master". This causes the statements in /etc/master to be read prior to the remaining statements in *infile*. /etc/master contains "driver" and "pseudo driver" statements which describe I/O drivers, pseudo-drivers and major number assignments for software supplied by HP. Only users who write kernel software (eg. drivers,

pseudo-drivers) need to understand these statements. They are described in the Systems Administrators manual. All other statements are briefly described below.

args on <module-id> **lu** <integer> **section** <integer> ;
Specifies the module name, logical unit number and section number to be used for writing the argument list during an "exec" system call.

acctresume <integer> ;
acctresume "<anychars>" ;
Specifies the percentage of file system space which must be free to allow process accounting to be reenabled after it has been suspended because of insufficient free space (see acctsuspend).

acctsuspend <integer> ;
acctsuspend "<anychars>" ;
Specifies the percentage of file system space which must be free to allow process accounting (see acctresume).

dst <integer> ;
Specifies whether daylight savings time is to be used. A value of 0 means not to use daylight savings time. A value of 1 indicates that USA daylight savings time is to be used.

include <identifier> ;
Causes a pseudo driver to be included in the kernel.

io {
<identifier> **lu** <integer> **address** <integer> ;
. . .
<identifier> **address** <integer> **{**
<identifier> **lu** <integer> **address** <integer> ;
}
. . .
}
Specifies how many I/O devices and how they are connected.

maxdsiz <integer> ;
maxdsiz "<anychars>" ;
Specifies the maximum process data segment size (in bytes).

maxssiz <integer> ;
maxssiz "<anychars>" ;
Specifies the maximum process stack size (in bytes).

maxtsiz <integer> ;
maxtsiz "<anychars>" ;
Specifies the maximum process shared text segment size (in bytes).

maxuprc <integer> ;
maxuprc "<anychars>" ;
Specifies the maximum number of processes that a user may have.

maxusers <integer> ;
maxusers "<anychars>" ;

Causes the macro MAXUSERS to be defined (eg. "#define MAXUSERS 8"). MAXUSERS is used to determine other tunable parameters (eg. nproc "(20 + 8 * MAXUSERS)";).

msgmap <integer> ;
msgmap "<anychars>" ;
Specifies the number of message map entries.

msgmax <integer> ;
msgmax "<anychars>" ;
Specifies the maximum number of bytes in a message.

msgmnb <integer> ;
msgmnb "<anychars>" ;
Specifies the maximum number of bytes for all messages which are queued on a message queue.

msgmni <integer> ;
msgmni "<anychars>" ;
Specifies the number of message queue identifiers.

msgseg <integer> ;
msgseg "<anychars>" ;
Specifies the number of units (each msgssz bytes long) available for messages.

msgssz <integer> ;
msgssz "<anychars>" ;
Specifies the size (in bytes) of each unit of memory used for messages (see msgseg).

msgtql <integer> ;
msgtql "<anychars>" ;
Specifies the number of message headers.

nbuf <integer> ;
nbuf "<anychars>" ;
Specifies the number of filesystem buffer cache buffer headers. If nbuf is set to 0, the kernel will allocate 10 percent of available memory to buffer space.

ncallout <integer> ;
ncallout "<anychars>" ;
Specifies the number of timeouts which may be pending.

nfile <integer> ;
nfile "<anychars>" ;
Specifies the maximum number of open files.

nflocks <integer> ;
nflocks "<anychars>" ;
Specifies the maximum number of file/record locks.

ninode <integer> ;
ninode "<anychars>" ;
Specifies the maximum number of open in-core inodes.

nproc <integer> ;
nproc "<anychars>" ;
Specifies the maximum number of processes which may simultaneously exist.

npty <integer> ;
npty "<anychars>" ;
Specifies the number of pty's (pseudo ttys).

ntext <integer> ;
ntext "<anychars>" ;
Specifies the maximum number of active shared text descriptors.

remove <identifier> ;
Removes a pseudo driver which was previously included with an include statement.

root on <module-id> **lu** <integer> **section** <integer>
Specifies module name, logical unit number and section number for the root of the file system ("/").

semaem <integer> ;
semaem "<anychars>" ;
Specifies the maximum value a semaphore may be adjusted due to a process dying.

semmap <integer> ;
semmap "<anychars>" ;
Specifies the number of semaphore map entries.

semmni <integer> ;
semmni "<anychars>" ;
Specifies the number of semaphore identifiers.

semmns <integer> ;
semmns "<anychars>" ;
Specifies the maximum number of semaphores.

semmnu <integer> ;
semmnu "<anychars>" ;
Specifies the maximum number of processes which can have pending "semaphore undo" requests on a semaphore.

semume <integer> ;
semume "<anychars>" ;
Specifies the maximum number of semaphores on which a process may have a pending "semaphore undo" request.

semvmx <integer> ;
semvmx "<anychars>" ;
Specifies the maximum value that a semaphore can be.

shmmax <integer> ;
shmmax "<anychars>" ;
Specifies the maximum number of bytes in a shared memory segment.

shmmni <integer> ;
shmmni "<anychars>" ;
Specifies the maximum number of shared memory segments.

shmseg <integer> ;
shmseg "<anychars>" ;
Specifies the maximum number of shared memory segments that can be simultaneously attached to a process.

swap on <module-id> **lu** <integer> **section** <integer>
[<module-id> **lu** <integer> **section** <integer>] ... ;
Specifies the module name, logical unit number and section number to be used for swapping. Multiple swap areas may be defined.

timeslice <integer> ;
timeslice "<anychars>" ;
Specifies the number of 10 millisecond intervals used for round-robin scheduling. A value of -1 disables round-robin scheduling.

timezone <integer> ;
Specifies the minutes west of Greenwich.

unlockable_mem <integer> ;
unlockable_mem "<anychars>" ;
Specifies the number of bytes of memory which cannot be locked.

EXAMPLES

```
args on         disc0 lu 0 section 1;
acctresume      4;
acctsuspend     2;
console on      mux0;
dst                     1;
dumps on                disc0 lu 0 section 1;
maxdsiz         0x8000;
maxssiz         0x1000;
maxtsiz         0x8000;
maxuprc                 25;
maxusers                32;
msgmap                  100;
msgmax          8192;
msgmnb                  16384;
msgmni          50;
msgseg          1024;
msgssz          8;
msgtql          40;
nbuf                    0;
ncallout        "(16 + NPROC)";
nfile           "(16*(NPROC+16+MAXUSERS)/10+32+2*NETSLOP)";
nflocks         200;
ninode          "((NPROC + 16 + MAXUSERS) +32)";
nproc           "(20 + 8 * MAXUSERS)";
npty                    60;
```

```
        ntext           "(24+MAXUSERS+NETSLOP)";
        root on         disc0 lu 0 section 0;
        semaem          16384;
        semmap          10;
        semmni          10;
        semmns          60;
        semmnu                  30;
        semume          10;
        semvmx          32767;
        shmmax                  0x4000000;
        shmmni          100;
        shmseg          12;
        swap on         disc0   lu 0 section 1
                                disc0   lu 1 section 1;
        timeslice               "(HZ/10)";
        timezone                8;
        unlockable_mem          0;

        io {
                cio_ca0 address 28 {
                        hpib0   address 0 {
                                disc0   lu 0 address 0;
                                disc0   lu 1 address 1;
                        }
                        mux0    lu 0 address 1;
                        hpib0   address 2 {
                                lpr0    lu 0 address 0;
                                tape0   lu 0 address 2;
                                instr0  lu 0 address 7;
                        }
                }
        }
```

AUTHOR

Uxgen was developed by HP.

FILES

/etc/devices(4)

SEE ALSO

insf(1), lssf(1), mksf(1).

NAME

val – validate SCCS file

SYNOPSIS

val –
val [–**s**] [–**r**SID] [–**m**name] [–**y**type] files

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.

–rSID 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 or **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 = cannot 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), help(1), prs(1).

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.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

vc – version control

SYNOPSIS

vc [–**a**] [–**t**] [–**c***char*] [–**s**] [keyword=value ... keyword=value]

DESCRIPTION

The *vc* command copies lines from the standard input to the standard output under control of its *arguments* and *control statements* encountered in the standard input. In the process of performing the copy operation, user declared *keywords* may be replaced by their string *value* when they appear in plain text and/or control statements.

The copying of lines from the standard input to the standard output is conditional, based on tests (in control statements) of keyword values specified in control statements or as *vc* command arguments.

A control statement is a single line beginning with a control character, except as modified by the –**t** keyletter (see below). The default control character is colon (:), except as modified by the –**c** keyletter (see below). Input lines beginning with a backslash (\) followed by a control character are not control lines and are copied to the standard output with the backslash removed. Lines beginning with a backslash followed by a non-control character are copied in their entirety.

A keyword is composed of 9 or less alphanumerics; the first must be alphabetic. A value is any ASCII string that can be created with *ed*(1); a numeric value is an unsigned string of digits. Keyword values may not contain blanks or tabs.

Replacement of keywords by values is done whenever a keyword surrounded by control characters is encountered on a version control statement. The –**a** keyletter (see below) forces replacement of keywords in *all* lines of text. An uninterpreted control character may be included in a value by preceding it with \. If a literal \ is desired, then it too must be preceded by \.

Keyletter Arguments

–**a** Forces replacement of keywords surrounded by control characters with their assigned value in *all* text lines and not just in *vc* statements.

–**t** All characters from the beginning of a line up to and including the first *tab* character are ignored for the purpose of detecting a control statement. If one is found, all characters up to and including the *tab* are discarded.

–**c***char* Specifies a control character to be used in place of :.

–**s** Silences warning messages (not errors) that are normally printed on the diagnostic output.

Version Control Statements

:dcl keyword[, ..., keyword]
Used to declare keywords. All keywords must be declared.

:asg keyword=value
Used to assign values to keywords. An **asg** statement overrides the assignment for the corresponding keyword on the *vc* command line and all previous **asg**'s for that keyword. Keywords declared, but not assigned values have null values.

:if condition
:
:

:end
Used to skip lines of the standard input. If the condition is true all lines between the *if* statement and the matching *end* statement are copied to the standard output. If the condition is false, all intervening lines are discarded, including control

statements. Note that intervening *if* statements and matching *end* statements are recognized solely for the purpose of maintaining the proper *if-end* matching. The syntax of a condition is:

```
<cond>      ::= [ "not" ] <or>
<or>        ::= <and> | <and> "|" <or>
<and>       ::= <exp> | <exp> "&" <and>
<exp>       ::= "(" <or> ")" | <value> <op> <value>
<op>        ::= "=" | "!=" | "<" | ">"
<value>     ::= <arbitrary ASCII string> | <numeric string>
```

The available operators and their meanings are:

=	equal
!=	not equal
&	and
|	or
>	greater than
<	less than
( )	used for logical groupings
not	may only occur immediately after the *if*, and when present, inverts the value of the entire condition

The > and < operate only on unsigned integer values (e.g., : 012 > 12 is false). All other operators take strings as arguments (e.g., : 012 != 12 is true). The precedence of the operators (from highest to lowest) is:

```
= != > <      all of equal precedence
&
|
```

Parentheses may be used to alter the order of precedence.
Values must be separated from operators or parentheses by at least one blank or tab.

::text

Used for keyword replacement on lines that are copied to the standard output. The two leading control characters are removed, and keywords surrounded by control characters in text are replaced by their value before the line is copied to the output file. This action is independent of the **-a** keyletter.

:on

:off

Turn on or off keyword replacement on all lines.

:ctl char

Change the control character to char.

:msg message

Prints the given message on the diagnostic output.

:err message

Prints the given message followed by:

ERROR: err statement on line ... (915)

on the diagnostic output. *Vc* halts execution, and returns an exit code of 1.

SEE ALSO

ed(1), help(1).

DIAGNOSTICS

Use *help*(1) for explanations.

EXIT CODES

0 – normal
1 – any error

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

vi – screen-oriented (visual) display editor based on ex

SYNOPSIS

vi [**–t** *tag*] [**–r** *file*] [**–l**] [**–w***n*] [**–x**] [**–R**] [**+***command*] name ...
view [**-t** *tag*] [**–r** *file*] [**–l**] [**–w***n*] [**–x**] [**–R**] [**+***command*] name ...
vedit [**–t** *tag*] [**–r** *file*] [**–l**] [**–w***n*] [**–x**] [**–R**] [**+***command*] name ...

REMARKS

The decryption facilities provided by this software are under control by the United States Government and cannot be exported without special licenses. These capabilities are only available by special arrangment with HP.

DESCRIPTION

Vi (visual) is a display-oriented text editor based on an underlying line editor *ex*(1). It is possible to use the command mode of *ex* from within *vi* and vice-versa.

When using *vi*, changes you make to the file are reflected in what you see on your terminal screen. The position of the cursor on the screen indicates the position within the file. Additional details on using *vi* and *ex* can be found in the *vi*and*ex* editor tutorials.

INVOCATION

The following invocation options are interpreted by *vi*:

–t *tag*	Edit the file containing the *tag* and position the editor at its definition.
–r*file*	Recover *file* after an editor or system crash. If *file* is not specified a list of all saved files will be printed.
–l	**LISP** mode; indents appropriately for lisp code, the () {} [[and]] commands in *vi* and *open* are modified to have meaning for *lisp* .
–w*n*	Set the default window size to *n*. This is useful when using the editor over a slow speed line.
–x	Encryption mode; a key is prompted for allowing creation or editing of an encrypted file.
–R	Read only mode; the **readonly** flag is set, preventing accidental overwriting of the file.
+*command*	The specified *ex* command is interpreted before editing begins.

The *name* argument indicates files to be edited.

The *view* invocation is the same as *vi* except that the **readonly** flag is set.

The *vedit* invocation is intended for beginners. The **report** flag is set to 1, and the **showmode** and **novice** flags are set. These defaults make it easier to get started learning the editor.

VI MODES

Command	Normal and initial mode. Other modes return to command mode upon completion. ESC (escape) is used to cancel a partial command.
Input	Entered by **a i A I o O c C s S R**. Arbitrary text may then be entered. Input mode is normally terminated with ESC character, or abnormally with interrupt.
Last line	Reading input for **:** **/** **?** or **!**; terminate with CR to execute, interrupt to cancel.

COMMAND SUMMARY

Sample commands

← ↓ ↑ →	arrow keys move the cursor
h j k l	same as arrow keys
i*text*ESC	insert text *abc*

cw*new*ESC	change word to *new*
ea*s*ESC	pluralize word
x	delete a character
dw	delete a word
dd	delete a line
3dd	delete 3 lines
u	undo previous change
ZZ	exit vi, saving changes
:q!CR	quit, discarding changes
/*text***CR**	search for *text*
^U ^D	scroll up or down
:*ex cmd***CR**	any ex or ed command

Counts before vi commands

Numbers may be typed as a prefix to some commands. They are interpreted in one of these ways.

line/column number	**z G |**
scroll amount	**^D ^U**
repeat effect	most of the rest

Interrupting, canceling

ESC	end insert or incomplete cmd
^?	(delete or rubout) interrupts
^L	reprint screen if ^? scrambles it
^R	reprint screen if ^L is → key

File manipulation

:wCR	write back changes
:qCR	quit
:q!CR	quit, discard changes
:e *name***CR**	edit file *name*
:e!CR	reedit, discard changes
:e + *name***CR**	edit, starting at end
:e +*n***CR**	edit starting at line *n*
:e #CR	edit alternate file
	synonym for **:e #**
:w *name***CR**	write file *name*
:w! *name***CR**	overwrite file *name*
:r *name***CR**	read file *name* into text
:<lineno>r *name***CR**	read file *name* into text starting at line <lineno>
:r !*cmd*	read output from *cmd* into text at cursor line
:shCR	run shell, then return
:!*cmd***CR**	run *cmd*, then return
:nCR	edit next file in arglist
:n *args***CR**	specify new arglist
^G	show current file and line
:ta *tag***CR**	to tag file entry *tag*
^]	**:ta**, following word is *tag*

In general, any *ex* or *ed* command (such as *substitute* or *global*) may be typed, preceded by a colon and followed by a **CR**.

Positioning within file

^F	forward screen
^B	backward screen
^D	scroll down half screen
^U	scroll up half screen
G	go to specified line (end default)
/pat	next line matching *pat*
?*pat*	prev line matching *pat*
n	repeat last / or ?
N	reverse last / or ?
/pat/+n	nth line after *pat*
?*pat***?–***n*	nth line before *pat*
]]	next section/function
[[	previous section/function
(	beginning of sentence
)	end of sentence
{	beginning of paragraph
}	end of paragraph
%	find matching ( ) { or }

Adjusting the screen

^L	clear and redraw
^R	retype, eliminate @ lines
zCR	redraw, current at window top
z–CR	... at window bottom
z.CR	... at window center
*/pat/z–CR**	*pat* line at window bottom
z*n***.**	use *n* line window
^E	scroll window down 1 line
^Y	scroll window up 1 line

Marking and returning

``	move cursor to previous context
''	... at first non-white in line
m*x*	mark current position with letter *x*
`*x*	move cursor to mark *x*
'*x*	... at first non-white in line

Line positioning

H	top line on screen
L	last line on screen
M	middle line on screen
+	next line, at first non-white
–	previous line, at first non-white
CR	return, same as +
↓ or **j**	next line, same column
↑ or **k**	previous line, same column

Character positioning

^	first non white
0	beginning of line
$	end of line
h or →	forward
l or ←	backwards
^**H**	same as ←
space	same as →
f*x*	find *x* forward
F*x*	**f** backward
t*x*	upto *x* forward
T*x*	back upto *x*
;	repeat last **f F t** or **T**
,	inverse of **;**
|	to specified column
%	find matching ({) or }

Words, sentences, paragraphs

w	word forward
b	back word
e	end of word
)	to next sentence
}	to next paragraph
(	back sentence
{	back paragraph
W	blank delimited word
B	back **W**
E	to end of **W**

Commands for LISP Mode

)	Forward s-expression
}	... but do not stop at atoms
(	Back s-expression
{	... but do not stop at atoms

Corrections during insert

^**H**	erase last character
^**W**	erase last word
erase	your erase, same as ^**H**
kill	your kill, erase input this line
****	quotes ^**H**, your erase and kill
ESC	ends insertion, back to command
^**?**	interrupt, terminates insert
^^**D**	("caret cntl-D") kill *autoindent* for one line
0^**D**	kill *autoindent* for duration of insertion
^**V**	quote non-printing character

Insert and replace

a	append after cursor
i	insert before cursor
A	append at end of line
I	insert before first non-blank
o	open line below
O	open above
r*x*	replace single char with *x*

R*text***ESC** replace characters

Operators

Operators are followed by a cursor motion, and affect all text that would have been moved over. For example, since **w** moves over a word, **dw** deletes the word that would be moved over. Double the operator, e.g. **dd** to affect whole lines.

d	delete
c	change
y	yank lines to buffer
<	left shift
>	right shift
!	filter through command
=	indent for LISP

Miscellaneous Operations

C	change rest of line (**c$**)
D	delete rest of line (**d$**)
s	substitute chars (**cl**)
S	substitute lines (**cc**)
J	join lines
x	delete characters (**dl**)
X	... before cursor (**dh**)
Y	yank lines (**yy**)

Yank and Put

Put inserts the text most recently deleted or yanked. However, if a buffer is named, the text in that buffer is put instead.

p	put back text after cursor
P	put before cursor
"*x***p**	put from buffer *x*
"*x***y**	yank to buffer *x*
"*x***d**	delete into buffer *x*

Undo, Redo, Retrieve

u	undo last change
U	restore current line
.	repeat last change
"*d***p**	retrieve *d*'th last delete

WARNINGS

On some small machines, *vi* does not support the full command set due to space limitations. The commands which are not supported are detailed in "An Introduction to Display Editing with Vi". The most notable commands which are missing are the macro and abbreviation facilities, and the *vedit* invocation. (Since arrow keys are done with macros, arrow keys do not work on such machines.)

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.

HARDWARE DEPENDENCIES

Series 500

(For HP27128A Asynchronous Serial Interface, HP27130A 8-Channel Asynchronous Multiplexor) On tty ports that support ENQ/ACK handshake and where that handshake is enabled ('stty ienqak'), the Control-F (^F, ASCII ACK) character is read by the interface and/or driver and discarded. In *vi*(1) this means that typing Control-F will not advance to the next page. Refer to *stty*(1) and *termio*(7) for information on alternate

handshakes, and how to disable ENQ/ACK.

AUTHOR

Vi and *ex* were developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

ex(1), edit(1).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

vis, inv – make unprintable characters in a file visible or invisible

SYNOPSIS

vis [**−n**] [**−s**] [**−t**] [**−u**] [**−x**] file ...
inv [**−n**] [**−s**] [**−t**] [**−u**] [**−x**] file ...

DESCRIPTION

Vis reads characters from each *file* in sequence and writes them to the standard output, converting those which are not printable into a visible form. *Inv* performs the inverse function, reading printable characters from each *file* and writing them, returned if appropriate to non-printable form, to standard out.

Non-printable characters are represented using C-like escape conventions:

****	backslash
\b	backspace
\e	escape
\f	form-feed
\n	new-line
\r	carriage return
\s	space
\t	horizontal tab
\v	vertical tab
*****n*	the 8-bit character whose ASCII code is the 3-digit octal number ***n***.
\x*n*	the 8-bit character whose ASCII code is the 2-digit hexadecimal number ***n***.

Space, horizontal tab, and new line may be treated as printable (and therefore passed unscathed to the output) or non-printable dependent on the options selected. Backslash, although printable, is expanded by *vis*, to a pair of backslashes so that when passed back through *inv*, it can be mapped back to a single backslash.

If no input file is given, or if the argument − is encountered, *vis* and *inv* read from the standard input file.

The options are:

−n causes new-line, space, and horizontal tab to be treated as non-printable characters. Thus *vis* expands them visibly as **\n**, **\s**, and **\t**, rather passing them directly to the output. *Inv* discards these character, expecting only the printable expansions. New-line characters are inserted by *vis* every 16 characters so that the output will be in form acceptable for most editors.

−s makes *vis* and *inv* 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.

−t treats horizontal tab and space as non-printable characters, in the same manner in which **−n** options treats them.

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

−x causes *vis* output to be in hexadecimal form rather than the default octal form. Either form is accepted to *inv* as input.

AUTHOR

Vis was developed by the Hewlett-Packard Company.

SEE ALSO

cat(1), echo(1), od(1).

WARNING

Command formats such as

```
vis file1 file2 >file1
```

will cause the original data in file1 to be lost.

INTERNATIONAL SUPPORT

8-bit and 16-bit data, customs, messages

NAME

vmstat – report virtual memory statistics

SYNOPSIS

vmstat [**–dfsSz**] [interval [count]]

DESCRIPTION

Vmstat normally reports certain statistics kept about process, virtual memory, trap and cpu activity. If given a **–d** argument, it also reports disk transfer information in the form of transfers per second. If given a **–f** argument, it instead reports on the number of *forks* and *vforks* since system startup and the number of pages of virtual memory involved in each kind of fork. If given a **–s** argument, it instead prints the contents of the *sum* structure, giving the total number of several kinds of paging related events that have occurred since boot. Giving a **–S** argument causes vmstat to execute normally with the exception that the number of processes swapped in and out are displayed instead of page reclaims and address translation faults. The **–z** option requires super-user capabilities. It clears all accumulators in the sum structure.

If none of these options are given, *vmstat* will report in the first line a summary of the virtual memory activity since the system has been booted. If *interval* is specified, then successive lines are summaries over the last *interval* seconds. The command **vmstat 5** will print what the system is doing every five seconds. This is a good choice of printing interval since this is how often some of the statistics are sampled in the system; others vary every second. If a *count* is given, the statistics are repeated *count* times. The format fields are:

Procs: information about numbers of processes in various states.

- r — in run queue
- b — blocked for resources (i/o, paging, etc.)
- w — runnable or short sleeper (< 20 secs) but swapped

Memory: information about the usage of virtual and real memory. Virtual pages are considered active if they belong to processes that are running or have run in the last 20 seconds.

- avm — active virtual pages
- free — size of the free list

Page: information about page faults and paging activity. These are averaged each five seconds, and given in units per second.

- re — page reclaims
- at — address translation faults
- pi — pages paged in
- po — pages paged out
- fr — pages freed per second
- de — anticipated short term memory shortfall
- sr — pages scanned by clock algorithm, per-second

Faults: trap/interrupt rate averages per second over last 5 seconds.

- in — (non clock) device interrupts per second
- sy — system calls per second
- cs — cpu context switch rate (switches/sec)

Cpu: breakdown of percentage usage of CPU time

us	user time for normal and low priority processes
sy	system time
id	cpu idle

AUTHOR

Vmstat was developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

FILES

/dev/kmem
/hp-ux

Series 500 Only

NAME

vstat – collect virtual memory performance statistics

SYNOPSIS

vstat command

Remarks:

Implemented only on the Series 500.

DESCRIPTION

Vstat uses *system*(3S) to execute the specified *command*. Upon completion, *vstat* prints the real, user, and system times for *command* plus counts of several Virtual Memory operations that occurred while *command* was being executed. Statistics are printed to stderr.

Virtual memory statistics are kept on a system-wide basis only, so **all** system Virtual Memory activity during the command's execution is reported by *vstat*.

Two kinds of logical memory objects are supported by the Series 500 processor: unpaged segments and paged segments. Unpaged segments are variable length objects that are typically used for code segments, stack segments, and some data segments. On the other hand, paged segments are divided into one or more pages of equal size, and can only be used for data. Paged segments are not limited to 512K bytes like unpaged segments. Thus, a paged segment can provide a large address space and efficient memory use because unused pages can be swapped to backing storage. While an unpaged segment provides faster memory access time, it must be entirely resident when accessed. This difference in memory efficiency can be illustrated by considering a program that accesses only the first two Kbytes of a 200-Kbyte array. If the array is allocated in a paged segment, only those pages that contain the first two Kbytes need be resident in memory. If the array is allocated in an unpaged segment, the entire 200 Kbyte segment must be resident in order to access any given part of the array.

Separate memory management algorithms are used for paged segments and unpaged segments. The system keeps statistics for the operation of each algorithm.

Statistics reported by *vstat* include:

time information	Process time information as in time(1).
page faults	Attempted accesses to addresses within absent pages. A page fault can be handled by recovering the page from the page cache, reading the page from backing storage, demand loading the page from the a.out file, or creating a new page in memory. The current page size for the system can be determined with uconfig(8).
page writes	Pages written to backing storage.
page reads	Pages read from backing storage.
page dloads	Pages demand loaded from an a.out file. This indicates that demand loading has been specified by chatr(1).
pages replaced	Pages moved from a working set to the page cache by the working set scan algorithm. Replaced pages are not immediately swapped to backing storage.
pages init'd	Pages created and initialized to all zeroes in memory.
normal scans	The number of working set scans performed by the paging system.
mr scans	The number of FIFO scans performed by the paging system. FIFO scans are an override of the normal working-set algorithm.

segment faults Attempted accesses to addresses within absent unpaged segments. A segment fault can be handled by reading the segment from backing storage, demand loading the segment from the a.out file, or creating a new segment in memory.

segment writes/bytes Segments/bytes written to backing storage.

segment reads/bytes Segments/bytes read from backing storage.

segment dloads/bytes Segments/bytes demand loaded from an a.out file. This indicates that demand loading has been specified by chatr(1).

segment inits/bytes Segments/bytes created and initialized to all zeroes in memory.

Several factors can keep the numerical totals from matching correctly. For example, simultaneous faults to shared code segments combined with the effects of the clustering algorithm in the paging system can prevent the total number of reads, demand loads, and inits from equalling the number of faults.

EXAMPLES

The command:

vstat ls -l /bin

creates a shell to execute 'ls -l /bin' and reports the resulting statistics to stderr.

The command:

vstat sleep 60

executes 'sleep 60' and has the effect of reporting on all system Virtual Memory activity over a 60 second period. Note that the times information is relevant only to the sleep command and is not interesting in this example.

The most practical use of *vstat* is to determine whether a program is virtual memory bound or CPU bound. Accurate determination requires that no other system activity occur during command execution. If user time plus system time (total CPU time) accounts for all or most of the elapsed time, the program is CPU bound and virtual memory activity is probably not a major factor. If elapsed time is greater than CPU time, the statistics for virtual memory I/O activity will show whether virtual memory is causing idle time. For page I/O, add the total number of page reads, page writes, and page dloads, then multiply by 30 msec. For segment I/O, add the number of segment reads, segment writes, and segment dloads, then multiply by 80 msec. If these times account for a significant portion of the idle time, the program is probably virtual memory bound. The numbers for page and segment I/O time are gross approximations. They are useful for general analysis, but should not be used to predict performance.

SEE ALSO

time(1), times(2), chatr(1), ld(1), uconfig(1M)
The section on Memory Management in the Concepts chapter of the HP-UX System Administrator Manual for the HP 9000 Series 500.

WARNINGS

Vstat requires an effective user id of *root* to execute.

Statistics include the creation of the shell by the system(3s) library call.

NAME

vt - login to another system over lan

SYNOPSIS

vt nodename [lan device]
vt **-p** [lan device]

DESCRIPTION

vt enables a user to log in on another HP 9000 system (*nodename*) over an HP local area network. The **-p** option will cause *vt* to send a poll request over the local area network to find out what systems currently have *vtdaemon(1M)* running. An asterisk (*) following a nodename in the response indicates that the system is a vt gateway. Plus signs (+) following the nodename indicate how many vt gateways have to be traversed to reach that system.

The optional argument *lan device* specifies a character special device name to use instead of the default device name to send/receive data to/from the local area network. The major number for this device must correspond to a CIO IEEE802.3 local area network device.

Once a connection has been established, *vt* enters input mode. In this mode, text typed is sent to the remote host. To issue *vt* commands when in input mode, precede them with the *vt* escape character. When in command mode, the normal terminal editing conventions are available.

The connection should automatically be terminated upon logging off of the remote machine. If the connection is not terminated then this indicates that the *ptydaemon* on the remote system has either been terminated or restarted. In this case the user should enter command mode and use the **quit** command to terminate the connection.

Commands

The following commands are available. Only enough of each command to uniquely identify it need be typed.

cd *remote-directory*
Change the working directory on the remote machine to *remote-directory*. This command is applicable for file transfer only.

escape [*escape-char*]
Set the *vt* escape character. If a character is not specified *vt* will prompt for one. If current *vt* escape character will be printed if the user just hits return in response to this prompt.

help

? Print a *vt* command summary.

lcd [*directory*]
Change the local working directory. If no *directory* is specified, use the user's home directory. This command is applicable for file transfer and shell escapes only.

get *remote-file local-file*

receive *remote-file local-file*
Retrieve the *remote-file* and store it on the local machine as *local-file*. *vt* will prompt for the file names if they are not specified. The file transfer can be aborted by typing the interrupt character or hitting the break key.

put *local-file remote-file*

send *local-file remote-file*
Retrieve the *local-file* and store it on the remote machine as *remote-file*. *vt* will prompt for the file names if they are not specified. The file transfer can be aborted by typing the interrupt character or hitting the break key.

quit Terminate the connection and exit *vt*.

user *user-name*[:[*password*]]
Identify yourself to the remote *vt* server. *vt* will prompt for a password (after disabling local echo) if a colon (:) is appended to *user-name*. It is necessary to do this before any file transfer command can be used.

! [*shellcommand*]
Shell escape. The given command is given to a sub-shell to execute. If no command is given, then a shell is started and connected to the user's terminal.

HARDWARE DEPENDENCIES
Series 500:

The HP 2285A lan device is not supported by *vt*.

FILES
/dev/ieee default lan device name.

SEE ALSO
vtdaemon(1M), lan(4)

DIAGNOSTICS
The diagnostics produced by *vt* are intended to be self-explanatory.

WARNINGS
vt uses the Hewlett-Packard **LLA** (Link Level Access) direct interface to the HP network drivers. *vt* uses the multicast address **0x01AABBCCBBAA.** It should not be used or deleted by other applications accessing the network. *vt* uses the following **IEEE 802.3** *sap* (service access point) values: **0x90, 0x94, 0x98, 0x9C, 0xA0, 0xA4, 0xA8, 0xAC, 0xB0, 0xB4, 0xB8, 0xBC, 0xC0, 0xC4, 0xC8, 0xCC, 0xD0 and 0xD4**. They should not be used by other applications accessing the network.

NAME

wait – await completion of process

SYNOPSIS

wait

DESCRIPTION

Wait until all processes started with **&** have completed, and report 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), wait(2)

BUGS

Not all the processes of a 3- or more-stage pipeline are children of the shell, and thus cannot be waited for.

NAME

wc – word, line, and character count

SYNOPSIS

wc [**–lwc**] [names]

DESCRIPTION

Wc counts lines, words, and characters in the named files, or in the standard input if no *names* 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 **l**, **w**, and **c** may be used in any combination to specify that a subset of lines, words, and characters are to be reported. The default is **–lwc**.

When *names* 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.

INTERNATIONAL SUPPORT

8-bit data, 8-bit filenames, messages.

NAME

what – identify files for SCCS information

SYNOPSIS

what [**–s**] files

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 SCCS command *get*(1), which automatically inserts identifying information, but it can also be used where the information is inserted manually. Only one option exists:

–s Quit after finding the first occurrence of pattern in each file.

SEE ALSO

get(1), help(1).

DIAGNOSTICS

Exit status is 0 if any matches are found, otherwise 1. Use *help*(1) for explanations.

BUGS

It is possible that an unintended occurrence of the pattern @ @(#) could be found just by chance, but this causes no harm in nearly all cases.

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit filenames, messages.

NAME

whereis – locate source, binary, and/or manual for program

SYNOPSIS

whereis [**–bsm**] [**–u**] [**–BMS** dir ... **–f**] name ...

DESCRIPTION

Whereis locates source/binary and manuals sections for specified files. The supplied 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 program in a list of standard places. If any of the **–b, –s** or **–m** flags are given then *whereis* searches only for binaries, sources or manual sections respectively (or any two thereof). The **–u** flag may be used to 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.

Finally, the **–B –M** and **–S** flags may be used to change or otherwise limit the places where *whereis* searches. The **–f** file flag is used to terminate the last such directory list and signal the start of file names.

EXAMPLES

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 *
```

WARNINGS

Since the program uses *chdir*(2) to run faster, pathnames given with the **–M –S** and **–B** must be full; i.e. they must begin with a "/".

FILES

/usr/src/*
/bin, /etc, /lib, /usr/{bin, games, lib}
/usr/man/*
/usr/local/{man/*, bin, games, include, lib}
/usr/contrib/{man/*, bin, games, include, lib}

AUTHOR

Whereis was developed by the University of California, Berkeley.

INTERNATIONAL SUPPORT

8-bit filenames.

NAME

which - locate a program file including aliases and paths

SYNOPSIS

which [name] ...

DESCRIPTION

Which takes a list of names and looks for the files which would be executed had these names been given as commands. Each argument is expanded if it is aliased, and searched for along the user's path. Both aliases and path are taken from the user's .cshrc file.

FILES

~/.cshrc source of aliases and path values

DIAGNOSTICS

A diagnostic is given for names which are aliased to more than a single word, or if an executable file with the argument name was not found in the path.

BUGS

Which reports **.cshrc** aliases even when not invoked from *csh*.

NAME

who – who is on the system

SYNOPSIS

who [**–uTlHqpdbrtas**] [file]

who am i

who am I

DESCRIPTION

Who can list the user's name, terminal line, login time, elapsed time since activity occurred on the line, and the process-ID of the command interpreter (shell) for each current system user. It examines the **/etc/utmp** file to obtain its information. If *file* is given, that file is examined. Usually, *file* will be **/etc/wtmp**, which contains a history of all the logins since the file was last created.

Who with the **am i** or **am I** option identifies the invoking user.

Except for the default **–s** option, the general format for output entries is:

name [state] line time activity pid [comment] [exit]

With options, *who* can list logins, logoffs, reboots, and changes to the system clock, as well as other processes spawned by the *init* process. These options are:

–u This option lists only those users who are currently logged in. The *name* is the user's login name. The *line* is the name of the line as found in the directory **/dev**. The *time* is the time that the user logged in. The *activity* is the number of hours and minutes since activity last occurred on that particular line. A dot (.) indicates that the terminal has seen activity in the last minute and is therefore "current". If more than twenty-four hours have elapsed or the line has not been used since boot time, the entry is marked old. This field is useful when trying to determine whether a person is working at the terminal or not. The *pid* is the process-ID of the user's shell. The *comment* is the comment field associated with this line as found in **/etc/inittab** (see *inittab*(4)). This can contain information about where the terminal is located, the telephone number of the dataset, type of terminal if hard-wired, etc.

–T This option is the same as the **–u** option, except that the *state* of the terminal line is printed. The *state* describes whether someone else can write to that terminal. A + appears if the terminal is writable by anyone; a – appears if it is not. **Root** can write to all lines having a + or a – in the *state* field. If a bad line is encountered, a **?** is printed.

–l This option lists only those lines on which the system is waiting for someone to login. The *name* field is **LOGIN** in such cases. Other fields are the same as for user entries except that the *state* field does not exist.

–H This option will print column headings above the regular output.

–q This is a quick *who*, displaying only the names and the number of users currently logged on. When this option is used, all other options are ignored.

–p This option lists any other process which is currently active and has been previously spawned by *init*. The *name* field is the name of the program executed by *init* as found in **/etc/inittab**. The *state*, *line*, and *activity* fields have no meaning. The *comment* field shows the *id* field of the line from **/etc/inittab** that spawned this process. See *inittab*(4).

-d This option displays all processes that have expired and not been respawned by *init*. The *exit* field appears for dead processes and contains the termination and exit values (as returned by *wait*(2)), of the dead process. This can be useful in determining why a process terminated.

-b This option indicates the time and date of the last reboot.

-r This option indicates the current *run-level* of the *init* process. The last three fields contain the current state of *init*, the number of times that state has been previously entered, and the previous state. These fields are updated each time *init* changes to a different run state.

-t This option indicates the last change to the system clock (via the *date*(1) command) by **root**. See *su*(1).

-a This option processes **/etc/utmp** or the named *file* with all options turned on.

-s This option is the default and lists only the *name*, *line*, and *time* fields.

FILES

/etc/inittab
/etc/utmp
/etc/wtmp

SEE ALSO

date(1), login(1), init(1), mesg(1), su(1), wait(2), inittab(4), utmp(4).

INTERNATIONAL SUPPORT

8-bit filenames.

NAME

whoami – print effective current user id

SYNOPSIS

whoami

DESCRIPTION

Whoami prints who you are. It works even if you are su'd, while 'who am i' does not since it uses /etc/utmp.

FILES

/etc/passwd Name data base

AUTHOR

Whoami was developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

who (1).

NAME

write – interactively write (talk) to another user

SYNOPSIS

write user [line]

DESCRIPTION

Write copies lines from your terminal to that of another user. When first called, it sends the message:

Message from *yourname* **(tty??)** [*date*]...

to the person you want to talk to. When it has successfully completed the connection, it also sends two bells to your own terminal to indicate that what you are typing is being sent.

The recipient of the message should *write* back at this point. Communication continues until an end of file is read from the terminal, an interrupt is sent, or the recipient has executed "mesg n". At that point *write* writes **EOT** on the other terminal and exits.

If you want to write to a user who is logged in more than once, the *line* argument may be used to indicate which line or terminal to send to (e.g., **tty00**); otherwise, the first writable instance of the user found in **/etc/utmp** is assumed and the following message posted:

user is logged on more than one place.
You are connected to "*terminal*".
Other locations are:
terminal

Permission to write may be denied or granted by use of the *mesg*(1) command. Writing to others is normally allowed by default. Certain commands, in particular *nroff*(1) and *pr*(1) disallow messages in order to prevent interference with their output. However, if the user has super-user permissions, messages can be forced onto a write-inhibited terminal.

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 them to *write* back before starting to send. Each person should end a message with a distinctive signal (i.e., **(o)** for "over") so that the other person knows when to reply. The signal **(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), nroff(1), pr(1), sh(1), who(1).

DIAGNOSTICS

user is not logged on	if the person you are trying to *write* to is not logged on.
Permission denied	if the person you are trying to *write* to denies that permission (with *mesg*).
Warning: cannot respond, set mesg -y	if your terminal is set to *mesg n* and the recipient cannot respond to you.
Can no longer write to user	if the recipient has denied permission (*mesg n*) after you had started writing.

INTERNATIONAL SUPPORT

8- and 16-bit data, messages.

NAME

xargs – construct argument list(s) and execute command

SYNOPSIS

xargs [flags] [command [initial-arguments]]

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 file, is searched for, using one's **$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 (\) will escape 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 will be 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: The user is 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) will execute 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 will terminate if either it receives a return code of **-1** from, or if it cannot execute, *command*. When *command* is a shell program, it should explicitly *exit* (see *sh*(1)) with an appropriate value to avoid accidentally returning with **-1**.

EXAMPLES

The following will move all files from directory $1 to directory $2, and echo each move command just before doing it:

```
ls $1 | xargs -i -t mv $1/{} $2/{}
```

The following will combine 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
```

The user is asked 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 will execute *diff*(1) with successive pairs of arguments originally typed as shell arguments:

```
echo $* | xargs -n2 diff
```

SEE ALSO

sh(1).

DIAGNOSTICS

Self-explanatory.

Series 800 Only

NAME

xdb - C, FORTRAN, and Pascal Symbolic Debugger

SYNOPSIS

xdb [**-d** *dir*] [**-r** *file*] [**-p** *file*] [*objectfile* [*corefile*]]

TABLE OF CONTENTS

DESCRIPTION

Xdb, is 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, (i.e. the -g flag). The support module */usr/lib/xdbend.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. 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.

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.

SMART TERMINAL SUPPORT

This is the user interface supported on most HP terminals. The top of the screen is a "window" into the current source file, and the bottom of the screen is for XDB and user program input and output. Separating the two areas is a line (in inverse video) indicating the current file, procedure, and line number. Within the source file window, a ">" points to the current location (which may or may not be the location at which the user program is currently stopped).

DUMB TERMINAL SUPPORT

This is the user interface on non-HP or un-supported terminal types. This does not support windows. The source file is displayed one line at a time. With this interface some commands supply more information then the "window" interface to compensate.

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 "*command* [*location*] [*command-arguments*] [*command-list*]". Numeric modifiers 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.

command-list 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**", "**i**" the abreviated {if} command, and "**!**" commands. In all other cases commands inside "**{}**" are ignored.

count The number of repetitions specified for a command.

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".

expr Any expression, but with limitations stated below.

file A file name.

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

line A *number* that refers to a particular line in a file.

location A particular *line* in a file (and its corresponding address in the user's program if there exists executable code for that line). *location* has the following general forms:

line
file [: (*line* | *#label*)]
proc [: *proc* [. . .]] [: (*line* | *#labe*

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

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

var A variable name.

Variable Name Conventions

Variables are referenced exactly as they are named in your source file(s). Case sensitivity is controlled by the "**Z**" command.

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 *var*.

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 *var*.

. *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. 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 "p $xyz = 3*4" creates special symbol "$xyz", and assigns to it the value 12. Special variables may be used just like any other variables.

$pc, **$sp**, **$r0**, etc.
These are the names of the program counter, the stack pointer, the Precision Architecture CPU 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.

$step 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.

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**|**L**|**l** [+|-]*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. The "**l**" and "**L**" exponent forms are allowed for compatibility with Pascal. However, all floating point constants are taken as doubles, regardless.

Character constants must be entered in '' and are treated as **integer**s. C string constants must be entered in "" and are treated like "**char ***" (e.g. pointer to **char**). FORTRAN and Pascal strings may be enclosed in either '' or "". 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. The debugger knows what langauge the object file (i.e. a.out), is written in and will switch to the language specific operators used by that langauge. The user may also create a composite program, built from 2 or more of the supported langauges, and debug without regard to which parts are which. The global variable $lang will be set as necessary by the debugger, based on the symbol table data supplied with the object file. The $lang var can be set to "C", "FORTRAN" and "Pascal" or "Default".

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, then set "p $lang = Pascal", you must say "p $lang := C" to return to C.

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().

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 buffer in the file */usr/lib/xdbend.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/xdbend.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:

p 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:

```
p 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/xdbend.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 *xdbend.c*.

Note that procedure name "_end_" is declared in *xdbend.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.

v View the source one window forward from the *current* source window. One or two lines from the previous window are preserved for context. If the "dumb" interface is in use only the next source line is displayed.

v [*location*]
View the source at the specified *location*, placing it in the center of the window. If the "dumb" interface is in use only the source line *location* is displayed.

V [*depth*]
View current procedure at *depth* in source window. *Depth* is an offset into the current procedure. If not specified the *depth* defaults to zero, which is where the program is currently stopped.

L Display the file name, procedure name, line number, and the current source statement corresponding to the object code being executed or examined. Allows user to determine where they are in program under test, and this is useful in assertion and breakpoint command lists.

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

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

w [*size*] Set size of source viewing window. Normally set to 15 lines for a 24 line terminal. If the "dumb" interface is being used, this command prints *size* lines centered around the current location.

/[*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, "p 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*(3S)). 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 and Modification Commands

p *expr* If *expr* does not look like anything else (such as a command), it is handled as if you had typed "**p** *expr*/**n**" (print expression in normal format), unless followed by "**;**" or "**}**", in which case nothing is printed. Note that modification of variables is done by using the assignment operator in the expression (ex. "p foo = 7" in C or FORTRAN, or "p foo := 7" in Pascal).

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

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

p -[[\] *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 "-".

p +[[\] *format*]
Go forward to the following 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 "+".

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** | **m** | **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), **m**acros, **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

t [*depth*] [*format*]
Trace the stack for the first *depth* (default 20) levels. Use the *format* if specified.

T [*depth*] [*format*]
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) Use the *format* if specified.

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* may contain shell variables and metacharacters, 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 it exists.

c [*location*]
Continue from breakpoint ignoring the signal. Set temporary breakpoint at specified *location.*

C [*location*]
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! Set temporary breakpoint at specified *location.*

s [*count*] 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**".

S [*count*]
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*(3S). 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 "**bc**" 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.

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:

lb 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.

b [*location*] [*count*] [*commands*]
Set a permanent breakpoint at the current location (or at *location*). Set the *count* number of times through breakpoint. 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).

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

db *

bp [*commands*]
Set permanent breakpoints at the beginning (first executable line) of every debuggable procedure. When any procedure breakpoint is hit, *commands* are executed.

It is permissible to set other permanent or temporary breakpoints at the same locations as these "procedure" breakpoints. If a procedure and non-procedure breakpoint are both hit at the same location, the non-procedure breakpoint has priority; the effect is the same as if there were no procedure breakpoint. It is not possible to alter the "count" of a procedure breakpoint. Procedure breakpoints must be activated and deleted as a group; it is not possible to set or delete individual ones.

Procedure breakpoints are useful for procedure stepping and tracing. For example, the command

```
bp Q;t 1;c
```

sets up procedure tracing by printing the current procedure at each breakpoint.

dp Delete all "procedure" breakpoints. All breakpoints set by commands other than "**bp**" will remain set.

bb [*depth*] [*count*] [*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.

bx [*depth*] [*count*] [*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.

bu [*depth*] [*count*] [*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. "**bu 0**" is a way to set a temporary breakpoint at the current value of **$pc**.

bt [(*depth* | *proc*)] [*count*] [*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.

ba *address* [*count*] [*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.

bc *number count*
Set count of breakpoint *number*, to *count*.

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.

aa *number*
Activate assertion *number.*

da *number*
Delete assertion *number.*

da * Delete all assertions, currently in effect.

sa *number*
Suspend assertion *number.*

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

x [*mode*]
Force an exit from assertions mode. If *mode* 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) {ta; x 1; c} {p 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 after the "**x**" command, the "**c**" command is not executed.

Another example:

```
a if (abc != $abc) {p $abc = abc;p  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.

z [*signal*] [**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 a "**lz**" command to list the current handling of all signals. Note that just "**z** *signal*" with no options tells you the state of the selected signal.

For example, assuming a start up state of (don't ignore, don't report, don't stop), the command "**z 14 sr**" sets the alarm clock signal (at least for HP-UX) to **stop** (but still don't **ignore**) and **report** that it occurred. Doing "**z 14 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, and the signal will not be passed to the child process. 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. Note: The file name can not be "**t**", "**f**", or "**c**", or begin with a "@".

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 ">@".

tr [@] Toggle recording [record-all]; if ON turn it OFF, if OFF turn it ON.

> 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.

Macro Facility

def *name* [*replacement-text*]
Defines *name* to be the macro whose value is *replacement-text*. *Name* may be any string of letters or digits. *Replacement-text* may be any string of letters, digits, blanks, tabs, or other printing characters, with the restriction that it may not be continued onto a new line. This command may not be abreviated as 1 char. It must be "def" or "define" only.

undef *name*
Remove the macro definition from *name* so that *name* no longer exists as a replacement string macro. As a special case "*" may be entered for *number* to undefine all macros. This command may not be abreviated as 1 char. It must be "undef" or "undefine" only.

tm Toggle the state of the macro substitution mechanism between active and suspended. When macro substitution is suspended, the currently defined macros continue to exist, but they are not replaced in the command-line by their definitions. Additional macros may still be defined while macro substitution is suspened.

Miscellaneous Commands

sm Suspends the "more" facility so that debugger output is no longer pagenated. This is most useful when breakpoints or assertions will be printing a great deal of information to the screen, and you do not want the debugger to keep waiting for you to hit the space bar.

am Activates the "more" facility so that debugger output is pagenated.

td Toggles dissassembly mode. When in dissassembly mode, the source window displays the code in assembly language, and the single step command steps assembly instruction at a time. The display consists of: the source line number, the address in hex, the address in the form of nearest label plus offset, and the assembly instruction. If the debugger is

the form of nearest label plus offset, and the assembly instruction. If the debugger is already in split screen mode, then the td command changes the level of single stepping, but has no change on the display other then the mode displayed in the line seperarating the source statements from the assembly instructions.

ts Toggles split screen mode. When in split screen mode, the source window is half source code and half assembly instructions. In split screen mode, the td command still toggles the debugger between symbolic (or source) mode and assembly mode, as indicated by the line seperating the source from the assembly. The only difference is whether single stepping is at the source statement or assembly instruction level.

<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**.

! [*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*(3S). 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,

```
b 14 {!{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.

[*text*] Flags this *text* as a comment to be echoed to the command window. The # must appear as the first non-blank character on the line and the remainder of the line is treated as a comment. It is also written to the currently open record file.

D *dirs* Adds *dirs* to the list of additional directory search paths for source files. This command is equivalent to the command-line option -d.

f ["*printf-style-format*"]
Set address printing format, using *printf*(3S) 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.

g *line* | *#label*
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* or the line *#label* appears on 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.

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

tc Toggle case sensitivity in searches. This affects everything: File names, procedure names, variables, and string searches!

SYMBOL TABLE DEPENDENCIES

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.

FORTRAN ENTRY points are flagged "ENTRY" by the "**l p**" command.

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".

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.

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/xdb.help	Text file listed by the "**help**" command.
/usr/lib/xdb.error	Text file which explains debugger error and warning messages.
/usr/lib/xdbend.o	Object file to link with all debuggable programs.
/usr/lib/xdbend.c	Source file for xdbend.o.

SEE ALSO

cc(1), echo(1), ld(1), more(1), creat(2), exec(2), fork(2), open(2), printf(3S), system(3S), a.out(4).

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/xdb.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 not a code address in the child process, strange results or errors may ensue.

If you set the address printing format to something *printf*(3S) 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*(3C), 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".

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. User should use $sizeof which works in any language.

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**", "**mod**", "**addr**", and "**sizeof**" are added.

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, such as (int) or (char).

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*).

XDB does not support identically-named procedures (legal in Pascal if the procedures are in different scopes). XDB will always use the first procedure with the given name.

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.

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

yacc – yet another compiler-compiler

SYNOPSIS

yacc [**–vdlt**] [**–N**<*secondary*><*n*> ...] grammar

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*.

If the **–v** flag is given, the file **y.output** is prepared, which contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.

If the **–d** flag is used, the file **y.tab.h** is generated with 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.

If the **–l** flag is given, the code produced in **y.tab.c** will not contain any **#line** constructs. Generally, this should only be used after **y.tab.c** has compiled successfully, since the **#line** directives allow the C compiler to give error messages that refer to the *yacc* source file rather than the **y.tab.c** file. This option is useful, however, for symbolic debugging, since some symbolic debuggers may be confused by line numbers that are not in order.

The **–N**<*secondary*><*n*> option allows the sizes of certain internal *yacc* tables to be reset. *Secondary* is one of the letters from the set { **a m s p n e c l w** } and specifies the table; *n* is the new size. Tables that can be reset by using secondary letters are as follows:

a	a-array size; default is 12 000.
m	mem array size; default is 12 000.
s	number of states; default is 1000.
p	number of productions; default is 800.
n	number of non-terminals; default is 600.
e	temp-space size; default is 1250.
c	name-space size; default is 5000.
l	look-ahead set table size; default is 650.
w	working set table size; default is 650.

If an array overflows, *yacc* issues a fatal error message including a suggestion of which table to reset. For example:

```
too many states, try –Ns option
```

Runtime debugging code is always generated in **y.tab.c** under conditional compilation control. By default, this code is not included when **y.tab.c** is compiled. However, when *yacc*'s **–t** option is used, this debugging code will be compiled by default. Independent of whether the **–t** option was used, the runtime debugging code is under the control of **YYDEBUG**, a pre-processor symbol. If **YYDEBUG** has a non-zero value, then the debugging code is included. If its value is zero, then the code will not be included. The size and execution time of a program produced without the runtime debugging code will be smaller and slightly faster.

ERRORS

The number of reduce-reduce and shift-reduce conflicts is reported on the standard error 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.

FILES

y.output	
y.tab.c	
y.tab.h	defines for token names
yacc.tmp,	
yacc.acts, yacc.debug	temporary files
/usr/lib/yaccpar	parser prototype for C programs

WARNINGS

Because file names are fixed, at most one *yacc* process can be active in a given directory at a time.

The maximum number of terminal symbols is fixed at 2000 and cannot be reset using the –N option.

SEE ALSO

lex(1), malloc(3X).

LR Parsing by A. V. Aho and S. C. Johnson, Computing Surveys, June, 1974.

YACC - Yet Another Compiler Compiler in *HP-UX Concepts and Tutorials.*

INTERNATIONAL SUPPORT

8-bit data and filenames.

NAME

intro – introduction to glossary section

DESCRIPTION

This section contains a glossary of common HP-UX terms. References to other HP-UX documentation are included as appropriate. References to entities such as *wait*(2), *sh*(1), or *fopen*(3S) refer to entries in the other sections of this manual. References to items in italics but having no parenthetical suffixes refer to other entries in this glossary. Finally, any references to italicized manuals refer to separate manuals that are included with your system or are available for purchase.

SEE ALSO

The introduction to this manual.

The glossary section of the *Documentation and Terminology Guide*.

.o ("dot-oh") The format of an unlinked object file. See *a.out*.

absolute path name
A path name beginning with a slash (/). It indicates that the file's location is given relative to the root directory (/), and that the search begins there.

access Access to system resources is governed by three values: the effective user ID, the effective group ID, and the group access list.

access groups The group access list is an additional set of group ID's used only in determining resource accessibility. Access checks are performed as described below in file access permissions.

address In the context of peripheral devices, a set of values which specify the location of an I/O device to the computer. The exact details of the formation of an address differ between systems. On the Series 300 and 500, the address is composed of up to four elements: the select code, bus address, unit number (id), and volume number (id).

affiliation See *terminal affiliation.*

a.out The format of executable object code files on HP-UX. The format is machine-dependent, and is described in the *a.out*(4) for each implementation. Object code which is as yet unlinked is in the same format, but is referred to as a *.o* ("dot-oh") file. *A.out* is also the default output file name used by the linker, *ld*(1).

archive A file which is made up of the contents of other files (such as a group of object (i.e. *.o*) files to be used by the linker, *ld*(1)). An archive file is created and maintained by *ar*(1), or by similar programs, such as *tar*(1) or *cpio*(1). An archive is often called a library.

ASCII An acronym for American Standard Code for Information Interchange. It consists of a set of characters including letters, numerals, punctuation, and control characters, each of which is represented internally by 7 bits (0 - 127).

asynchronous I/O
An I/O operation for which the user process need not wait for completion before continuing execution.

backup The process of making a copy of all or part of the file system in order to preserve it should a system crash occur (usually due to a power failure, hardware error, etc.). This is a highly recommended practice.

block (1) The fundamental unit of information HP-UX uses for access and storage allocation on a mass storage medium. The size of a block varies between implementations, and between file systems. In order to present a more uniform interface to the user, most system calls and utilities use "block" to mean 512 bytes, independent of the actual block size of the medium. This is the meaning of "block" unless otherwise specified in the manual entry.

(2) On media such as 9 track tape which write variable length strings of data, the size of those strings. Block is often used to distinguish from *record* with a block containing several records, with the number of records being the *blocking*factor.

block special file A special file associated with a mass storage device (such as a hard disk or a CS-80 tape cartridge drive) that transfers data in units of blocks. Block special files may be mounted.

boot or *boot-up* The process of loading, initializing, and running an operating system.

boot area On the Series 300, a portion of a mass storage medium (block zero) on which the volume header and a small "bootstrap" program used in booting the operating system reside. The boot area is reserved exclusively for use by HP-UX.

On the Series 500, the portion of an SDF mass storage medium which contains an operating system.

boot ROM On the Series 300, a program residing in ROM (Read Only Memory) that executes each time the computer is powered-up. The function of the boot ROM is to run tests on the computer's hardware, find all devices accessible through the computer, and then load either a specified operating system or the first operating system found according to a specific search algorithm.

The Series 500 computers have a program that is identical in function, but differs in implementation. See *system loader*.

bus address A number which makes up part of the address HP-UX uses to "find" a particular device. The *bus address* is determined by a switch setting on a peripheral device which allows the computer to distinguish between two devices connected to the same interface. A bus address is sometimes called a "device address".

CS/80 or *CS-80* A family of mass storage devices that communicate via the *CS/80* (Command Set '80) command set. Examples are the HP 7908, HP 7911, HP 7912, and HP 7914 disk/tape drives.

character special file
A special file associated with devices which transfer data character-by-character. Examples are printers, terminals, nine-track magnetic tapes, and disks accessed in "raw" mode (see *raw disk*).

child process A new process created by a pre-existing process via the *fork*(2) system call. The new process is thereafter known to the pre-existing process as its *child process*. The pre-existing process is the *parent process* of the new process. See *parent process* and *fork*.

command A stand-alone unit of executable code (a program), or a file containing a list of other programs to execute in order (a shell script). In HP-UX, commands are executed through a command interpreter called a shell, often *sh*(1). Arguments following the command name are passed on to the command program. You can write your own commands, either as executable programs, or as shell scripts (written in the shell programming language).

command interpreter
A program which reads lines of text from standard input (typed at the keyboard or redirected from a file), and interprets them as requests to execute other programs. A command interpreter for HP-UX is called a shell. See *sh*(1) and *csh*(1).

control character
A member of a character set which produces action in a device other than printing or displaying a character. In the ASCII character set, control characters are those in the range 0 through 31, and 127. Control characters can be generated by holding down [CRTL], [CONTROL], or [CNTL] (depending on what the control key is labeled on your terminal) and pressing a character key (as you would use SHIFT). These two-key sequences are often written as ctrl-d, for example, or ^D, where ^ stands for the control key. Both representations assume that the control key is held down while the second key is pressed.

crash The unexpected shutdown of a program or system. If the operating system crashes, this is a "system crash", and requires the system to be re-booted.

current directory
See *working directory.*

current working directory
See *working directory.*

daemon A process which runs in the background, and which is usually immune to termination instructions from a terminal. Its purpose is to perform various scheduling, clean-up, and maintenance jobs. *Lpsched*(1M) is an example of a daemon that exists to perform these functions for line printer jobs queued by *lp*(1). An example of a permanent daemon (i.e. it never should die) is *cron*(1M).

data encryption A method for encoding information in order to protect sensitive or proprietary data. For example, all users' passwords are automatically encrypted by HP-UX. The encryption method used by HP-UX converts ASCII text into a base-64 representation using the alphabet ., /, 0-9, A-Z, a-z. See *passwd*(4) for the numerical equivalents associated with this alphabet.

default search path
The sequence of directory prefixes that *sh*(1), *time*(1), and other HP-UX commands apply in searching for a file known by an incomplete path name (i.e. a path name not beginning with a slash, /). It is defined by the environment variable **PATH** (see *environ*(5)). *Login*(1) sets **PATH** equal to **:/bin:/usr/bin**, which means that your working directory is the first directory searched, followed by **/bin**, followed by **/usr/bin**. You can redefine the search path by modifying the value of **PATH**. This is usually done in **/etc/profile**, and/or in the **.profile** file found in your home directory.

delta A term used in the Source Code Control System (SCCS) to describe a unit of one or more textual changes to an SCCS file. Each time you edit an SCCS file, the changes you make to the file are stored separately as a delta. Then, using the *get*(1) command, you can specify which deltas are to be applied to or excluded from the SCCS file, thus yielding a particular version of the file. Contrast this with the *vi* or *ed* editor, which incorporates your changes into the file immediately, prohibiting you from obtaining a previous version of that file. See *SCCS, SCCS file.*

demon See *daemon.*

device file See *special file.*

directory A file which provides the mapping between the names of files and their contents. For every file name contained in a directory, that directory contains a pointer to the file's *inode* called a *link*. A file may have several links appearing anywhere on the same file system. Each user is free to create (using *mkdir*(1)) as many directories as needed, providing that the parent directory of the new directory gives the permission to do so. Once a directory has been created, it is ready to contain ordinary files and other directories. An HP-UX directory is named and behaves exactly like an ordinary file, with one exception: no user (including the super-user) is allowed to write data on the directory itself; this privilege is reserved for the HP-UX operating system.

By convention, a directory contains at least two links, . and .., referred to as *dot* and *dot-dot* respectively. Dot refers to the directory itself and dot-dot refers to its parent directory. For purposes of deletion, a directory containing only . and .. is considered empty.

effective group ID

Every process has an effective group ID that is used to determine file access permissions. A process's effective group ID is determined by the file (command) that process is executing. If that file's set-group-ID bit is set (located in the mode of the file, see *mode*), the process's effective group ID is set equal to the file's group ID. This makes the process appear to belong to the file's group, perhaps enabling the process to access files which must be accessed in order for the program to execute successfully. If the file's set-group-ID bit is not set, the process's effective group ID is inherited from the process's parent. The setting of the process's effective group ID lasts only as long as the program is being executed, after which the process's effective group ID is set equal to its real group ID. See *group*, *real group ID*, and *set-group-ID bit*.

effective user ID A process has an effective user ID that is used to determine file access permissions (and other permissions with respect to system calls, if the effective user ID is 0, which means super-user). A process's effective user ID is determined by the file (command) that process is executing. If that file's set-user-ID bit is set (located in the mode of the file, see *mode*), the process's effective user ID is set equal to the file's user ID. This makes the process appear to be the file's owner, enabling the process to access files which must be accessed in order for the program to execute successfully. (Many HP-UX commands which are owned by *root*, such as *mkdir* and *mail*, have their set-user-ID bit set so other users can execute these commands.) If the file's set-user-ID bit is not set, the process's effective user ID is inherited from that process's parent. The setting of the process's effective user ID lasts only as long as the program is being executed, after which the process's effective user ID is set equal to its real user ID. See *real user ID* and *set-user-ID bit*.

environment The set of defined shell variables (some of which are PATH, TERM, SHELL, EXINIT, HOME) which define the conditions under which your commands run. These conditions can include your terminal characteristics, your home directory, and your default search path. Each shell variable setting in the current process is passed on to all child processes that are created, provided that each shell variable setting has been exported via the *export* command (see *sh*(1)). Unexported shell variable settings are meaningful only to the current process, and any child processes created are given the default settings given certain shell variables in **/etc/profile** and/or **$HOME/.profile**.

end-of-file (1) the data returned when attempting to read past the logical end of a file via *stdio*(3S) routines. In this case end-of-file is not properly a character.

(2) The character [CTRL]-[D].

(3) A character defined by *stty*(1) or *ioctl*(2) (see *termio*(7)) to act as end-of-file on your terminal. Usually this is [CTRL]-[D].

(4) The indication (as the function result) which indicates end of data when using *read*(2).

file An HP-UX file is simply a stream of bytes representing ASCII text (text files) or binary data (such as executable code). Thus, directories, ordinary files, special files, etc., can all be considered files. Every file must have a file name (see *file name*) which enables the user (and many of the HP-UX commands) to reference the contents of the file. The size of a file is exactly the number of bytes the file contains. The system imposes no particular structure on the contents of a file

(although some programs do). Files may be accessed serially or randomly (indexed by byte offset). The interpretation of file contents and structure is up to the programs that access the file.

file access permissions

Every file in the file system has a set of access permissions. These permissions are used in determining whether a process may perform a requested operation on the file (such as opening a file for writing). Access permissions are established at the time a file is created. They may be changed at some later time through the *chmod*(2) call.

File access is determined according to whether a file may be read, written, or executed. Directory files use the execute permission to control whether or not the directory may be searched.

File access permissions are interpreted by the system as they apply to three different classes of users: the owner of the file, those users in the file's group, anyone else. Every file has an independent set of access permissions for each of these classes. When an access check is made, the system decides if permission should be granted by checking the access information applicable to the caller.

Read, write, and execute/search permissions on a file are granted to a process if:

The process's effective user ID is super-user.

The process's effective user ID matches the user ID of the owner of the file and the appropriate access bit of the "owner" portion (0700) of the file mode is set.

The process's effective user ID does not match the user ID of the owner of the file, and either the process's effective group ID matches the group ID of the file, or the group ID of the file is in the process's group access list, and the appropriate access bit of the "group" portion (070) of the file mode is set.

The process's effective user ID does not match the user ID of the owner of the file, and the process's effective group ID does not match the group ID of the file, and the group ID of the file is not in the process's group access list, and the appropriate access bit of the "other" portion (07) of the file mode is set.

Otherwise, the corresponding permissions are denied.

file descriptor A small integer identifier that is used to refer to a file that has been opened for reading and/or writing, and is an index into the user's table of open files. The opened file must be identified by its file descriptor when using system calls to read or write the file.

The value of a file descriptor has a range from 0 to a system defined maximum. For systems at HP-UX STANDARD and above, the minimum value for this number is 60. For systems below HP-UX STANDARD the minimum value is 20. No file descriptor may have a value outside the range 0-59 or 0-19, depending on the implementation.

A file descriptor is obtained through system calls such as *open*(2), *creat*(2), *dup*(2), *fcntl*(2) or *pipe*(2). The file descriptor is used as an argument by calls such as *read*(2), *write*(2), *ioctl*(2), and *close*(2).

file name A string of up to 14 characters that is used to refer to the contents of an ordinary file, special file, or directory. These characters may be any ASCII character except ASCII values 0 (null) and 47 (slash - /). Note that it is generally unwise

to use *, ?, [, !, or] as part of file names because of the special meaning the shell attaches to these characters (see *sh*(1)). It is also not wise to begin a file name with −, +, or =, because some programs assume that these characters indicate that a command argument follows. Although permitted, it is advisable to avoid the use of characters that do not have a printable graphic on the hardware you commonly use, or are likely to confuse the hardware.

file pointer A data element, obtained through any of the *fopen*(3S) standard I/O library routines that "points to" (refers to) a file opened for reading and/or writing, and which keeps track of where the next I/O operation will take place in the file (in the form of a byte offset relative to the beginning of the file). After obtaining the file pointer, it must thereafter be used to refer to the open file when using any of the standard I/O library routines. (See *stdio*(3S) for a list of these routines.)

file system The supporting data structures, HP-UX directory structure, and associated files that reside on one or more mass storage volumes. Refer to the *System Administrator Manual* supplied with your system for details concerning file system implementation and maintenance.

filter A command which reads data from the standard input, performs a transformation on the data, and writes it to the standard output.

fork An HP-UX system call (*fork*(2)) which, when invoked by an existing process, causes a new process to be created. The new process is called the *child process*; the existing process is called the *parent process*. The child process is created by making an exact copy of the parent process. The parent and child processes are able to identify themselves by the value returned by their corresponding *fork* call (see *fork*(2) for details).

group An association of zero or more users who must all be permitted to access the same set of files. The members of a group are defined in the file **/etc/passwd** via a numerical group ID (users with identical group IDs are members of the same group). An ASCII group name is associated with each group ID in the file **/etc/group** (the members of each group can be listed in **/etc/group**, also, but this information is purely for user benefit, and is of little use to the system). A group name is associated with every file in the file system, and the mode of each file contains a set of permission bits which apply only to groups of which the file owner is a member. Thus, **if** you are a member of the group name associated with the file (as determined by the information in **/etc/group** and **/etc/passwd**), and **if** the appropriate permissions are given to your group in the file's mode, you may access the file. See *real group ID*, *effective group ID*, *privelged group* and *set-group-ID bit*.

group access list The group access list is an additional set of group ID's used only in determining resource accessibility. Access checks are performed as described in file access permissions.

hierarchical directory
A directory (or file system) structure in which each directory may contain other directories as well as files.

home directory The directory name given by the value of the shell variable HOME. When you first log in, *login*(1) automatically sets HOME equal to your login directory (see *login directory*). You may change its value at any time, however. This is usually done in the **.profile** file contained in your login directory. Setting HOME in no way affects your login directory, but simply gives you a convenient way of referring to what should be your most commonly-used directory.

host name An ASCII string of at most 8 characters (of which only 6 are supported by all the various manufactuer's UNIX operating systems) which uniquely identifies an HP-UX system on a *uucp* network. The host name for your system may be viewed and/or set with the *hostname*(1) command. Systems without a defined host name are described as "unknown" on the *uucp* network. Do not confuse a host name with a *node name*, which is a string that uniquely identifies an HP-UX system on a Local Area Network (LAN). Although your host and node names may be identical, they are set and used by totally different software. See *node name.*

inode Each ordinary or special file, or directory has associated with it an inode. The inode contains, among other things, the file's size, protection mask, the number of links, and pointers to the disk blocks where the file's contents can be found. Each connection between an inode and its entry in one or more directories is called a link.

image The current state of your computer (or your portion of the computer, on a multi-user system) during the execution of a command. Often thought of as a "snapshot" of the state of the machine at any particular moment during execution.

init A special process (the initialization process) usually with a process ID of 1. It is the ancestor of every other process in the system and is used to start login processes.

interleave factor A number which determines the order in which sectors on a mass storage medium are accessed. It can be optimized to make data acquisition more efficient.

Internal Terminal Emulator (ITE)
The "device driver" code contained in the HP-UX kernel and associated with the computer's built-in keyboard and display or a particular keyboard and display connected to the computer, depending on the Series and Model of your HP-UX computer. See *system console* and the *System Adminstrator Manual* supplied with your system for details.

interrupt signal The signal sent by SIGINT (see *signal*(2)). This signal generally terminates whatever program you are running. The key which sends this signal can be redefined with *ioctl*(2) or *stty*(1) (see *termio*(7)). It is often the ASCII DEL (rubout) character (the [DEL] key) or the [BREAK] key. [CONTROL]-[C] is often used instead.

intrinsic See *system call.*

I/O redirection A mechanism provided by the HP-UX shell for changing the source of data for standard input and/or the destination of data for standard output and standard error. See *sh*(1).

job control Job control allows users to selectively stop (suspend) the execution of processes and continue (resume) their execution at a later point.

The user employs this facility via the interactive interface jointly supplied by the system tty driver and *csh*(1). The tty driver recognizes a user-defined *suspend character* which causes all current foreground processes to stop and the user's job control shell to resume. The job control shell provides commands which continue stopped processes in either the foreground or background. The tty driver will also stop a background process when it attempts to read from or write to the users terminal. This allows the user to finish or suspend their foreground task without interruption and continue the stopped background process at a more

convenient time.

See *csh*(1), *signal*(2), and *termio*(7).

kernel The HP-UX operating system. The kernel is the executable code responsible for managing the computer's resources, such as allocating memory, creating processes, and scheduling programs for execution. The kernel resides in RAM (Random Access Memory) whenever HP-UX is running.

library An archive file containing a set of subroutines and variables which may be accessed by user programs. For example, **/lib/libc.a** is a library containing all functions of section (2), and all functions of section (3) marked (3C) and (3S), in the HP-UX Reference. Similarly, **/lib/libm.a** is a library containing all functions in section (3) marked (3M) in the HP-UX Reference. See *intro*(3).

LIF An acronym for Logical Interchange Format. A standard format for mass storage implemented on many Hewlett-Packard computers to aid in media transportability. The *lif**(1) commands are used to perform various functions using LIF.

link A directory entry for any type of file. The information constituting a link includes the name of the file, and where the contents of that file may be found on a mass storage medium. One physical file may have several links to it. If the links appear in different directories, the file may or may not have the same name in each. If the links appear in one directory, however, each link must have a unique name in that directory. Multiple links to directories are not allowed (except for the super-user). See *cp*(1), *link*(1), *link*(2), and *unlink*(2). Also, to prepare a program for execution, see *linker*.

linker The linker combines one or more object programs into one program, searches libraries to resolve user program references, and builds an executable file in *a.out* format. This executable file is ready to be executed through the program loader, *exec*(2). The linker is invoked with the *ld*(1) command. The linker is often called a *link editor*.

logical block size The smallest unit of memory which can be allocated on a Series 500 SDF volume; a multiple of the physical sector size. This value is set at system initialization time; see *init*.

login The process of gaining access to HP-UX. This consists of successful execution of the login sequence defined by *login*(1) which varies depending on the system configuration. It includes providing a login name and possibly one or more passwords.

login directory The directory in which you are placed immediately after you log in. This directory is defined for each user in the file **/etc/passwd**. The shell variable HOME is set automatically to your login directory by *login*(1) immediately after you log in. See *home directory*.

magic number The first word of an *a.out*–format or archive file. This word contains the system ID, which states what machine (hardware) the file will run on, and the file type (executable, shareable executable, archive, etc.).

major number A number used exclusively to create special files that enable I/O to/from specific devices. This number indicates which device driver to use for the device. Refer to *mknod*(2) and the *System Administrator Manual* supplied with your system for details.

message queue identifier
A message queue identifier (msqid) is a unique positive integer created by a

msgget(2) system call. Each msqid has a message queue and a data structure associated with it. The data structure is referred to as *msqid_ds* and contains the following members:

```
struct  ipc_perm msg_perm; /* operation permission struct */
ushort  msg_qnum;          /* number of msgs on q */
ushort  msg_qbytes;        /* max number of bytes on q */
ushort  msg_lspid;         /* pid of last msgsnd operation */
ushort  msg_lrpid;         /* pid of last msgrcv operation */
time_t  msg_stime;         /* last msgsnd time */
time_t  msg_rtime;         /* last msgrcv time */
time_t  msg_ctime;         /* last change time */
                           /* Times measured in secs since */
                           /* 00:00:00 GMT, Jan. 1, 1970 */
```

Message queue identifiers may be created using *stdipc*(3C).
Msg_perm is a ipc_perm structure that specifies the message operation permission (see below). This structure includes the following members:

```
ushort  cuid;      /* creator user id */
ushort  cgid;      /* creator group id */
ushort  uid;       /* user id */
ushort  gid;       /* group id */
ushort  mode;      /* r/w permission */
```

Msg_qnum is the number of messages currently on the queue. **Msg_qbytes** is the maximum number of bytes allowed on the queue. **Msg_lspid** is the process id of the last process that performed a *msgsnd* operation. **Msg_lrpid** is the process id of the last process that performed a *msgrcv* operation. **Msg_stime** is the time of the last *msgsnd* operation, **msg_rtime** is the time of the last *msgrcv* operation, and **msg_ctime** is the time of the last *msgctl*(2) operation that changed a member of the above structure.

message operation permissions

In the *msgop*(2) and *msgctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

```
00400   Read by user
00200   Write by user
00060   Read, Write by group
00006   Read, Write by others
```

Read and Write permissions on a msqid are granted to a process if one or more of the following are true:

The process's effective user ID is super-user.

The process's effective user ID matches **msg_perm.[c]uid** in the data structure associated with *msqid* and the appropriate bit of the "user" portion (0600) of **msg_perm.mode** is set.

The process's effective user ID does not match **msg_perm.[c]uid** and either the process's effective group ID matches **msg_perm.[c]gid** or one of **msg_perm.[c]gid** is in the process's group access list and the appropriate bit of the "group" portion (060) of **msg_perm.mode** is set.

The process's effective user ID does not match **msg_perm.[c]uid** and the process's effective group ID does not match **msg_perm.[c]gid** and neither of **msg_perm.[c]gid** is in the process's group access list and the

appropriate bit of the "other" portion (06) of **msg_perm.mode** is set. Otherwise, the corresponding permissions are denied.

metacharacter A character which has special meaning to the HP-UX shell. The set of metacharacters includes: *, ?, !, [,], <, >, ;, |, ', `, ", and &. Refer to *sh*(1) for the meaning associated with each.

minor number A number used exclusively to create special files that enable I/O to/from specific devices. This number is passed to the device driver and is used to select which device in a family of devices is to be used, and possibly some operational modes. The exact format and meaning of the minor number is both system and driver dependent. Refer to the *System Administrator Manual* supplied with your system for details. See *address*.

On the Series 300 and 500, for HP-IB devices, this number indicates the HP-IB address, select code, and the unit and/or volume numbers.

mode A 16-bit word associated with every file in the file system, stored in the inode. The least-significant 12 bits of this word determine the read, write, and execute permissions for the file owner, file group, and all others, and contain the set-user-ID, set-group-ID, and "sticky" (save text image after execution) bits. The least-significant 12 bits are settable by the *chmod*(1) command if you are the file's owner or the super-user. The sticky bit can only be set by the super-user. These 12 bits are sometimes referred to as *permission bits*. The most-significant 4 bits specify the file type for the associated file and are set as the result of *open*(2) or *mknod*(2) system calls.

mountable file system
A removable blocked file system contained on some mass storage medium with its own root directory and an independent hierarchy of directories and files. See *block special file* and *mount*(1M).

multi-user state The condition of the HP-UX operating system in which terminals in addition to the system console are allowing communication between the system and its users. By default, the Series 300 multi-user state is state 2, and the Series 500 multi-user state is state 1. Do not confuse the multi-user system with the multi-user state. A multi-user system is a system which may have more than one user actively communicating with the system when it is in the multi-user state. The multi-user state removes the single-user restriction imposed by the single-user state. See *single-user state*. See *inittab*(4).

new-line The character with an ASCII value of 10 (line-feed) used to separate lines of characters. It is represented by \n in the C language and in various utilities. The terminal driver (see *tty*(7)) normally interprets the carriage-return/line-feed sequence sent by a terminal as a single new-line character.

node name A string of up to 31 characters, not including control characters or spaces, that uniquely identifies a node on a Local Area Network (LAN). The node name for each system is set by the *npowerup* command, which is one of the commands supplied with the optional LAN/9000 product. Do not confuse a node name with a *host name*, which is a string that uniquely identifies an HP-UX system on a *uucp* network. Your node and host names can be identical, but they are used and set by totally different software. See *host name*, *LAN/9000 User's Guide*, and *LAN/9000 Node Manager's Guide*.

ordinary file A type of HP-UX file containing ASCII text (e.g. program source), binary data (e.g. executable code), etc. Ordinary files can be created by the user through I/O redirection, editors, or HP-UX commands.

orphan process Whenever a parent process terminates for any reason and leaves behind one or more child processes that are still active, those child processes are called *orphan processes*. *Init*(1M) inherits (becomes the effective parent of) all orphan processes.

OSF An acronym for Operating System File. An OSF resides in the SDF boot area on a Series 500 system, and contains all or part of an operating system.

owner The owner of a file is usually the creator of that file. However, the ownership of a file can be changed by the super-user or the current owner with the *chown*(1) command or the *chown*(2) system call. The file owner is able to do whatever he wants with his files, including remove them, copy them, move them, change their contents, etc. He is also able to change the files' modes.

parent directory A directory's parent directory is the directory one level above it in the file hierarchy. All directories except the root directory (/) have one (and only one) parent directory. The parent directory is sometimes referred to as the *superior directory*.

parent process Whenever a new process is created by a currently-existing process (via *fork*(2)), the currently-existing process is said to be the parent process of the newly-created process. Every process has exactly one parent process (except the init process, see *init*), but each process can create several new processes with the *fork*(2) system call. The parent process ID of any process is the process ID of its creator.

password A string of ASCII characters used to verify the identity of a user. Passwords can be associated with users and groups. If a user has a password, it is automatically encrypted and entered in the second field of that user's line in the **/etc/passwd** file. A user may create or change a password for himself with the *passwd*(1) command.

path name (sometimes written as one word, *pathname*). A sequence of directory names separated by slashes, and ending with any file name. All file names except the last in the sequence *must* be directories. If a path name begins with a slash (/), it is an *absolute* path name (see *absolute path name*); otherwise it is a *relative* path name (see *relative path name*). A path name defines the path to be followed through the hierarchical file system in order to find a particular file.

More precisely, a path name is a null-terminated character string constructed as follows:

<path-name>::=<file-name>|<path-prefix><file-name>|/
<path-prefix>::=<rtprefix>|/<rtprefix>
<rtprefix>::=<dirname>/|<rtprefix><dirname>/

where <file-name> is a string of 1 to 14 characters other than the ASCII slash and null, and <dirname> is a string of 1 to 14 characters (other than the ASCII slash and null) that names a directory.

A slash by itself names the root directory.

Unless specifically stated otherwise, the null path name is treated as if it named a non-existent file.

permission bits The nine least-significant bits of a file's mode. These bits determine read, write, and execute permissions for the file's owner, the file's group, and all others.

pipe An inter-process I/O channel used to pass data between two processes. It is commonly used by the shell to transfer data from the standard output of one process to the standard input of another. On a command line, a pipe is signaled by a vertical bar (|). The output from the command(s) on the left of the vertical bar is channeled directly into the standard input of the command(s) on the right.

privileged groups
A privileged group is a group which has had a *setprivgrp* (see *getprivgrp*(2)) operation performed on it giving it access to some system calls otherwise reserved for the super-user.

proc1 See *init*.

process An invocation of a program, or the execution of an image. No command can be executed without a process in which it can execute. Alternately, a process cannot exist without a command or image in some stage of execution. Several processes can all be running the same program, but each may have different data and be in different stages of execution.

process group An association of one or more processes is called a process group. A process's membership in a particular process group is established by a numerical process group ID. Each process can belong to only one process group. Every process group has a process group leader. See *process group ID* and *process group leader*.

process group ID
A positive integer in the range 1 – 30000 associated with every active process, which establishes that process's membership with a particular process group. All members of a process group have the same process group ID. A process group ID is always the process ID of the process group leader. This grouping permits the signalling of related processes. See *kill*(2), *process group*, and *process group leader*.

process group leader
A process group leader is a process whose process group ID and process ID are equal. A process becomes a process group leader through the *setpgrp*(2) system call. All processes created by the process group leader become members of that process group. All processes created by the *init* process (see *init*) are process group leaders. For example, when you log in on the system, the shell you receive to interpret your commands is a process group leader, and all subsequent process's created by your shell are members of your shell's process group. See *process group ID* and *process group*.

process ID Each active process in the system is uniquely identified by a positive integer called a process ID. The range of this ID is from 1 to 30000. This permits the selective sending of signals to processes with *kill*(1), *kill*(2), or *signal*(2). The process ID of any user process is available with the *ps*(1) command. If a background process is created, the shell reports its process ID to you when execution has begun.

program A sequence of instructions to the computer in the form of binary code (resulting from the compilation and assembly of program source).

prompt The character(s) displayed by the shell on the display indicating that the system is ready for a command. The prompt is usually a dollar sign ($) for ordinary users and a pound sign (#) for the super-user, but the user can re-define it to be any string by setting the shell variable **PS1** in his **.profile** file.

quit signal The signal sent by SIGQUIT. See *signal*(2). The quit signal is generated by typing the character defined by the teletype handler as your quit signal. (See *stty*(1), *ioctl*(2), and *termio*(7).) The default is the ASCII FS character (ASCII value 28, generated by typing [CONTROL]-[\].) This signal usually causes a running program to terminate and generates a file containing the "core image" of the terminated process. The core image is useful for debugging purposes. (Some systems do not support core images, and on those systems no such file is generated.)

raw disk The name given to a disk for which there exists a character special file which allows direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O call.

real group ID A positive integer which is assigned to every user on the system. The association of a user and his real group ID is done in the file **/etc/passwd**. The modifier "real" is used because a user can also have an *effective* group ID (see *effective group ID*). The real group ID can then be mapped to a group name in the file */etc/group*, although it need not be. Thus, every user is a member of some group (which may be nameless), even if that group has only one member.

Every time a process creates a child process (via *fork*(2)), that process has a real group ID equal to the parent process's real group ID. This is useful for determining file access privileges within the process.

real user ID A positive integer which is assigned to every user on the system. A real user ID is assigned to every valid login name in the file **/etc/passwd**. The modifier "real" is used because a user can also have an *effective* user ID (see *effective user ID*).

Every time a process creates a child process (via *fork*(2)), that process has a real user ID equal to the parent process's real user ID. This is useful for determining file access privileges within the process.

regular expression
A string of zero or more characters which selects text. The characters contained in the string may all be literal, which means that the regular expression matches itself only, or one or more of the characters may be a metacharacter, which means that a single regular expression could match several literal strings. Regular expressions are most often encountered in text editors (such as *ed*(1), *ex*(1), or *vi*(1)), where searches are performed for a specific piece of text, or in commands that were created to search for a particular string in a file (most notably *grep*(1)). Regular expressions are also encountered in the shell, *sh*(1), especially when referencing file names on command lines. See *ed*(1).

relative path name
A path name that does not begin with a slash. It indicates that a file's location is given relative to your current working directory, and that the search begins there (instead of at the root directory). An example is **dir1/file2**, which searches for the directory **dir1** in your current working directory. **Dir1** is then searched for the file **file2**.

root directory (1) The highest level directory of the hierarchical file system, from which all other files branch. In HP-UX, the "/" character refers to the root directory. The root directory is the only directory in the file system that is its own parent directory.

(2) Each process has associated with it a concept of a root directory for the

purpose of resolving path name searches for those paths beginning with "/". A process's root directory need not be the root directory of the root file system, and can be changed by the *chroot*(1) command or *chroot*(2) system call. Such a directory appears to the process involved to have .. pointing to itself.

root volume The mass storage volume which contains the boot area (which contains the HP-UX kernel) and the root directory of the HP-UX file system.

saved group ID Every process has a saved group ID which retains the process's effective group ID from the last successful *exec*(2), or from the last super-user call to *setgid*. *Setgid* permits a process to set its effective group ID to this remembered value. Consequently, a process which executes a program with the set-group-ID bit set and with a group ID of 5 (for example) can set its effective group ID to 5 any time until the program terminates. See *exec*(2), *setuid*(2), *saved user ID*, *effective group ID*, and *set-group-ID bit*.

saved process group ID
Every process has a saved process group ID that retains the process's group ID from the last successful *exec*(2). See *setpgrp*(2), *termio*(7), and process group ID.

saved user ID Every process has a saved user ID which retains the process's effective user ID from the last successful *exec*(2), or from the last super-user call to *setuid*(2). *Setuid*(2) permits a process to set its effective user ID to this remembered value. Consequently, a process which executes a program with the set-user-ID bit set and with an owner ID of 5 (for example) can set its effective user ID to 5 any time until the program terminates. See *exec*(2), *setuid*(2), *saved group ID*, *effective user ID*, and *set-user-ID bit*.

SCCS An acronym for Source Code Control System. The Source Code Control System is a set of HP-UX commands which enable you to store changes to an SCCS file as separate "units" (called *deltas*). These units, each of which contains one or more textual changes to the file, can then be applied to or excluded from the SCCS file to obtain different versions of the file. The commands that make up SCCS are *admin*(1), *cdc*(1), *delta*(1), *get*(1), *prs*(1), *rmdel*(1), *sact*(1), *sccsdiff*(1), *unget*(1), *val*(1), and *what*(1). See *delta*, *SCCS file*.

SCCS file An ordinary text file which has been modified so that the Source Code Control System (SCCS) may be used with it. This modification is done automatically by the *admin*(1) command. See *SCCS*, *delta*.

SDF An acronym for Structured Directory Format. SDF is implemented on the Series 500 computers only, and provides tree-structured access to files through the root directory of the volume.

secondary prompt
One or more characters that the shell prints on the display, indicating that more input is needed. This prompt is much less often encountered than the shell's primary prompt (see *prompt*). When it occurs, it is usually caused by an omitted right quote on a string (which confuses the shell), or when you enter a shell programming language control-flow construct (such as a **for** construct) from the command line. By default, the shell's secondary prompt is the greater-than sign (>), but you can re-define it by setting the shell variable **PS2** appropriately in your **.profile** file.

select code On the Series 300 and 500 part of an address used for devices. A number determined by a setting on the interface card to which a peripheral device is connected, or by the particular I/O slot in which the I/O card resides. Multiple peripherals connected to the same interface card share the same select code.

semaphore identifier

A semaphore identifier (semid) is a unique positive integer created by a *semget*(2) system call. Each semid has a set of semaphores and a data structure associated with it. The data structure is referred to as *semid_ds* and contains the following members:

```
struct  ipc_perm sem_perm; /* operation permission struct */
ushort  sem_nsems;         /* number of sems in set */
time_t  sem_otime;         /* last operation time */
time_t  sem_ctime;         /* last change time */
                           /* Times measured in secs since */
                           /* 00:00:00 GMT, Jan. 1, 1970 */
```

Semaphore identifiers may be created using *stdipc*(3C).

Sem_perm is a ipc_perm structure that specifies the semaphore operation permission (see below). This structure includes the following members:

```
ushort  cuid;      /* creator user id */
ushort  cgid;      /* creator group id */
ushort  uid;       /* user id */
ushort  gid;       /* group id */
ushort  mode;      /* r/a permission */
```

The value of **sem_nsems** is equal to the number of semaphores in the set. Each semaphore in the set is referenced by a positive integer referred to as a *sem_num*. Sem_num values run sequentially from 0 to the value of sem_nsems minus 1. **Sem_otime** is the time of the last *semop*(2) operation, and **sem_ctime** is the time of the last *semctl*(2) operation that changed a member of the above structure.

A semaphore is a data structure that contains the following members:

```
ushort  semval;    /* semaphore value */
short   sempid;    /* pid of last operation  */
ushort  semncnt;   /* # awaiting semval > cval */
ushort  semzcnt;   /* # awaiting semval = 0 */
```

Semval is a non-negative integer. **Sempid** is equal to the process ID of the last process that performed a semaphore operation on this semaphore. **Semncnt** is a count of the number of processes that are currently suspended awaiting this semaphore's **semval** to become greater than its current value. **Semzcnt** is a count of the number of processes that are currently suspended awaiting this semaphore's **semval** to become zero.

semaphore operation permissions

In the *semop*(2) and *semctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

```
00400   Read by user
00200   Alter by user
00060   Read, Alter by group
00006   Read, Alter by others
```

Read and Alter permissions on a semid are granted to a process if one or more of the following are true:

The process's effective user ID is super-user.

The process's effective user ID matches **sem_perm.[c]uid** in the data structure associated with *semid* and the appropriate bit of the "user" portion (0600) of **sem_perm.mode** is set.

The process's effective user ID does not match **sem_perm.[c]uid** and the appropriate bit of the "group" portion (060) of **sem_perm.mode** is set.

The process's effective user ID does not match **sem_perm.[c]uid** and the process's effective group ID does not match **sem_perm.[c]gid** and neither of **sem_perm.[c]gid** is in the process's group access list and the appropriate bit of the "other" portion (06) of **sem_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

set-group-ID bit A single bit in the mode of every file in the file system. If a file is executed whose set-group-ID bit is set, the effective group ID of the process which executed the file is set equal to the real group ID of the owner of the file. See *effective group ID*, *group*, and *real group ID*.

set-user-ID bit A single bit in the mode of every file in the file system. If a file is executed whose set-user-ID bit is set, the effective user ID of the process which executed the file is set equal to the real user ID of the owner of the file. See *effective user ID* and *real user ID*.

shared memory identifier
A shared memory identifier (shmid) is a unique positive integer created by a *shmget*(2) system call. Each shmid has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. The data structure is referred to as *shmid_ds* and contains the following members:

```
struct  ipc_perm shm_perm; /* operation permission struct */
int     shm_segsz;         /* size of segment */
ushort  shm_cpid;          /* creator pid */
ushort  shm_lpid;          /* pid of last operation */
short   shm_nattch;        /* number of current attaches */
time_t  shm_atime;         /* last attach time */
time_t  shm_dtime;         /* last detach time */
time_t  shm_ctime;         /* last change time */
                           /* Times measured in secs since */
                           /* 00:00:00 GMT, Jan. 1, 1970 */
```

Shared memory identifiers may be created using *stdipc*(3C).

Shm_perm is a ipc_perm structure that specifies the shared memory operation permission (see below). This structure includes the following members:

```
ushort  cuid;      /* creator user id */
ushort  cgid;      /* creator group id */
ushort  uid:       /* user id */
ushort  gid;       /* group id */
ushort  mode;      /* r/w permission */
```

Shm_segsz specifies the size of the shared memory segment. **Shm_cpid** is the process id of the process that created the shared memory identifier. **Shm_lpid** is the process id of the last process that performed a *shmop*(2) operation. **Shm_nattch** is the number of processes that currently have this segment attached. **Shm_atime** is the time of the last *shmat* operation, **shm_dtime** is

the time of the last *shmdt* operation, and **shm_ctime** is the time of the last *shmctl*(2) operation that changed one of the members of the above structure.

shared memory operation permissions
In the *shmop*(2) and *shmctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

00400	Read by user
00200	Write by user
00060	Read, Write by group
00006	Read, Write by others

Read and Write permissions on a shmid are granted to a process if one or more of the following are true:

The process's effective user ID is super-user.

The process's effective user ID matches **shm_perm.[c]uid** in the data structure associated with *shmid* and the appropriate bit of the "user" portion (0600) of **shm_perm.mode** is set.

The process's effective user ID does not match **shm_perm.[c]uid** and either the process's effective group ID matches **shm_perm.[c]gid** or one of **shm_perm.[c]gid** is in the process's group access list and the appropriate bit of the "group" portion (060) of **shm_perm.mode** is set.

The process's effective user ID does not match **shm_perm.[c]uid** and the process's effective group ID does not match **shm_perm.[c]gid** and neither of **shm_perm.[c]gid** is in the process's group access list and the appropriate bit of the "other" portion (06) of **shm_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

shell The shell functions as both a command interpreter and an interpretive programming language. The shell is automatically invoked for every user who logs in, in order to provide a user-interface to the HP-UX operating system. See *sh*(1) and the tutorials supplied with your system for details.

shell program See *shell script.*

shell script A sequence of shell commands and shell programming language constructs stored in a file and invoked as a user command (program). No compilation is needed prior to execution, because the shell recognizes the commands and constructs that make up the shell programming language. A shell script is often called a *shell program* or a *command file.* See the shell programming article included in *HP-UX Selected Articles.*

signal Signals are software interrupts sent to processes, informing them of special situations or events. See *signal*(2).

single-user state A condition of the HP-UX operating system in which the system console provides the only communication mechanism between the system and its user. By default, the Series 300 single-user state is state 1, and the Series 500 multi-user state is state 2. Do not confuse the single-user state, in which the software is limiting a multi-user system to a single-user communication, with a single-user system, which can never communicate with more than one fixed terminal. See *multi-user state.*

special file Often called a *device file*, this is a file associated with an I/O device. Special files are read and written the same as ordinary files, but requests to read or write

result in activation of the associated device. Due to convention and consistency, these files should always reside in the **/dev** directory.

special processes
Processes with certain (small) process ID's are special. On a typical system, the ID's of 0, 1, and 2 are assigned as follows: Process 0 is the scheduler. Process 1 is the initialization process *init*, and is the ancestor of every other process in the system. It is used to control the process structure. On paging systems with virtual memory process 2 is the paging daemon.

On the Series 500, there is no process 0 and the scheduler does not exist as an identifiable entity. The paging daemon also does not exist as an identifiable entity.

standard error The destination of error and special messages from a program. The standard error file is often called *stderr*, and is automatically opened for writing on file descriptor 2 for every command invoked. By default, the user's terminal is the destination of all data written to *stderr*, but it can be redirected elsewhere. Unlike standard input and standard output which are never used for data transfer in the "wrong" direction, standard error is occasionally read. This is not recommended practice as I/O redirection is likely to break a program doing this.

standard input The source of input data for a program. The standard input file is often called *stdin*, and is automatically opened for reading on file descriptor 0 for every command invoked. By default, the user's terminal is the source of all data read from *stdin*, but it can be redirected from another source.

standard output The destination of output data from a program. The standard output file is often called *stdout*, and is automatically opened for writing on file descriptor 1 for every command invoked. By default, the user's terminal is the destination of all data written to *stdout*, but it can be redirected elsewhere.

stream A term most often used in conjunction with the standard I/O library routines documented in section (3) of this manual. A stream is simply a file pointer (declared as **FILE *stream**) returned by the *fopen*(3S) library routines. It may or may not have buffering associated with it (by default, buffering is assigned, but this may be modified with *setbuf*(3S)).

sticky bit A single bit in the mode of every file in the file system. If set, the contents of the file stay permanently in memory instead of being swapped back out to disk when the file has finished executing. Only the super-user can set the sticky bit. The sticky bit is read each time the file is executed (via *exec*(2)).

sub-directory A directory that is one (or perhaps more) levels lower in the file system hierarchy than a given directory. Sometimes called a *subordinate directory.*

subordinate directory
See *sub-directory.*

super block A block on each file system's mass storage medium which describes the file system. The contents of the super-block vary between implementations. Refer to the *System Administrator Manual* supplied with your system, and the appropriate *fs*(4) entry for details.

super-user The HP-UX system administrator. This user has access to all files, and can perform privileged operations. He has a real and effective user ID of 0, and, by convention, the user name of *root.*

superior directory
See *parent directory.*

system asynchronous I/O
is a method of performing I/O whereby a process informs a driver or subsystem that it wants to know when data has arrived or when it is possible to perform a write request. The driver or subsystem maintains a set of buffers through which the process performs I/O. See the *ioctl*(2), *select*(2), *read*(2), and *write*(2) manual pages for more information.

system call An HP-UX operating system kernel function available to the user through a high-level language (such as FORTRAN, Pascal, or C). Also called an "intrinsic" or a "system intrinsic." The available system calls are documented in section (2) of the *HP-UX Reference manual.*

system console A keyboard and display (or terminal) given a unique status by HP-UX and associated with the special file **/dev/console**. All boot ROM or system loader error messages, HP-UX system error messages, and certain system status messages are sent to the system console. Under certain conditions (such as the single-user state), the system console provides the only mechanism for communicating with HP-UX. See *HP-UX Selected Articles* and the *System Administrator Manual* provided with your system for details on configuration and use of the system console.

system loader On the Series 500, a piece of executable code that permanently resides in the computer. When the computer is powered up, the system loader is automatically loaded and run. Its function is to find and load an operating system. The Series 300 has an identical program which differs only in implementation. On the Series 300, the program resides in Read Only Memory (ROM). For a complete description of the system loader's function for all implementations, see *boot ROM.*

terminal affiliation
The process by which a process group leader establishes an association between itself and a particular terminal. A terminal becomes affiliated with a process group leader (and subsequently all processes created by the process group leader, see *terminal group*) whenever the process group leader executes (either directly or indirectly) an *open*(2) or *creat*(2) system call to open a terminal. Then, **if** the process which is executing *open*(2) or *creat*(2) is a process group leader, and **if** that process group leader is not yet affiliated with a terminal, and **if** the terminal being opened is not yet affiliated with a process group, the affiliation is established.

An affiliated terminal keeps track of its process group affiliation by storing the process group's process group ID in an internal structure.

Two benefits are realized by terminal affiliation. First, all signals sent from the terminal are sent to all processes in the terminal group. Second, all processes in the terminal group can perform I/O to/from the generic terminal driver **/dev/tty**, which automatically selects the affiliated terminal.

Terminal affiliation is broken with a terminal group when the process group leader terminates, after which the hangup signal is sent to all processes remaining in the process group. Also, if a process (which is not a process group leader) in the terminal group becomes a process group leader via the *setpgrp*(2) system call, its terminal affiliation is broken.

See *process group*, *process group leader*, *terminal group*, and *setpgrp*(2).

terminal group A terminal group is a process group whose process group leader has established affiliation with a particular terminal. Once a process group leader has established affiliation with a terminal, all processes in that process group created **after** the affiliation are members of that terminal group. Processes existing before and during the time when affiliation is established do not inherit the affiliation, and are thus not part of the terminal group. A terminal group is sometimes called a *tty group*.

This grouping is used to terminate a group of related process upon termination of one of the processes in the group; see *exit*(2) and *signal*(2).

See *process group*, *process group leader*, *terminal affiliation*, and *setpgrp*(2).

tty group ID See *terminal group*.

unit number Part of an address used for devices. A number whose meaning is software- and device-dependent, but which is often used to specify a particular disk drive in a device with a multi-drive controller. See the *System Administrator Manual* supplied with your system for details.

volume number Part of an address used for devices. A number whose meaning is software- and device-dependent, but which is often used to specify a particular volume on a multi-volume disk drive. See the *System Administrator Manual* supplied with your system for details.

working directory
Each process has associated with it the concept of a current working directory. For a shell, this appears as the directory in which you currently reside. This is the directory in which relative path name (i.e., a path name that does not begin with "/") searches begin. It is sometimes referred to as the *current directory*, or the *current working directory*.

zombie process The name given to a process which terminates for any reason, but whose parent process has not yet waited for it to terminate (via *wait*(2)). The process which terminated continues to occupy a slot in the process table until its parent process waits for it. Because it has terminated, however, there is no other space allocated to it either in user or kernel space. It is therefore a relatively harmless occurrence which will rectify itself the next time its parent process waits. The *ps*(1) command lists zombie processes as "defunct."

Permuted Index

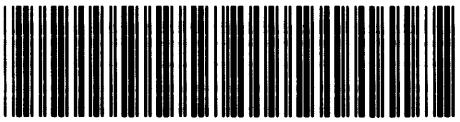
Permuted Index

Permuted Index

Permuted Index

Permuted Index

HP Part Number
09000-90010
Microfiche No. 09000-99010
Printed in U.S.A. 4/87



09000-90665
For Internal Use Only