CRAY RESEARCH, INC.

CRAY® COMPUTER SYSTEMS

UNICOS USER COMMANDS
REFERENCE MANUAL

SR-2011

RECORD OF REVISION CRAY RESEARCH, INC. PUBLICATION NUMBER SR-2011

Requests for copies of Cray Research, Inc. publications should be directed to the Distribution Center and comments about these publications should be directed to:

CRAY RESEARCH, INC.
Technical Publications
1345 Northland Drive
Mendota Heights, Minnesota 55120

Revision	Description
March 1986	Documentation to support the UNICOS release 1.0 running on Cray computer systems. This documentation is derived from UNIX System V under license from AT&T Technologies, Inc.
July 1986	On-line documentation only to support the UNICOS release 1.1 running on Cray computer systems.
October 1986	Documentation to support the UNICOS release 2.0 running on Cray computer systems. This documentation contains the following new commands: cft77, clear, csh, csim, dda, dispose, drd, ex, expand, fetch, flodump, flow, lastcomm, more, nasa, ps, premult, printenv, prof, rdrop, rdist, sag, sar, sc, scpqsub, scpreroute, sim, stty, target, tput, tset, ul, unexpand, uupick, uuname, uustat, uuto, uux, vi, wdrop, and whereis. See the following commands for revisions: adb, ar, as, asa, awk, cb, cft, dd, df, finger, ld, nice, pascal, rlogin, segldr, sim, update, units, and qsub. Other commands have some minor revisions.

PREFACE

The UNICOS Commands Reference Manual provides descriptions of commands and application programs for system users of the Cray operating system UNICOS. It supplements the information contained in the other manuals in the UNICOS documentation set.

This manual describes programs that are invoked directly by the user or by command language procedures. Commands described in section 1 of this manual generally reside in the directory **/bin** (for binary programs). Programs can also reside in **/usr/bin** to save space in **/bin**, **/usr/ucb** (those commands ported from the Fourth Berkeley Software Distribution), and **/lib**. The **/bin**, **/usr/bin**, and **/usr/ucb** directories are searched automatically by the command interpreter called the *shell* (see *sh*(1)). You must change the path (or specify the path on the command line) if you want to use a command in **/lib**. The commands for which this is necessary are specified as such in the SYNOPSIS section on the man page.

This manual is a reference manual for UNICOS programmers. It is assumed that the reader has a working knowledge of either the Cray operating system UNICOS or the UNIX Operating System.

Other Cray Research, Inc. (CRI), manuals that may be helpful to the reader are listed below (all other manuals referred to are CRI publications unless otherwise specified):

- The appropriate CRI library reference manual:
  Programmer's Library Reference Manual, publication SR-0113
  System Libray Reference Manual, publication SM-0114
  CRAY X-MP and CRAY-1 C Library Reference Manual, publication SR-0136
  CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual, publication SR-2013
- UNICOS Administrator Commands Reference Manual, publication SR-2022
- UNICOS System Calls Reference Manual, publication SR-2012
- UNICOS File Formats and Special Files Reference Manual, publication SR-2014
- UNICOS Kernel Error Message Manual, publication SR-2015
- UNICOS Administrator Commands Reference Manual, publication SR-2022
- Cray C Reference Manual, publication SR-2024

CONVENTIONS

Throughout the UNICOS documentation, the following conventions are used:

command(1)	refers to an entry in the UNICOS User Commands Reference Manual, publication SR-2011.
command(1M)	refers to an entry in the UNICOS Administrator Commands Reference Manual, publication SR-2022.
routine(2)	refers to an entry in the UNICOS System Calls Reference Manual, publication SR-2012.
routine(3*x*)	refers to an entry in the appropriate CRI library reference manual. The optional letter following the number 3 indicates the section reference.
entry(4*x*)	refers to an entry in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014. The optional letter following the number 4 indicates the section reference.

Each manual consists of sections with independent entries of a page or more in length. The name of the entry appears in the upper corners of its pages. Entries within each section are alphabetized, with the exception of the introductory entry that begins each section. The page numbers of each entry start at 1. Some entries may describe several routines, commands, etc. In such cases, the entry appears only once, alphabetized under its "major" name.

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

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

SYNOPSIS summarizes the use of the program being described. The following conventions are used in the **SYNOPSIS:**

> **Boldface** strings are literals and are to be typed just as they appear.
>
> *Italic* strings usually represent substitutable argument prototypes and program names found elsewhere in the manual.
>
> Square brackets **[ ]** around an argument prototype indicate that the argument is optional. When an argument prototype is given as *name* or *file*, it always refers to a *file* name.
>
> Ellipses **...** indicate that the previous argument prototype may be repeated.
>
> An argument beginning with a minus, plus, or equal sign (-,+,or =) is often recognized to be some sort of flag argument, even if it appears in a position where a file name could appear. Therefore, it is recommended that files names do not begin with -, +, or =.

DESCRIPTION discusses the subject at hand.

EXAMPLES gives examples of usage, where appropriate.

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

SEE ALSO gives pointers to related information.

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

WARNINGS points out potential pitfalls.

LIMITATIONS identifies restrictions of the program being described.

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

All entries are available on line through the *man*(1) command.

A contents list follows this introduction.

USER COMMANDS (1)

intro	Introduction to general-utility commands
acctcom	Searches and prints process accounting files
acquire	Makes a request for a file from a front-end station
adb	Invokes the absolute debugger
admin	Creates and administers SCCS files
adstape (CRAY X-MP and CRAY-1 only)	Prepares IOS deadstart tape
apml (CRAY X-MP and CRAY-1 only)	Invokes the APML assembler
ar (CRAY-2 only)	Maintains archives and libraries for portable archives
ar (CRAY X-MP and CRAY-1 only)	Maintains archives and libraries for portable archives
as	Invokes the Cray assembler (CAL)
asa	Interprets ASA carriage control characters
at	Executes commands at a later time
awk	Scans and processes patterns
banner	Makes posters
basename	Prints portions of pathnames on standard output
batch (see *at*(1))	Executes commands at a later time
bc	Invokes the arbitrary-precision arithmetic language preprocessor
bdiff	Compares very large files for differences
bind (CRAY X-MP and CRAY-1 only)	Binds APML relocatable modules together
bmxio (CRAY X-MP and CRAY-1 only)	Provides an interface to block mux devices
cal	Prints a calendar
calendar	Reminder service
cat	Concatenates and prints files
cb	C program beautifier
cc (CRAY-2 only)	Invokes the C compiler
cc (CRAY X-MP and CRAY-1 only)	Invokes the C compiler
cd	Changes working directory
cdc	Changes the delta commentary of an SCCS delta
cflow	Generates C flow graph
cft (CRAY-2 only)	Invokes the CRAY-2 Fortran compiler
cft (CRAY X-MP and CRAY-1 only)	Invokes the CFT Fortran compiler
cft77	CFT77 Fortran compiler, not machine-specific
chgrp (see *chown*(1))	Changes owner or group
chmod	Changes mode
chown	Changes owner or group
chsh (CRAY X-MP and CRAY-1 only)	Changes default login shell
clear (CRAY X-MP and CRAY-1 only)	Clears terminal screen
cmp	Compares two files
comb	Combines SCCS deltas
comm	Selects or rejects lines common to two sorted files
cord (CRAY X-MP and CRAY-1 only)	Invokes the core dump program
cp	Copies files
cpio	Copies file archives in and out
cpp	Invokes the C language preprocessor
cray1 (see *machid*(1))	Provides truth value about processor type
cray2 (see *machid*(1))	Provides truth value about processor type
crayxmp (see *machid*(1))	Provides truth value about processor type
crontab	Copies files into the user crontab file

paste	Merges same lines of several files or subsequent lines of one file
pcat (see *pack*(1))	Compresses and expands files
pdp11 (see *machid*(1))	Provides truth value about processor type
pg	Lets you peruse files one screenful at a time
plcopy (CRAY X-MP and CRAY-1 only)	Converts COS PLs into UNICOS PLs
pr	Prints files
premult (CRAY X-MP only)	Invokes the premult preprocessor
printenv (CRAY X-MP and CRAY-1 only)	Prints out the environment
prof (CRAY-2 only)	Displays profile data
prs	Prints an SCCS file
ps (CRAY-2 only)	Reports process status
ps (CRAY X-MP and CRAY-1 only)	Reports process status
pwd	Prints working directory name
qdel	Deletes or signals NQS requests
qdev	Displays the status of NQS devices
qlimit	Shows supported batch limits and shell strategy for named host
qpr	Submits a hard-copy print request to NQS
qstat	Displays the status of NQS queues
qsub	Submits an NQS batch request
rcp (TCP/IP NETWORK)	Copies remote files
rdist (CCRAY X-MP and CRAY-1 only)	Remote file distribution program
rdrop (CRAY X-MP and CRAY-1 only)	Reloads a recoverable drop file
red (see *ed*(1))	Invokes the ed text editor
regcmp	Compiles regular expression
remsh (TCP/IP NETWORK)	Invokes a remote shell
reset (see *tset*(1))	Terminal dependent initialization
rlogin (TCP/IP NETWORK)	Invokes a remote login
rm	Removes files or directories
rmail (see *mail*(1))	Lets you send or read mail
rmdel	Removes a delta from an SCCS file
rmdir (see *rm*(1))	Removes files or directories
rsh (see *sh*(1))	Shell, the standard/restricted command programming language
ruptime (TCP/IP NETWORK)	Shows host status of local machines
rwho (TCP/IP NETWORK)	Indicates who is logged in on local machines
sact	Prints current SCCS file editing activity
sag (CRAY X-MP and CRAY-1 only)	System activity graph
sar (CRAY X-MP and CRAY-1 only)	Extracts operating system activity information
sc	Invokes the front end for the scm source code control program
sccsdiff	Compares two versions of an SCCS file
scm	Invokes the source control program
scpqsub	Allows you to submit jobs to NQS from USCP spawned job
scpreroute	Allows you to define station processing of job output
script (CRAY-2 only)	Makes typescript of terminal session
script (CRAY X-MP and CRAY-1 only)	Makes typescript of terminal session
sdiff	Side-by-side difference program
sed	Invokes the stream editor
segldr	Invokes the segment loader (SEGLDR)

sh	Shell, the standard/restricted command programming language
sim (CRAY-2 only)	Invokes the interactive Cray Simulator
size	Prints section sizes of executable files
sleep	Suspends execution for an interval
sno	Invokes the SNOBOL interpreter
sort	Sorts and/or merges files
split	Splits a file into pieces
strings (CRAY X-MP and CRAY-1 only)	Finds the printable strins in a object or other binary file
stty (CRAY X-MP and CRAY-1 only)	Sets the options for a teminal
su	Lets you become super user or another user
sum	Prints checksum and block count of a file
sync	Flushes file system cache to disk
tail	Displays the last part of a file
tar	Tape file archiver
target (CRAY X-MP and CRAY-1 only)	Verify target CPU characteristics
tee	Duplicates output
telnet (TCP/IP NETWORK)	User interface to the TELNET protocol
test	Performs a conditional evaluation
tftp (TCP/IP NETWORK)	Invokes the trivial file transfer program
time	Times a command
timex	Time a command; report process data and system activity
to2 (CRAY-2 only)	Copies files between VAX and CRAY-2
touch	Updates access and modification times of a file
tovax (see *to2*(1))	Copies files between VAX and CRAY-2
tput (CRAY X-MP and CRAY-1 only)	Makes *terminfo* data available to the shell
tr	Translates characters
true	Provides truth values about processor type
tset (CRAY X-MP and CRAY-1 only)	Terminal dependent initialization
tsort (CRAY-2 only)	Performs a topological sort
tty	Gets the name of the terminal
u370 (see *machid*(1))	Provides truth value about processor type
u3b (see *machid*(1))	Provides truth value about processor type
u3b5 (see *machid*(1))	Provides truth value about processor type
ul (CRAY X-MP and CRAY-1 only)	Underlines text
umask	Sets file-creation mode mask
uname	Prints name of current system
unexpand (see *expand*(1))	Replaces tabs in data
unget	Undoes a previous get of an SCCS file
uniq	Reports repeated lines in a file
units	Unit conversion program
unpack (see *pack*(1))	Compresses and expands files
update	Invokes the UPDATE utility
uucp (CRAY X-MP and CRAY-1 only)	UNIX system to UNIX system copy
uulog (see *uucp*(1))	UNIX system to UNIX system copy
uuname (see *uucp*(1))	UNIX system to UNIX system copy
uupick (see uuto(1))	Public UNIX-to-UNIX system file copy
uustat	Uucp status inquiry and job control
uuto (CRAY X-MP and CRAY-1 only)	Public UNIX-to-UNIX system file copy
uux (CRAY X-MP and CRAY-1 only)	UNIX-to-UNIX system command execution
val	Validates SCCS file
vax (see *machid*(1))	Provides truth value about processor type

NAME

intro – Introduction to general-utility commands

DESCRIPTION

This section describes, in alphabetical order, commands that are of general utility for the Cray operating system UNICOS.

COMMAND SYNTAX

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

name [*option*] [*cmdarg*]

name	The name of an executable command.
option	– *noargletter* 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	An argument (character string) satisfying the preceding *argletter*.

cmdarg	Path name (or other command argument) *not* beginning with a – or, a – by itself indicating the standard input.

MESSAGES

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

SEE ALSO

getopt(1)
getopt(3C) in the CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual, publication SR-2013

NAME

acctcom – Searches and prints process accounting files

SYNOPSIS

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

DESCRIPTION

The *acctcom* command reads *file*, the standard input, or **/usr/adm/pacct** , in the form described by *acct*(4F) 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 prefixed 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 you do not specify *files*, 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, that is, in chronological order by process completion time. The **/usr/adm/pacct** file is the current file to be examined. The options are as follows:

–a	Shows some average statistics about the processes selected. The statistics are printed after the output records.
–b	Reads backwards, showing latest commands first. This option has no effect when the standard input is read.
–f	Prints the **fork/exec** flag and system exit status columns in the output.
–h	Instead of mean memory size, shows 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	Prints columns containing the I/O counts in the output
–k	Instead of memory size, shows total kcore-minutes; this is an integral of memory usage over time. One kcore minute is 1024 words used for 1 minute.
–m	Shows mean core size (the default)
–r	Shows CPU factor (user time/(system-time + user-time)
–t	Shows separate system and user CPU times
–v	Excludes column headings from the output
–l *line*	Shows only processes belonging to terminal /dev/*line*

–u *user*	Shows 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*	Shows only processes belonging to *group*; the *group* may be designated by either the group ID or group name.
–s *time*	Selects processes existing at or after *time*, given in the format *hr* [:*min* [:*sec*]]
–e *time*	Selects processes existing at or before *time*
–S *time*	Selects processes starting at or after *time*
–E *time*	Selects processes ending at or before *time*. Using the same *time* for both –S and –E shows the processes that existed at *time*.
–n *pattern*	Shows only commands matching *pattern* that may be a regular expression as in *ed*(1) except that + means one or more occurrences
–q	Does not print any output records, just prints the average statistics as with the **–a** option.
–o *ofile*	Copies selected process records in the input data format to *ofile*; suppresses standard output printing.
–H *factor*	Shows only processes that exceed *factor*, where factor is the "hog factor" as explained in the preceding **–h** option.
–O *sec*	Shows only processes with CPU system time exceeding *sec* seconds
–C *sec*	Shows only processes with total CPU time, system plus user, exceeding *sec* seconds
–I *chars*	Shows only processes transferring more characters than the cut-off number given by *chars*

FILES

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

BUGS

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

SEE ALSO

ps(1), su(1),
acct(1M), acctcms(1M), acctcon(1M), acctmerg(1M), acctprc(1M), acctsh(1M), fwtmp(1M), runacct(1M) in the UNICOS Administrator Commands Reference Manual, publication SR-2022
acct(2) in the UNICOS System Calls Reference Manual, publication SR-2012
acct(4F), utmp(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

acquire – Makes a request for a file from a front-end station

SYNOPSIS

acquire *localpath* [**–n***SFN*] [**–i***TERMID*] [**–m***MF*] [**–d***DC*] [**–f***FM*] [**–t**'*TEXT*'] [**–u***USER*]

DESCRIPTION

The *acquire* command assures you that the requested file, specified by the *localpath* argument exists. If the file exists at the time you invoke the *acquire* command, the command returns directly with a positive status. If the file does not already exist, the *acquire* command creates a request file for USCP (UNICOS Station Call Processor). If any slot information is associated with the requesting user, it is also copied into the request file. USCP uses station protocol to make the request for a file from the designated station (specified by the *–m* option). The *acquire* command then waits until the transfer status has been determined. The transfer status is returned when a negative reply is received from the station (requested file did not transfer) or when a positive reply is received from the station (requested file has been saved on the Cray computer system). It is possible for the station to return a postpone status, in which case the *acquire* command resets the request for USCP to find and again waits for the transfer status. The *acquire* command accepts the following options:

localpath A path name (either full or relative to current working directory) where the requested file is to reside when the transfer is complete. The *localpath* must be a location where the requesting user has permission to write. This argument is required.

–n*SFN* The name associated with the requested file on the specified front end. This argument is stored in the request record PDN field. Only 15 characters are significant. If you do not specify *SFN*, the field is filled with the filename from *localpath.*

–i*TERMID* The terminal ID associated with the requested file on the spcified front end. The size limit is 8 characters. If you do not specify *TERMID*, the default is the terminal ID associated with the requesting user on the front-end station that user originated from.

–m*MF* A two character front-end ID for a station that has access to the requested file. If you do not specify the mainframe, the stored ID of the station from which the requesting user originated is used.

–f*FM* A two character file format code. Valid codes are:

CB	Character blocked; the default
CD	Character deblocked
BB	Binary blocked
BD	Binary deblocked
TR	Transparent
UD	UNICOS Data

For further descriptions of the valid format codes, see the Front End Protocol Internal Reference Manual, CRI publication SM-0042.

–d*DC* A two character dispostion code interpreted by the receiving system. Valid codes are:

IN	File is to be executed as a job by the receiving system.
ST	File is to be saved by the receiving system.

–t'*TEXT*' Text to be interpreted by the specified station for processing the request. The field can contain label information, routing, etc., possibly in the form of control statements for the station. Text field information should be enclosed by single quotes ('). If you do not specify this option, the request text field is filled with binary 0's.

–u*USER* The user ID associated with the requested file on the specified front end. If you do not specify *USER*, this field is left blank for the request.

LIMITATIONS

If you are not accessing the Cray computer system through USCP, defaults for *TERMID* and *MF* do not exist. The request is queued without regard to whether the mainframe ID specified belongs to a currently active station. If the associated station is not active or has no streams assigned (that is, interactive only station), the user process waits indefinitely.

SEE ALSO

fetch(1), dispose(1)
Front End Protocol Internal Reference Manual, publication SM-0042

NAME

adb – Invokes the absolute debugger

SYNOPSIS

adb [–w] [*objfil* [*corfil*]]

DESCRIPTION

The *adb* command is a general purpose debugging program, which examines files and provides a controlled environment in which to execute UNICOS programs. Requests to *adb* are read from the standard input and responses are written to the standard output.

The *adb* command accepts the following arguments:

objfil	An executable program file, preferably containing a symbol table; if it does not contain a symbol table, the symbolic features of *adb* cannot be used although you can still examine the file. The default for *objfil* is **a.out**.
corfil	Assumed to be a core image file produced after executing *objfil*; the default for *corfil* is **core**.
–w	Indicates that both *objfil* and *corfil* are created if necessary and opened for reading and writing so that files can be modified using *adb*.

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

In general, requests to *adb* are of the format:

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

Address and *count* are expressions (see the EXPRESSION subsection). If *address* is present, *dot* is set to *address*. Initially *dot* is set to 0. For most commands, *count* specifies how many times the command is executed; default *count* is 1.

The interpretation of an address depends on the context in which it is used. If a subprocess is being debugged, addresses are interpreted in the usual way in the address space of the subprocess. For further details of address mapping see the ADDRESSES section.

Expressions

Expressions have the following meanings:

.	The value of *dot*
+	The value of *dot* incremented by the current increment
^	The value of *dot* decremented by the current increment
"	The last *address* typed
integer	An octal number. If you have used a $d command, hexadecimal (leading Ox), and octal (leading 0) bases are accepted.
integer.fraction	A 64-bit floating-point number
'*cccccccc*'	The ASCII value of up to 8 characters. A \ may be used to escape a ' symbol.
< *name*	The value of *name*, which is either a variable name or a register name. *Adb* maintains a number of variables (see the VARIABLES subsection) named by single letters or digits. If *name* is a register name, the the register value is obtained from the system header in *corfil*. The register names on the CRAY-2 computer system are **a0** through **a7**, **s0** through **s7**, **p**, **q**, **l**, **m**, **v000** through **v777**. The register names on the CRAY X-MP and CRAY-1 computer systems are **a0** through **a7**, **s0** through **s7**, **p**, **vl**, **vm**, **v000** through **v777**, **b00** through **b77**, and **t00** through **t77**.

symbol A sequence of upper or lowercase letters, underscores or digits, not starting with a digit. The value of *symbol* is taken from the symbol table in *objfil*.

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

Monadic operators:

**exp*	The contents of the location addressed by *exp* in *corfil*
@*exp*	The contents of the location addressed by *exp* in *objfil*
–*exp*	Integer negation
~ *exp*	Bitwise complement

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

exp1+exp2
Integer addition

exp1–exp2
Integer subtraction

*exp1*exp2*
Integer multiplication

exp1 %exp2
Integer division

exp1 &exp2
Bitwise conjunction

exp1 |exp2
Bitwise disjunction

exp1 #exp2
E1 rounded up to the next multiple of *e2*

Commands

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

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

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

=*f* The value of *address* itself is printed in the styles indicated by the format *f*

A *format* consists of one or more characters that specify a style of printing. Each format character may be preceded by a decimal integer and an asterisk (3**x*) 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, the previous format is used. The format letters available are as follows:

o	8	Prints 8 bytes in octal. All octal numbers output by *adb* are preceded by 0.
O	8	Prints 8 bytes in octal
q	8	Print in signed octal
Q	8	Prints long signed octal
d	8	Print in decimal
D	8	Prints long decimal
x	8	Prints 8 bytes in hexadecimal
X	8	Prints 8 bytes in hexadecimal
u	8	Prints as an unsigned decimal number

U 8	Prints long unsigned decimal
b 1	Prints the addressed byte in octal
c 1	Prints the addressed character
C 1	Prints the addressed character using the following escape convention. Character values 000 to 040 are printed as @ followed by the corresponding character in the range 0100 to 0140. The character @ is printed as @@.
s *n*	Prints the addressed characters until a zero character is reached
S *n*	Prints a string using the @ escape convention; *n* is the length of the string including its zero terminator.
Y 8	Prints 4 bytes in date format (see *ctime*(3C))
i n	Prints as instructions; *n* is the number of instructions.
a 0	Prints the value of *dot* in symbolic form
p 8	Prints the addressed value in symbolic form using the same rules for symbol lookup as **a** (refer back to **format** in the COMMANDS section)
t 0	When preceded by an integer, it tabs to the next appropriate tab stop. For example, **8t** moves to the next 8-space tab stop.
r 0	Prints a space
n 0	Prints a new line
"..." 0	Prints 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

Repeats the previous command with a *count* of 1

[?/]**l** *value mask*

Words starting at *dot* are masked with *mask* and compared with *value* until a match is found. If **L** is used, the match is for 8 bytes at a time instead of 2. If no match is found, *dot* is unchanged; otherwise, *dot* is set to the matched location. If *mask* is omitted, –1 is used.

[?/]**w** *value ...*

Writes the 2-byte *value* into the addressed location. If the command is **W**, writes 8 bytes. Word aligned addresses are required when writing to a subprocess address space.

[?/]**m** *b1 exp1 f1* [?/]

New values for (*b1*, *exp1*, and *f1*) are recorded. If less than three expressions are given, the remaining map parameters are left unchanged. If the **?** or **/** is followed by *****, the second segment (*b2*, *exp2*, and *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. (For example, **/m?** cause / to refer to *objfil*.)

>name *Dot* is assigned to the variable or register named

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

$modifier

Miscellaneous commands. The available *modifiers* are as follows:

<f	Reads commands from the file *f* and return
>f	Sends output to the file *f*, which is created if it does not exist
r	Prints the general registers. *Dot* is set to **pc**. For UNICOS on CRAY-2 computer system, **r** also prints the instruction addressed by **pc**.
b	Prints all breakpoints and their associated counts and commands
c	C stack backtrace. If *address* is given, it is taken as the address of the current frame. If *count* is given then only the first *count* frames are printed. For UNICOS running on CRAY-2 computer system, if **C** is used, the values of all automatic variables are

	printed for each active function
w	Sets the page width for output to *address* (default 80)
s	Sets the limit for symbol matches to *address* (default 255)
o	All integers input are regarded as octal
d	Resets integer input as described in EXPRESSIONS section
p	Changes to register set address
q	Exits from *adb*
u	Prints all nonzero variables in octal
m	Prints the address map
v	Prints vector registers at *address* for *count*. (CRAY-2 computer system only)
v*n*	Prints vector register *n*; default for *n* is 0. (CRAY X-MP and CRAY-1 computer system only)
j	Prints **b** registers. (CRAY X-MP and CRAY-1 computer system only)
k	Prints **t** registers. (CRAY X-MP and CRAY-1 computer system only)

:*modifier*

Manage a subprocess. Available modifiers are as follows:

b*c*	Sets 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 0, the breakpoint causes a stop.
d	Deletes breakpoint at *address*
r	Runs *objfil* as a subprocess. If *address* is given explicitly, 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 entry to the subprocess.
c*s*	The subprocess is continued with signal *s* (see *signal*(2)). If *address* is given, the subprocess is continued at this address. If no signal is specified, 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, *objfil* is run as a subprocess as for **r**. In this case no signal can be sent; the remainder of the line is treated as arguments to the subprocess.
k	The current subprocess, if any, is terminated

Variables

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

0	Last value printed
1	Last offset part of an instruction source.
2	Previous value of variable 1.

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

b	Base address of the data segment
d	Data segment size
e	Entry point
m	"Magic" number (0405, 0407, 0410, or 0411)
s	Stack segment size
t	Text segment size

p	Number of the current register set
q	Total number of register sets in the core file

For dumps that do not involve multiprocessing, var[VARP]=0 and var[VARQ]=1.

To change from one register set to another, *n*$p moves the register and Local Memory map so that register set *n* may be examined. The numbering starts from 0. Thus, 0 <= n < var[VARQ].

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, exp1, and f1*) and (*b2, exp2, and f2*). The second triple may be specified by following the ? or / with an *. The *address* file is calculated as follows:

if ((no *) and (*b1≤address<exp1*)), *file address=address+f1−b1*

else if ((*) and (*b2≤address<exp2*)), *file address=address+f2−b2*

else address is invalid.

For UNICOS running on CRAY-2 computer system, the second map for the / (core) file is set to permit accessing Local Memory.

For UNICOS running on CRAY-XMP and CRAY-1 computer systems, the second map for the / (core) file is the map for the data section if there is split code and data.

If either file is not of the kind expected, *b1* and *f1* are set to 0 and *exp1* is set to the maximum file size; in this way, the whole file can be analyzed with no address translation. All addresses are byte addresses, except for register number in a $v display for UNICOS running on CRAY-2 computer system.

FILES

a.out
core

MESSAGES

Exit status is 0, unless last command failed or returned nonzero status.

BUGS

A breakpoint set at the entry point is not effective on initial entry to the program.

SEE ALSO

ptrace(2) in the UNICOS System Calls Reference Manual, publication SR-2012
a.out(4F), core(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

admin – Creates and administers SCCS (Source Code Control System) 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

The *admin* command creates new SCCS files and changes parameters of existing ones. Arguments to *admin*, which may appear in any order, consist of keyletter arguments, which begin with –, and named files (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 nonSCCS 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. NonSCCS files, unreadable files, and directories 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 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 (EOF) is encountered. If this keyletter is omitted, then the SCCS file is created empty. 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). 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* 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 as follows:

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

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

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

dSID Default delta number (SID) to be used by a *get*(1) command

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

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

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

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

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

n Causes *delta*(1) to create a "null" delta in each of those releases (if any) being skipped when a delta is made in a *new* release (such as, 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 (see *passwd*(4F)), or numerical UNICOS system group ID (see *group*(4F)), to be added to the list of users which may make deltas (changes) to the SCCS file. A group ID is equivalent to specifying all *login* names common to that group ID. Several **a** keyletters may be used on a single *admin* command line. As many *login*s, or numerical group IDs, as desired may be on the list simultaneously. If the list of users is empty, 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 (that is 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). Messages 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*(4F)), and to compare a newly computed checksum (the sum of all the characters in the SCCS file except those in the first line) with the checksum that is stored in the first line of the SCCS file. Appropriate error messages are produced.

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

–z The SCCS file checksum is recomputed and stored in the first line of the SCCS file (see **–h**). 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 prevents simultaneous updates to the SCCS file by different users. See *get*(1) for further information.

MESSAGES

Use *help*(1) for explanations.

SEE ALSO

delta(1), ed(1), get(1), help(1), prs(1), what(1)
sccsfile(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

adstape – Prepares IOS deadstart tape

SYNOPSIS

/lib/adstape [**–i** *in*] [**–o** *names*] [**–v** *overlays*]

DESCRIPTION

The *adstape* command reads *in*, a collection of APML absolute binary records and writes *names*.

The *adstape* command accepts the following arguments:

names A comma-separated list of file names; there is one for each initial, nonoverlay program in the input file. In writing the output binary records, the Permanent Dataset Table (PDT) and the Task Execution Table (TXT) header are removed. The default *names* is **tape.boot,disk.boot,dsdump,kernel.**

–v *overlays*
If the input file contains overlays, they are written to the *overlays* file. As above, PDTs and the TXT header are removed Each overlay is preceded by a word count pointing to the next overlay. The default *overlays* is **overlays.**

–i *in* A file containing APMLbinary records ; the file should have been prepared by *bind*(1). The default *in* is **a.out.**

Error information is written to the standard output file. The operational procedures for the use of *adstape* are considerably different in COS.

MESSAGES

The exit code is nonzero if any problems are encountered.

SEE ALSO

apml(1)
The Operational Aids Reference Manual, publication SM-0044.

NAME

apml – Invokes the APML assembler

SYNOPSIS

/lib/apml [**–t** *bsys*] [**–r** *xref*] [**–g** *sym*] [**–l** *listing*] [**–m** *tmwords*] [–L] [**–s** *text(s)*] [**–h**] [**–i** *list*] [**–o** *binary*] *name.s*

DESCRIPTION

The *apml* command assembles the file you specified in *name.s* and places the binary object file in *name.o,* unless you have specified the **–o** option. The listing is written to the file *listing* if specified. The error messages and statistics information are written to standard error.

–t *bsys*	Names the output file to which *apml* writes the binary system text. There is no default.
–r *xref*	Names the output file to which *apml* writes the binary cross-reference text. You must supply *xref.*.If you omit *xref*, *apml* ignores the **–r** option.
–g *sym*	Names the output file to which *apml* writes the Symbol Table. You must supply the name of the output file. If you omit *sym*, *apml* ignores the **–g** option.
–l *listing*	Names the output file to which *apml* writes the assembler listing. The default is no listing.
–m *tmwords*	Specifies an integer number of memory words to be reserved for the table manager work area. The default is 65,476 words.
–L	Requests output of the statistical 'logfile' messages to **stderr**. If the –L option is used, the amount of excess work area is reported as UNUSED: *nnnnn*. It should not be necessary to increase this except on very large assemblies, such as I/O Subsystem. The work area is not expandable at runtime; therefore if sufficient space is not preallocated for the assembly to complete, *apml* aborts.
–s *text(s)*	Any number of system texts; must be separated by commas.
–h	Causes all list pseudos to be processed regardless of the location of the field name (the default is not enabled). If **–h** is enabled, **–i** is ignored.
–i *nlist*	Processes list pseudos whose location field names you specify in *nlist*. The *nlist* argument can be a single name or a list of names separated by commas.
–o *binary*	Names the binary object file. The default is *name.o* if *name.s* is the input.
name.s	Specifies the file containing the assembler source code.

FILES

binary	Binary object file
stderr	Warning, error, and statistics messages (statistics are provided only if requested by **–L**)
sym	Symbol table
name.s	Assembler input
xref	Binary cross-reference
bsys	Binary systems text; there is no default name
listing	Assembler listing (only if requested)
text(s)	Any number of systems texts (must be separated by commas)
/tmp/APML.?	Temporary intermediate files

BUGS

The –g option occasionally causes an assembly to fail.

SEE ALSO

The APML Assembler Reference Manual, CRI publication SM-0036

NAME

ar – Maintains archives and libraries for portable archives

SYNOPSIS

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

DESCRIPTION

The *ar* command maintains groups of files combined into a single archive file. It primarily creates and updates library files as used by the link editor.

The *ar* command accepts the following arguments:

Key One character from the set **d, r, q, t, p, m,** and **x** that can be optionally concatenated with one or more of the following: **v, u, a, i, b, c,** or **l**. The meanings of the *key* letters are as follows:

- **d** Deletes the named files from the archive file
- **r** Replaces the named files in the archive file. If the optional **u** character is used with **r,** then only those files with dates of modification later than the archive files are replaced.
- **q** Quickly appends the named files to the end of the archive file. Optional positioning characters are invalid. The command does not check if the added members are already in the archive. It is useful only to avoid quadratic behavior when creating a large archive piece-by-piece.
- **t** Prints a table of contents of the archive file. If no names are given, all files in the archive are tabled. If names are given, only those files are tabled.
- **p** Prints the named files in the archive
- **m** Moves the named files to the end of the archive. If a positioning character is present, the *posname* argument must be present and, as in **r**, specifies where the files are to be moved.
- **x** Extracts the named files. If no names are given, all files in the archive are extracted. In neither case does **x** alter the archive file.
- **v** Gives 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**, gives a long listing of all information about the files. When used with **x**, precedes each file with a name.
- **c** Suppresses the message that is produced by default when *afile* is created
- **l** Places temporary files in the local current working directory **/tmp**; this option causes them to be placed in the local directory.
- **z** Suppresses the feature of putting the filename into the PDT name

posname Specifies that new files are to be placed after (**a**) or before (**b** or **i**) *posname*. Otherwise, new files are placed at the end. *Posname* must be used with the **abi** key arguments.

afile Specifies the archive file

name Constituent files in the archive file, *afile*.

FILES

/tmp/ar*	Temporary files

BUGS

If the same file is mentioned twice in an argument list, it may be put in the archive twice. The module name from the PDT should be the same as the file name.

SEE ALSO

ld(1), lorder(1)
a.out(4F), relo(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

ar – Maintains archives and libraries for portable archives

SYNOPSIS

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

DESCRIPTION

The *ar* command maintains groups of files combined into a single archive file. 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.

The portable archive format and structure is described in detail in *ar*(4F).

The *ar* command accepts the following arguments:

Key An optional **–**, followed by one character from the set **drqtpmx**, optionally concatenated with one or more of **vuaibcls**. The meanings of the *key* characters are:

d Delete the named files from the archive file.

r Replace the named files in the archive file. If the optional character **u** is used with **r**, then only those files with dates of modification later than the archive files are replaced.

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. Useful only to avoid quadratic behavior when creating a large archive piece-by-piece.

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

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.

x Extract the named files. If no names are given, all files in the archive are extracted. In neither case does **x** alter the archive file.

v 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**, give a long listing of all information about the files. When used with **x**, precede each file with a name.

c Suppress the message that is produced by default when *afile* is created.

l 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**.

posname Specifies that new files are to be placed after (**a**) or before (**b** or **i**) *posname*. Otherwise, new files are placed at the end. *Posname* must be used with the **abi** key arguments.

afile Specifies the archive file

name Constituent files in the archive file, *afile*.

FILES

/tmp/ar*	Temporary files

SEE ALSO

ar(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NOTES

This archive format is not compatible with UNIX System V. It was changed in anticipation of the day when *segldr*(1) will be able to read archives.

BUGS

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

NAME

as – Invokes the Cray assembler CAL

SYNOPSIS

as [**–o** *objfile*] [**–l** *lstfile*] [**–L** *msgfile*] [**–b** *bdflist*] [**–B**] [**–c** *bdfile*] [**-D** *micdef*] [**–g** *symfile*] [**–G**] [**–C** *cpu*] [**–h**] [**–H**] [**–i** *nlist*] [**–I** *options*] [**–m** *mlevel*] [**–n** *number*] [**–f**] [**–F**] [**–j**] [**–J**] [**–V**] *filename*

DESCRIPTION

The *as* command assembles the named file. The following options, each a separate argument, can appear in any order but must precede the *filename* argument. If errors are encountered during assembly, a diagnostic message is issued to **stderr**.

–o *objfile* Relocatable assembly output; stored in *objfile* file. By default, the relocatable output file name is formed by removing the path name and the **.s** suffix, if they exist, from the input file and appending a **.o** suffix. *Objfile* must be processed by a link editor or loader.

–l *lstfile* Assembly output source listing; stored in *lstfile* file; default, the output source listing is suppressed.

–L *msgfile*
Assembly output message listing; stored in *msgfile* file; default, the output message listing is suppressed.

–b *bdflist* Reads the binary definition files stored in one or more files. The files named in *bdflist* can be designated using one of the following forms:

- List of files separated from one another by a comma
- List of files enclosed in double quotes and separated from one another by a comma and/or one or more spaces

Reads the default binary assembler definitions found in the file **/lib/asdef** unless suppressed with the **–B** option. The remaining files listed in *bdflist* are then read in the order in which they are specified.

–B Suppresses **/lib/asdef** as the default binary assembler definition file

–c *bdfile* Creates the binary definition file *bdfile*; by default, the creation of a binary definition file is suppressed.

–D *micdef*
Defines a globally-defined constant micro *mname* as follows:

micdef ::= *mname*[=[*string*]]

mname must be a valid identifier. If the = character is specified, it must immediately follow *mname*. The *string* that immediately follows the = character, if any, is associated with *mname*. If you do not specify the *string*, *mname* is associated with an empty string.

–g *symfile* Assembly output symbol file; stored in *symfile*. *Symfile* is used by the system debuggers. By default, the output symbol file is suppressed.

–G Forces all symbols to *symfile* if the **–g** option is used. Normally, nonreferenced symbols are not included.

–C *cpu* Code is generated for the specified CPU. The default is that code is generated for the characteristics of the host machine. The *cpu* option has the following syntax:

cpu ::= [*primary*] {, [*charac*] }
or
cpu ::= , [*charac*] { , [*charac*] }

primary *Primary* can be one of the following Cray computer systems:

Primary	Computer system	*Primary*	Computer system
cray-2	CRAY-2	cray-1m	CRAY-1M
cray-x4	CRAY X-MP models 48 and 416	cray-1s	CRAY-1S
cray-x2	CRAY X-MP models 22, 24, and 28	cray-1b	CRAY-1B
cray-x1	CRAY X-MP models 11, 12, 14, and 18	cray-1a	CRAY-1A
cray-xmp	CRAY X-MP	cray1	CRAY-1

charac Features of the *primary* computer. CRAY-2 computer systems have no special characters.

The CRAY X-MP and CRAY-1 computer systems let you specify the following logical and numeric traits:

Logical traits	Description	Logical traits	Description
avl	Additional vector logical	nocori	No control operand range interrupts
noavl	No additional vector logical	ema	Extended memory addressing
bdm	Bidirectional memory	noema	No extended memory addressing
nobdm	No bidirectional memory	hpm	Hardware performance monitor
cigs	Compressed index and gather/scatter	nohpm	No hardware performance monitor
nocigs	No compressed index and gather/scatter	pc	Programmable clock
cori	Control operand range interrupts	nopc	No programmable clock

Logical traits	Description	Logical traits	Description
readvl	Read vector length	vpop	Vector pop count
noreadvl	Do not read vector length	novpop	No vector pop count
statrg	Status register	vrecur	Vector recursion
nostatrg	No Status register	novrecur	No vector recursion

In the following table, n represents an unsigned decimal number.

Numeric traits	Description	Numeric traits	Description
bankbusy=n	Bank busy time in clock periods	memsize=n	Memory size in words
banks=n	Number of memory banks	memspeed=n	Memory speed in clock periods
clocktim=n	Clock time in picoseconds	numclstr=n	Number of cluster registers
ibufsize=n	Instruction buffer size in words	numcpus=n	Number of cpus

–h Enables all list pseudo instructions, regardless of the location field name.

–H Disables all list pseudo instructions, regardless of the location field name

–i *nlist* Restricts list pseudo instruction processing to those pseudo instructions whose location field names are given in *nlist*. The names specified by *nlist* can take one of the following forms:

- List of names separated from one another by a comma
- List of names enclosed in double quotes and separated from one another by a comma and/or one or more spaces

–I *options* List options. A list of more than one option must be specified without intervening blanks. It is not permitted to specify conflicting options (the same character in upper case and lower case) in the –I list. Options can be any of the following:

s Enables source statement listing (default)

S Disables source statement listing

e Enables edited statement listing (default)

E Disables edited statement listing

t Enables text source statement listing

T Disables text source statement listing (default)

l Enables listing control pseudo instructions

L Disables listing control pseudo instructions (default)

m Enables macro/opdef expansion

M Disables macro/opdef expansion (default)

d Enables dup/echo expansion

D Disables dup/echo expansion (default)

b Enables macro/opdef/dup/echo expansion binary only

B Disables macro/opdef/dup/echo expansion binary only (default)

c Enables macro/opdef/dup/echo expansion conditionals

C Disables macro/opdef/dup/echo expansion conditionals (default)

p Enables macro/opdef/dup/echo expansion of preedited lines

P Disables macro/opdef/dup/echo expansion of preedited lines (default)

x	Enables cross-reference listing (default)
X	Disables cross-reference listing
n	Enables nonreferenced local symbols included in the cross reference (default)
N	Disables nonreferenced local symbols included in the cross reference

–m *mlevel*

Message priority level for output source, message, and standard error file. *Mlevel* can be one of the following:

comment
note
caution
warning
error

By default, the message priority level is **warning**.

–n *number*

Maximum number of messages to be inserted into the output source, message, and standard error file. *Number* must be 0 or greater; the default is 100.

–f Enables the new statement format. By default, the old format is used when targeting for a CRAY X-MP or CRAY-1 computer system; otherwise, the new format is used. Statement format reverts to the format specified on the invocation statement at the end of every assembler segment.

–F Disables the new statement format. By default, the old format is used when targeting for a CRAY X-MP or CRAY-1 computer system; otherwise, the new format is used. Statement format reverts to the format specified on the invocation statement at the end of every assembler segment.

–j Enables editing; the default is enabled. Editing status reverts to the status specified on the invocation statement at the end of every assembler segment.

–J Disables editing; the default is enabled. Editing status reverts to the status specified on the invocation statement at the end of every assembler segment.

–V Causes the version number of the assembler being run and other statistical information (diagnostic messages of priority **comment**) to be written to the **stderr**.

filename Files to be assembled; all options must precede the *filename* argument.

FILES

/lib/asdef	Assembler binary definition file
tmpdir/as.?	Temporary file; *tmpdir* is a directory defined by $TMPDIR. Otherwise, a system default temporary directory is used.

SEE ALSO

cc(1), ld(1), nm(1)
tmpnam(3C) in the CRAY X-MP and CRAY-1 C Library Reference Manual, publication SR-0136
tmpnam(3S) in the CRAY-2 Computer System CRAY-2 UNICOS Libraries, Macros, and Opdefs Reference Manual, publication 2013
CAL Assembler Version 2 Reference Manual, publication SR-2003
Symbolic Machine Instructions Reference Manual, CRI publication SR-0085
CRAY X-MP and CRAY-1 CAL Assembler Version 2 Ready Reference, publication SQ-0083
CRAY-2 Computer System Functional Description, publication HR-2000
CRAY-2 CAL Assembler Version 2 Reference Card, publication SQ-2002

NAME

asa – Interprets ASA carriage control characters

SYNOPSIS

asa [*files*]

DESCRIPTION

The *asa* command 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 you do not supply any file names. The first character of each line is assumed to be a control character; the control characters have the following meanings:

' '	(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 correctly view the output of Fortran programs that use ASA carriage control characters, *asa* could be used as a filter thusly:

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

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

fsplit(1) nasa(1)

NAME

at, batch – Executes 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* 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 exported shell environment variables, current directory, **umask**, and **ulimit** are retained when the commands are executed. Open file descriptors, traps, and priority are lost.

Users are permitted to use *at* if their name appears in the file **/usr/lib/cron/at.allow**. If that file does not exist, the file **/usr/lib/cron/at.deny** is checked to determine if the user should be denied access to *at*. If neither file exists, only root is allowed to submit a job. The null file **at.allow** would mean no user is allowed to use **at**; a null file **at.deny** would mean no user is denied the use of **at**. The **allow/deny** files consist of one user name per line.

The *time* may be specified as 1, 2, or 4 digits. One-digit 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," except that it goes into a different queue. Also, "at now" responds 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
make 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 <<!
make 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/lib/cron/queue	Scheduling information
/usr/spool/cron/atjobs	Spool area

MESSAGES

At complains about various syntax errors and times out of range.

SEE ALSO

kill(1), mail(1), nice(1), ps(1), sh(1)
cron(1M) in the UNICOS Administrator Commands Reference Manual, publication SR-2022

NAME

awk – Scans and processes patterns

SYNOPSIS

awk [–Fc] [*prog*] [**–f** *file*] [*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. Each variable and field can be either a string or a number at any time. When a variable is set by the assignment v = expr its type is set to that of expr. (This includes +=, ++, etc.) An arithmetic expression is of type string, and so on. If the assignment is a simple copy, as in v1 = v2, then the type of v1 becomes that of v2. String constants are quoted (").

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

The expression in an action may contain any of several built-in functions. 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

WARNINGS

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.

SEE ALSO

grep(1), lex(1), sed(1)
UNICOS Support Tools Guide, publication SG-2016

NAME

banner – Makes posters

SYNOPSIS

banner *string* ...

DESCRIPTION

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

SEE ALSO

echo(1)

NAME

basename, dirname – Prints portions of pathnames on standard output

SYNOPSIS

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

DESCRIPTION

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

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

EXAMPLES

The following example, 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`
```

BUGS

On CRAY-2 UNICOS the *basename* of root (/) is null and is considered an error.

SEE ALSO

sh(1)

NAME

bc – Invokes the arbitrary-precision arithmetic language preprocessor

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.

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

The *bc* command accepts the following arguments:

–c	Specifies compile only
–l	Specifies the name of an arbitrary precision math library.

In the *bc* language, assignment to *scale* influences the number of digits to be retained on arithmetic operations in the manner of *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

The following table shows the syntax and functions of the *bc* language. The letters in the table (the syntax for *bc* programs) is as follows:

L	A letter a – z,
E	An expression
S	A statement

Type	Syntax/Function	Notes
Comments	strings enclosed in /* and /*	
Names	simple variables - L array elements - L [E] The words "ibase", "obase", and "scale"	Assignments to *ibase* or *obase* set the input and output number radix respectively.
Other operands	Arbitrarily long numbers with optional sign and decimal point (E) sqrt (E) Length (E) Number of significant decimal digits Scale (E) Number of digits right of decimal point L (E , ... , E)	
Operators	+ - * / % ^ ++ -- ==+ =- =* =/ =% =^	% is remainder; ^ is power Prefix and postfix; apply to names

Statements	E
	{S; ... ; S}
	if (E) S
	while (E) S
	for (E ; E ; E) S
	null statement
	break
	quit
		Either semicolons or new-lines can separate statements.
		
		The value of an expression statement is printed unless the main operator is an assignment.

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

Functions in -L math library	s(x) sine c(x) cosine e(x) exponential l(x) log a(x) arctangent j(n,x) Bessel function	All function arguments are passed by value.

EXAMPLE

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

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

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

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

FILES

/usr/lib/lib.b	Mathematical library
/usr/bin/dc	Desk calculator proper

LIMITATIONS

No **&&**, **| |** yet.

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

Quit is interpreted when read, not when executed.

SEE ALSO

dc(1)
UNICOS Support Tools Guide, publication SG-2016

NAME

bdiff – Compares very large files for differences

SYNOPSIS

bdiff *file1 file2* [*n*] [**–s**]

DESCRIPTION

The *bdiff* command 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 that are too large for *diff*. The *bdiff* command ignores lines common to the beginning of both files, splits the remainder of each file into *n*-line segments, and invokes *diff* upon corresponding segments. The value of *n* is 35,000 by default. The *bdiff* command accepts the following arguments:

file1	One of the files to compare
file2	The other file to compare. If *file1* or *file2* is –, *bdiff* reads the standard input.
n	If *n* is numeric, it is useful in those cases in which 35,000-line segments are too large for *diff*, casuing it to fail.
–s	Specifies that no diagnostics are to be printed by *bdiff* (however, this does not suppress possible exclamations by *diff*.)

If both optional arguments are specified, they must appear in the order indicated above.

Bdiff's output 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). Because of the segmenting of the files, *bdiff* does not necessarily find a smallest sufficient set of file differences.

FILES

/tmp/bd?????

MESSAGES

Use *help*(1) for explanations.

SEE ALSO

diff(1)

NAME

bind – Binds APML relocatable modules together

SYNOPSIS

/lib/bind [**–i** *inputfile*] [**–o** *outputfile*]

DESCRIPTION

Bind reads *inputfile*, a collection of APML binary records containing external references, and writes *outputfile*, a copy of the binary records with external references resolved.

The default *inputfile* is **a.o**. The default *outputfile* is **a.out**. Error information regarding duplicate or undefined symbols is written to standard output.

The exit code is nonzero if any undefined symbols were encountered during the binding process.

SEE ALSO

The COS Operational Aids Reference Manual, publication SM-0044

BUGS

Bind does not accept an archive as input.

NAME

bmxio – Provides an interface to block mux devices

SYNOPSIS

bmxio [–io] [**-t** *type*] [**-d** *dens*] [**-v** *vsn0*:*vsn1*:...] *fn*

DESCRIPTION

Bmxio provides a means to move data between a file and a block mux device (most often a tape).

Bmxio selects a device of the requested type; if it is a tape device, *bmxio* issues a mount message on the system console. The device is then read or written according to arguments you specify; any of the following:

–i	Input is taken from the block mux device and written to the specified output file
–o	Input is taken from the provided file and written to a block mux device
–t	Generic device type (the default is TAPE)
–d	Tape density; default is 6250 b/i; may specify 1600.
–v	List of volume names (*vsn*) separated by colons (:); required on input.
fn	File name (the default is **stdin / stdout**)

Bmxio supports multivolume, single file tape handling. On output, when the end of tape is detected, *bmxio* selects another device, issues a mount message, and continues output until EOF is detected on the input file. On input, when the end of volume is detected, *bmxio* checks for the next volume (*vsn*) in the list provided and continues reading until all volumes are processed.

NOTES

Labelled tapes are not currently supported.

Block mux devices are treated as system resources and are opened and closed by the super user upon initiazation.

NAME

cal – Prints a calendar

SYNOPSIS

cal [[*month*] *year*]

DESCRIPTION

The *cal* command prints a calendar for the present month, if you do not specify a year and a month. If you specify a year, *cal* prints a calendar for all twelve months of the year. If you also specify a month with the year, a calendar just for that month is printed.

Following are the specifications for the arguments:

year	A number representing a year, between 1 and 9999.
month	A number between 1 and 12, representing a month.

The calendar produced is that for England and her colonies. Note that "cal 85" refers to the early Christian era, not the 20th century. The year is always considered to start in January even though this is historically naive.

Try 9 (for September) 1752.

NAME

calendar – Reminder service

SYNOPSIS

calendar [–]

DESCRIPTION

The *calendar* command consults the **calendar** file in your 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," or "8/24" are recognized, but not "24 August" or "24/8". On weekends "tomorrow" extends through Monday.

When you specify the argument, *calendar* does its job for every user who has a file **calendar** in their login directory and sends them any positive results by *mail*(1). Normally, this is done daily by facilities in UNICOS.

FILES

/usr/lib/calprog	Program to figure out today's and tomorrow's dates
/etc/passwd	List of login directories
/tmp/cal*	Temporary files

BUGS

Your calendar must have public permission (see *chmod*(1)) for you to get reminder service.

Calendar's extended idea of "tomorrow" does not account for holidays.

SEE ALSO

mail(1)

NAME

cat – Concatenates and prints files

SYNOPSIS

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

DESCRIPTION

The *cat* command reads each *file* you specify on the command line in sequence and writes it to standard output. Thus:

```
cat file
```

prints *file*, and:

```
cat file1 file2 >file3
```

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

If you do not specify an input file, or if cat encounters the argument –, *cat* reads from standard input. The following options are available;

–u Causes the output to be unbuffered

–s Makes *cat* silent about nonexistent files

–v Prints nonprinting characters (with the exception of tabs, new-lines and form-feeds) visibly. Here, control characters are printed as ^X (CONTROL-*X*); the DEL character (octal 0177) is printed as ^? . Non-ASCII characters (with the high-order bit set) are printed as **M-***x*, where *x* is the character specified by the 7 low-order bits.

–t When used with the –v option, **–t** causes tabs to be printed as ^I's and form feeds to be printed as ^L's; *Cat* ignores this option if you do not specify the –v option.

–e Prints a **$** character at the end of each line (prior to the new-line). *Cat* ignores this option if you do not specify the –v option.

WARNING

Command formats such as

```
cat file1 file2 >file1
```

cause the original data in *file1* to be lost.

The *cat* command may hang if the file it attempts to write to the terminal is a binary file.

SEE ALSO

cp(1), pg(1), pr(1), more(1)

NAME

cb – C program beautifier

SYNOPSIS

cb [**–s**] [**–j**] [**–l** *length*] [*file...*]

DESCRIPTION

The *cb* command 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.

The *cb* command accepts the following arguments:

–s Changes the style of the code to conform to the style of Kernigham and Ritchie in *The C Programming Language*

–j Causes split lines to be put back together

–l *length*
Causes *cb* to split lines that are longer than *length*

WARNING

Punctuation that is hidden in preprocessor statements causes indentation errors.

SEE ALSO

cc(1)

NAME

cc – Invokes the C compiler

SYNOPSIS

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

DESCRIPTION

Cc is the UNICOS system C compiler. It accepts several types of arguments.

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

In the same way, arguments whose names end with **.s** are taken to be assembly source programs and are assembled, producing a **.o** file. The assembler used is not *as*(1) but a limited-purpose assembler intended for the output of the compilation phase of the C compiler (see the **–S** option).

The following options are interpreted by *cc*. See *ld*(1) for link editor options and *cpp*(1) for more preprocessor options.

–o *outfile*
: Produce an object file by the name *outfile*. The default name is **a.out**.

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

–f Compile C code into object code that does not use floating-point instructions, and link with appropriate library.

–O Invoke an object-code optimizer.

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

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

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

–D*name*[=*def*]
: Define *name* as if a **#define** directive. If no *=def* is given, *name* is defined as 1.

–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 "" are searched for first in the directory of the *file* argument, then in directories named in **–I** options, and last in directories on a standard list. For **#include** files whose names are enclosed in <>, the directory of the *file* argument is not searched.

–U*name* Remove any initial definition of *name*, where *name* is a reserved symbol that is predefined. The predefined symbols on the CRAY-2 Computer System are **CRAY**, **CRAY2**, and **unix**.

–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.

–# Produce information-only output indicating the actions *cc* would take based on the arguments provided. Note that # is the shell *sh*(1) comment character, but the adjacent – causes it to be treated differently.

–B*string* Construct path names for substitute preprocessor, compiler, optimizer, assembler and link editor passes by concatenating *string* with the suffixes **cpp**, **ccom**, **cci**, **ccas**, and **ld**. If *string* is empty it is taken to be **/lib/o**. By default, only **ccp**, **ccom**, and **cci** are modified by this option (see the **–t** option).

–t[p02al]
Find only the designated preprocessor (**p**), compiler (**0**), optimizer (**2**), assembler (**a**), and link editor (**l**) passes in the files whose names are constructed by a **–B** option. In the absence of a **–B** option, the *string* is taken to be **/lib/n**. The value **–t** "" is equivalent to **–tp02**.

–W*c,arg1[,arg2...]*
Hand off the arguments to pass *c* where *c* is one of [p02al] indicating preprocessor, compiler, optimizer, assembler, or link editor, respectively.

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

The AT&T C language standard was extended to include arbitrary length variable names. This portion of the standard has not been implemented on the CRAY-2 compiler.

FILES

file.c	input file
file.o	object file
a.out	linked output
/tmp/ctm*	temporary
/usr/tmp/ctm*	temporary
/lib/cpp	C preprocessor *cpp*(1)
/lib/ccom	compiler
/lib/ccas	Cray C compiler
/lib/pci	optional optimizer
/usr/lib/Oc*	backup compiler, *Occ*
/bin/as	assembler, *as*(1)
/bin/ld	link editor, *ld*(1)
/lib/crt0.o	runtime startoff
/lib/libc.a	standard C library, see the CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual, publication SR-2013
/lib/libm.a	math library, see the CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual, publication SR-2013
/lib/lmset	local memory definitions input to link editor

NOTES

The location of compiler and temporary files can be controlled by setting and exporting the environment variable **TMPDIR** (see *sh*(1) and *tmpnam*(3)).

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.

MESSAGES

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

SEE ALSO

adb(1), cpp(1), as(1), ld(1).
exit(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

cc – Invokes the C compiler

SYNOPSIS

cc [**–o***outfile*] [**–c**] [**–h***options*] [**–W**] [**–S**] [**–g**] [**–D***name*[=*def*]] [**–E**] [**–I***directory*] [**–O**] [**–P**] [**–U***name*] [**–C**] ... *file* ...

DESCRIPTION

The *cc* command invokes the UNICOS C compiler.

Cc expects several optional arguments followed by a list of one or more files. Files whose names end with **.c** are taken to be **C** source programs; they are compiled, and each object program is left in the file whose name is that of the source with **.o** substituted for **.c**. The **.o** file is normally deleted.

Similarly, all files whose name ends with **.s** are taken to be CAl; they are assembled, and each object program is left in the file whose name is that of the source with **.o** substituted for **.s**.

Any files whose name ends with **.o** are passed on to *ld*(1) and are loaded with all other **.o** files that were compiled or assembled to produce an executable program with the name **a.out**.

The *cc* command accepts the following options:

–o*outfile* Passed to *ld*. Override the default output file name, **a.out**, to be *outfile*.

–c Do not call *ld*. Force the object files to be produced and leave them in their respective **.o** files.

–W Passed to *as*(1)

–h*options* Pass the C compiler code generation options on the the C compiler. The following options can be used, with multiple arguments separated by a comma and no intervening spaces:

ema (noema)
This option causes the compiler to generate code sequences which will allow memory references up to addresses requiring a full 24-bits. The default is **noema**, which disables 24-bit addresses.

fastmd (nofastmd)
This option generates code sequences for **int** variables which are shorter when doing multiply or divide operations but allow for only 46 bits of significance. Without this option, **int** variables use the much longer (for 64 bits of significance) code sequences used by **long** variables for multiply and divide operations. The default is **fastmd**, which requires **int** variables to use the long multiply/divide code sequences.

abort (noabort)
This option controls whether a compilation does or does not abort if a fatal error is detected. The default is **abort**.

btreg (nobtreg)
This option generates optimization code sequences involving B and T register sets in specific instances. The default is **btreg**.

varargs (novarargs)
This option is not presently implemented.

–S Compiles the named C programs and leaves their assembly language output in the corresponding files suffixed **.s**.

–g Create a debug symbol table to be used with symbolic debuggers. Deferred implementation

–D*name*[=*def*]
Passed to the C preprocessor. Define *name* as if by a **#define** directive. If no =*def* is given, *name* is defined as 1.

–E Run only the C preprocessor and send the result to standard output.

–I*directory* Passed to the C preprocessor. Change the algorithm for searching for **#include** files whose names do not begin with / to look in *directory* 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 input file, then in directories named in **–I** options, and finally in the standard directories. For **#include** files whose names are enclosed in <>, the directory of the input file is not searched.

–O Invoke the optimizer. Deferred implementation

–P Run only the C preprocessor, and leave the result in the corresponding file suffixed with **.i**.

–U*name* Passed to the C preprocessor. Undefine *name* as if by a **#undef** directive.

–C Passed to the C preprocessor. By default, the C preprocessor strips C comments. If the –C options is specified, all comments are passed along to the C compiler. Only those comments on preprocessor directive lines are not passed along.

–# Produce information-only output indicating the actions *cc* would take based on the arguments provided. Note that # is the shell (*sh*(1)) comment character, but the adjacent – causes it to be treated differently.

FILES

file.c	Input C source file
file.s	Assembly language file
file.o	Object file
a.out	Executable output file
CAL	Temporary file for compiler output
/bin/cpp	C preprocessor
/bin/comp	Cray C compiler
/lib/libc.o	C compiler library
/bin/as	CRAY X-MP and CRAY-1 assembler
/bin/ld	CRAY X-MP and CRAY-1 loader

MESSAGES

The error messages produced by the C compiler are intended to be self-explanatory. If further explanation is required, please see the COS Message Manual, publication SR-0039.

SEE ALSO

as(1), ld(1), segldr(1)
Cray C Reference Manual, publication SR-2024.

NAME

cd – Changes working directory

SYNOPSIS

cd [*directory*]

DESCRIPTION

If *directory* is not specified, the value of the shell parameter **$HOME** (your home directory) is used as the new working directory. If *directory* specifies a complete path starting with /, ., or .., *directory* becomes the new working directory. If neither case applies, *cd* tries to find the designated directory relative to one of the paths specified by the **$CDPATH** shell variable. If **$CDPATH** is not defined, the search path defaults to **.** (dot). **$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 built in to the shell (see *sh*(1)).

SEE ALSO

pwd(1), sh(1)
chdir(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

cdc – Changes the delta commentary of an SCCS delta

SYNOPSIS

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

DESCRIPTION

The *cdc* command changes the *delta commentary*, for the *SID* specified by the **–r** keyletter of each named SCCS file. A *delta commentary* is defined to be the Modification Request (**MR**) and comment information normally specified via the *delta*(1) command (**–m** and **–y** keyletters).

If you specify a directory, *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 (SCCS) User's 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.

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.

FILES

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

MESSAGES

Use *help*(1) for explanations.

SEE ALSO

admin(1), delta(1), get(1), help(1), prs(1)
sccsfile(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014
the UNICOS Source Code Control System (SCCS) User's Guide, publication SG-2017

NAME

cflow – Generates C flow graph

SYNOPSIS

cflow [**–r**] [**–ix**] [**–i_**] [**–d***num*] *files*

DESCRIPTION

Cflow analyzes a collection of C, YACC, and LEX 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.) The output of all this nontrivial processing is collected and turned into a graph of external references, which is displayed on standard output.

Each line of output begins with a reference (that is, 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 (for example, **char ***), and, delimited by angle brackets, the name of the source file and the line number where the definition was found.

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 is found. For undefined references, only < > is printed.

EXAMPLES

Given the following in *file.c*:

```
int     i;

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

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

the command

```
cflow –ix file.c
```

produces the output

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

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

The following options are interpreted by *cflow*:

–r	Reverses 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	Includes external and static data symbols. The default is to include only functions in the flowgraph.
–i_	Includes names that begin with an underscore. The default is to exclude these functions (and data if **–ix** is used).
–dnum	Indicates the depth at which the flowgraph is cut off. By default, *num* is a very large number. Attempts to set the cutoff depth to a nonpositive integer are met with contempt.

MESSAGES

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

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.

SEE ALSO

cc(1), cpp(1), lex(1), lint(1), pr(1), yacc(1)

NAME

cft – Invokes the CRAY-2 Fortran compiler

SYNOPSIS

cft [**–b** *binfn*] [**–c** *calfn*] [**–t** *trncnt*] [**–d** *oplist*] [**–e** *oplist*] [**–i** *intsize*] [**–l** *listfn*] [**–m** *mlev*] [**–M** *maxblock*] [**–o** *optlst*] [– –] [*fname.f*]

DESCRIPTION

The CRAY-2 Fortran compiler is loaded and executed when a *cft* command is encountered in the **stdin** file.

You can specify options in any order. If you omit an option from the statement, the compiler uses the default value indicated in the following explanation. The compiler input file (see *fname*.**f**) is not optional; it must appear as the last item on the command line. Options are:

–b *binfn* Binary object code file name; *fname*.**o** is the default (see *fname*.**f**).

–c *calfn* Alternate CAL listing file. The **–eC** option writes pseudo CAL to *fname*.**s**. The default is no file.

–t *trncnt* Specifies the number of bits to be truncated for floating-point results (0-47)

–d –e *oplist*

Option argument list for use with the **–d** and **–e** options. Arguments in *oplist* following the **–d** option are disabled; those in *oplist* following the **–e** option are enabled. *Oplist* arguments must not appear in both a **–d** and **–e** argument list. If more than one argument appears in *oplist*, the argument list can be delimited by quotation marks for readability; arguments within the quotation marks can be separated by commas and blanks, or not separated at all. If *oplist* is not delimited by quotation marks, arguments can be separated by commas for readability; no other separator can be used. If an option argument is not specifically enabled or disabled, the compiler uses the default. Specific *oplist* arguments and defaults are:

a Aborts job after compilation if any program unit contains a fatal error; disabled by default.

b Lists beginning sequence number of each code generation block (g implies **b**); disabled by default.

B Controls generation of binary file to *fname*.**o** in the current directory or to *binfn*, if **–b** is specified; enabled by default.

c Lists common block names and lengths after each program unit; enabled by default.

C Creates pseudo CAL filename *fname*.**s** in the current directory unless an alternate filename is given by the **–c** option; disabled by default.

d Lists DO loop table; disabled by default.

e Controls recognition of compiler directive lines; enabled by default.

f Enables flowtrace processing

g Lists generated code for each program unit (see **CODE/NOCODE** directives); disabled by default.

h Lists the first statement of each program unit and error messages; all other list options in *oplist* are ignored or disabled. The **h** argument is disabled by default.

i	Enters compiler-generated statement labels in the symbol table; disabled by default.
j	Causes at least one execution of all DO loops; disabled by default.
l	Controls recognition of output listing control directives; enabled by default.
L	Controls generation of a listing file to *fname*.l in the current directory or to *listfn* if the **–l** option is specified; enabled by default.
n	Enters null symbols in symbol table (defined but not referenced); disabled by default.
p	Allows double precision; enabled by default. Disabling **p** (that is, specifying **p** in *oplist* following **–d**) at compile time causes the following:

1. All double-precision declaratives to be treated as real.
2. Double-precision functions to be changed to corresponding single-precision functions.
3. Double-precision constants to be converted as double precision and truncated to real.
4. D format edit descriptors to be changed to E edit descriptors.

q	Aborts compilation when 100 fatal error messages are counted; enabled by default.
s	Lists Fortran source code; enabled by default.
S	Causes compilation to proceed as if a SAVE statement was in every program unit.
t	Lists symbol table after each program unit; enabled by default.
v	Vectorizes inner DO loops; enabled by default.
x	Lists symbol table with cross references after each program unit (**x** overrides **t**); disabled by default.
z	Causes output of debug symbol table to binary **fname.o**.

–i *intsize* Integer size; default is 32 bits. Valid options are 32 (integer) and 64 (long integer).

–l *listfn* File to which the output listing is written. Listing options (for example, cross references, symbol tables, source code) are controlled by arguments in *oplist*. The default listing file name is *fname*.l.

–M *maxblock*
Length of code block being optimized or vectorized; default is 2560 words of intermediate text. Values greater than 2560 may increase optimization and internal compiler errors. A value of 1 eliminates vectorization and optimization between Fortran source statements.

–m *mlev* Message level; the highest severity level of CRAY-2 Fortran-produced messages to be suppressed ($0 \leq mlev \leq 4$). For example, **–m 2** allows CAUTION, WARNING, and ERROR messages to be issued. **–m 0** means no suppression takes place. The default is **–m 3**. The severity levels are defined as follows:

Message level	Severity type	Description
0	COMMENT	Comments on programming inefficiencies.
1	NOTE	May cause problems with other compilers
2	CAUTION	Possible user error (example: no path to this statement)
3	WARNING	Probable user error (example: using an array with too few subscripts)
4	ERROR	Fatal error. Fatal errors cannot be suppressed.

–o *optlst* Specifies optimization option. When selecting multiple options, separate values by commas. The options are:

nozeroinc Assumes constant increment integers are not incremented by variables with value 0. This is the default option.

zeroinc Assumes constant increment integers can be incremented by variables with the value 0. This option inhibits the vectorization of any DO loop with CIIs of the form CII = CII + VARIABLE.

lmlv Assigns local variables to local memory saved variables, equivalenced variables; variables used as actual arguments are excluded. This is the default option.

nolmlv Disables the **lmlv** option.

noifcon Disables conditional replacement optimization (default)

partialifcon Optimizes conditional replacement statements.

-- Optional symbol used to delimit the end of the options; the compiler considers whatever follows this symbol to be *fname*.**f**.

fname.**f** Compiler input file containing the Fortran source code to be compiled. The input file name must end in **.f**. If the **–l** option is omitted and the **–d** option does not contain the **L** option argument, the compiler output listing is written to the file *fname*.**l**. Similarly, if the **–b** option is not specified and **B** is not an argument to option **–d**, the binary object file is written to *fname*.**o**. The input file name must follow all other command line options.

FILES

fname.o Compiler binary object code output file. This file may be redirected using the **–b** option. The **–d** option with the *oplist* argument **B** enables the writing of this file.

fname.f CRAY-2 Fortran input Fortran source code file.

fname.l Compiler listing output file. This file may contain a source listing, cross references, generated code listing, error messages and diagnostics, and other listing options controlled by the **–e** and **–d** options. The listing file may be redirected using the **–l** or disabled using the **L** argument with the **–d** option.

fname.s Pseudo CAL filename; not written by default.

stderr Command line errors are written to **stderr**. If the compiler listing is disabled (with **–d l**), compile time errors and diagnostics are written to **stderr**.

MESSAGES

A full list of compiler diagnostics can be found in the CRAY-2 Fortran Reference Manual, publication SR-2007.

EXAMPLES

cft in.f

In this example, the file **in.f** is compiled and a binary object file is written to the file **in.o**. A listing file including source code, fatal and warning messages, symbol table, and list of common block names and length is written to the file **in.l**. The default optimization option **nonzeroinc** is set. Vectorization of inner DO loops, DOUBLE PRECISION, and recognition of compiler directive lines is enabled. Floating-point operation results are not truncated. Compilation terminates if more than 100 fatal errors are found.

cft –b binary –dLelpv –e"j q" –m 4 –ozeroinc — file.f

In this example, the file **file.f** is compiled, creating the binary object file name binary. Only fatal errors are written to **stderr**. No listing file is created. All compiler directive lines are ignored. All DO loops execute at least once, but do not vectorize. DOUBLE PRECISION is treated as REAL. The optimization option **zeroinc** is assumed.

cft –d B,L –m 0 input.f

In this example, the file **input.f** is compiled. No listing file or object file is created. All levels of compiler messages are written to **stderr**. Listing options are disabled, and all other compiler options are set to their default setting.

NAME

cft – Invokes the CFT Fortran compiler

SYNOPSIS

cft [**–a** *alloc*] [**–b** *binfile*] [**–c** *calfile*] [**–d** *oplist*] [**–e** *oplist*] [**–i** *intlen*] [**–l** *listfile*] [**–m** *mlev*] [**–o** *options*] [**–t** *trunc*] [**–u** *unroll*] [**–v** *msgs*] [**–A** *aids*] [**–C** *type,characteristic*] [**–E** *errfile*] [**–M** *maxblock*] [– –] *filename*

DESCRIPTION

The symbol – – may be used to delimit the end of the options. The Fortran source code to be compiled must be *filename*. *Filename* must end in **.f**. If the **–b** option is omitted, the default binary file is written to *filename*.**o**. The error messages and statistics information are written to standard error, stderr, unless **–eE** or **–E** *errfile* is specified. The *cft* command accepts the following options are:

–a *alloc* Allocation mode; *alloc* can be equal to **static** or **stack**. Static allocation is the default.

–b *binfile* Alternate binary object file. Default is *filename*.**o**. **–d B** disables binary object creation.

–c *calfile* Alternate CAL listing file. **–e C** writes pseudo CAL to *filename*.**s**. Default is no file.

–{d|e} *oplist*

Arguments in *oplist* following the **–d** option will be disabled, those in *oplist* following the **–e** option will be enabled. The arguments in the *oplists* must not appear in both the **d** and **e** *oplists*. The options are:

A	Enables non-ANSI messages to be printed at compile time. Disabled by default.
B	Creates binary object file *filename*.**o**. Enabled by default.
C	Creates pseudo CAL file *filename*.**s**. Disabled by default.
D	Writes sequence number labels at each executable Fortran statement to the Debug Symbol Table, allowing Breakpoints to be set with SID at statement sequence numbers. This option forces **–e iz** and **–M 1**. Disabled by default.
E	Creates an error listing file in *filename*.**e**. Disabled by default.
I	Sets all uninitialized stack variables to an undefined value. Disabled by default.
L	Creates a listing file *filename*.**l**. Disabled by default.
S	*Saveall* option allows compilation to occur as if a SAVE statement with an empty list were in each program unit. This option overrides **–o btreg**. Disabled by default.
a	Aborts job after compilation if any program unit contains compilation errors. Disabled by default.
b	Lists beginning sequence number of each code generation block (**g** implies **b**). Disabled by default.
c	Lists common block names and lengths after each program unit. Enabled by default.
d	Lists DO-loop table. Disabled by default.
e	Enables recognition of compiler directive lines. Enabled by default.
f	Enables the *flowtrace* option. Disabled by default.
g	Lists generated code for each program unit (see **CODE** and **NOCODE** compiler directives). Disabled by default.

h	Causes listing of the first statement of each program unit and error messages. All other list options in oplist are ignored or disabled. Disabled by default.
i	Enters compiler-generated statement labels in the symbol table. Disabled by default.
j	Causes all DO loops to execute at least once. Disabled by default.
l	Enables recognition of output control directives. Disabled by default.
m	Generates a table of machine characteristics. Disabled by default.
n	Enters null symbols in symbol table (defined but not referenced). Disabled by default.
o	Prints a message identifying any array references with out-of-bounds subscripts found during execution. Enables the BOUNDS compiler directives. Disabled by default.
p	Allows **double precision**. Enabled by default. Disabling **p** causes at compile time: 1. All **double precision** declarations to be treated as **real**. 2. **Double precision** functions to be changed to corresponding single precision functions. 3. **Double precision** constants to be converted as **double precision** and truncated to **real.** 4. **D format** edit descriptors to be changed to **E format** descriptors.
q	Aborts compilation when 100 fatal error messages have been issued. Enabled by default.
r	Rounds results on multiply operations. Enabled by default.
s	Lists Fortran source code. Enabled by default.
t	Lists symbol table after each program unit. Enabled by default.
u	Controls **int24** usage. Enabled by default.
v	Attempts to vectorize inner DO loops. Enabled by default.
x	Lists symbol table with cross reference after each program unit (**x** overrides **t**). Disabled by default.
z	Writes the Debug Symbol Table. Disabled by default.

–i *intlen* Specifies length of integers. "64" implies 64-bit integers. This is the default option. "24" implies 24-bit integers.

–l *listfile* Alternate source listing file. Default is no source listing. **–e L** enables a source listing file with the name *filename*.**l**.

–m *mlev* Highest message level to be suppressed; default is 3. Fatal errors are not suppressed.

Level	Severity	Description
0	COMMENT	Comments on programming inefficiencies
1	NOTE	May cause problems with other compilers
2	CAUTION	Possible user error
3	WARNING	Probable user error
4	ERROR	Fatal error

–o *options* Specifies optimization options. Only one from each group of options may be selected. If more than one argument is specified, arguments must be separated by commas. The option groups are:

nozeroinc	Assumes constant increment integers are not incremented by variables with the value 0 (default option).
zeroinc	Assumes constant increment integers can be incremented by variables with the value 0. This option inhibits the vectorization of any DO loop in which there are CIIs of the form CII=CII+*variable*.
noifcon	Disables optimization of conditional replacement statements of the form IF(*logical exp*) *var=expression* except where CFT replaces these statements with MIN/MAX intrinsic functions (default option).
partialifcon	Allows CFT to optimize conditional replacement statements of the form IF(*logical exp*) *var=expression* if *var* is of type real, integer, or logical, and *expression* does not involve division or an external function reference. The optimization causes CFT to generate code similar to *var*=CVMG*x*(*expression,var,condition*). If the optimization is performed, the IF statement will not inhibit vectorization or break an optimization block.
fullifcon	Allows CFT to optimize conditional replacement statements as described for PARTIALIFCON, except conditional replacement statements involving division and external functions are also optimized.
fastmd	Causes CFT to use the fast integer multiply and divide algorithms. Operands and results are limited to 46 bits; there is no overflow detection.
slowmd	Causes CFT to generate the full 64 bit integer multiply and divide (default option).
safedorep	Enables replacement of 1-line DO loops with a call to a $SCILIB routine performing the same operation more efficiently . Replacement does not occur when a one-line DO loop contains potential dependencies or equivalenced variables (default option).
fulldorep	Enables replacement of 1-line DO loops with a call to a $SCILIB routine. Potential dependencies and equivalences are ignored.
nodorep	Disables replacement of 1-line DO loops. **Nodorep** has no effect on vectorization of loops in the program.
invmov	Enables movement of invariant code from DO loops (default option).
noinvmov	Disables movement of invariant code from DO loops.
unsafeif	Enables instructions to move over a branch instruction.
safeif	Disables instructions moving over a branch instruction (default option).
bl	Enables scalar loops to be bottom loaded (default option).
nobl	Disables bottom loading of scalar loops.
btreg	Causes CFT to allocate specific scalar variables in a program unit to a T registers.
nobtreg	Causes CFT to allocate all user variables to memory. **Nobtreg** does not affect the allocation of compiler-generated variables to B or T registers or the use of B or T registers temporarily holding values during expression evaluation (default option).

cvl	Enables compilation of conditional vector loops (default option).
nocvl	Disables compilation of conditional vector loops.
keeptemp	Updates scalar temporary variables in DO loops (default option).
killtemp	Does not update scalar temporary variables in DO loops. The variable values will be undefined when the DO loop terminates.

–t *trunc* Number of bits truncated for floating-point results. Truncated bits are zeroed. Range is 0<=*trunc*<=47. Default is 0.

–u *unroll* Specifies that inner DO loops with constant limits iterating *unroll* times or less may use DO-loop unrolling. The maximum value for *unroll* is 9. Default is 3. **–u 0** turns off DO-loop unrolling.

–v *msgs* Enables the LOOPMARK utility. If *msgs* is **messages** an explanation as to what was done to the loop will be printed. If *msgs* is **nomessages** no explanation is given. *Msgs* is required. The LOOPMARK utility is disabled by default.

–A *aids* Determines the number of vectorization inhibition messages to be printed when the *aids* argument is equal to :

loopnone	No messages
looppart	3 per compiler block; 100 per compilation. (default)
loopall	All messages are issued

–C *type,characteristics*

Specifies the mainframe type running the binary object. This defaults to the CPU type of the current machine

cray-1m	Generates code for the CRAY-1 M computer systems.
cray-1, cray-1a, cray-1b, cray-1s	Generates code for the CRAY-1 S computer systems.
cray-xmp, cray-x1, cray-x2, cray-x4	Generates code for the CRAY X-MP computer systems.

Optional machine characteristics can be specified following the machine type. Specifying a machine characteristic requires specifying a machine type.

[no]avl	Target machine does/does not have two vector logical functional units.
[no]bdm	Target machine does/does not have bidirectional memory.
[no]cigs	Target machine does/does not have compressed index gather/scatter hardware.
[no]ema	Enables/disables extended addressing capability.
[no]vpop	Enables/disables vector pop count capability.
[no]vrecur	Target machine does/does not have vector recursion.
ibufsize=*words*	Instruction buffer size in words (16 or 32).
memspeed=*cps*	Memory speed in clock periods.

–E *errfile* Alternate error listing file. **–e E** writes errors to *filename*.e. Default is **stderr**.

–M *maxblock*

Length of code block being optimized or vectorized. Default is 4000 words of internal intermediate text. Values greater than 4000 may increase optimization and internal compiler errors. *Maxblock*=1 eliminates optimization and vectorization.

FILES

filename.e	Error listing file
filename.f	Fortran source file; must always be specified.
filename.l	Source listing filename; default is no filename.
filename.o	Binary object filename
filename.s	Pseudo CAL file name; default is no filename.

MESSAGES

The full range of CFT diagnostics can be found in the Fortran (CFT) Reference Manual, publication SR-0009.

SEE ALSO

The Fortran (CFT) Reference Manual, publication SR-0009

NAME

cft77 – Fortran compiler, not machine-specific

SYNOPSIS

cft77 [**-a** *alloc*] [**-b** *binfile*] [**-d** *offstrng*] [**-e** *onstrng*] [**-i** *intlen*] [**-l** *listfile*] [**-m** *msglev*] [**-o** *optim*] [**-s** *calfile*] [**-t** *trunc*] [**-C** *cpu,hdw*] [--] *file.f*

DESCRIPTION

The *cft77* command invokes the CFT77 compiler. Keywords in the *cft77* command can be in any order. If a keyword and option are omitted from the statement, the compiler uses a default value. If an entry in the command is not a recognized keyword, the job is aborted. If a keyword option is unrecognized, duplicated, or in conflict with another option, the job is usually aborted.

The command showing all default values is as follows (*file.f* must be specified, and other appearances of file use the same name. –s defaults to no file.):

cft77 **–a** *static* **–b** *file.o* **–d** *ADLSacfgjosx* **–e** *Bpqr* **–i46** **–l** **–m3** **-o** *full,nozeroinc* **–t0** -C *cray–xmp,nocigs,noema,novpop* — *file.f*

If conflicting list output options appear on a control statement, a warning message appears in the logfile and the option is used with the highest precedence (1 being the highest) as follows:

1. -l 0
2. -e L
3. -e h
4. -e cgsx
5. -d cgsx
6. CDIR$

Thus, if ON=S and the NOLIST directive is compiled, the directive is ignored.

–a *alloc* Memory allocation scheme for entities in memory. *alloc* can be either **static** (the default), or **stack.**

static All memory is statically allocated; a stack is not used, except automatic arrays and array temporaries, which are allocated on the heap.

stack Constants and entities in a DATA statement, SAVE statement, or a common block are statically allocated. All other entities are allocated on the stack, except automatic arrays and array temporaries, which are allocated on the heap.

–b *binfile* Creates file *binfile* (if it does not already exist), on which the compiler writes binary object modules. With **–b 0,** no binary load files are written. The default is *file.o.*

–d *eoplist* Disables or enables up to 12 compiler options. The options establish settings throughout an executable program. Compiler directives can turn many options on or off within programs, but only those options not included in the **–e** or **–d** string. If an option appears in one of the strings, directives for that option are ignored. **–d** *ADLSacfgjosx* **–e** *Bpqr*. The **–d** options are as follows.

L **–e***L* enables all available kinds of listings to the output file (*file.l* or the file specified by the -l parameter). These include generated code, cross reference listing, common blocks, and vectorization information. *L* supersedes *c, g, s,* and *x,* which supersedes compiler directives. The default is disabled (output is enabled by *c, g, s,* and *x* options).

D Generates a symbol table for the debugger on the file specified by **–b** binfile. The default file is *file.o*, where file is specified by *file.f* in the command. The default for the *D* option is disabled.

B Enables creation of a binary object file; that is, **–d***B disables the object file. See parameter* **–b**. The default is enabled.

A Generates messages to note all non ANSI usages. The default is disabled.

S Creates CAL file *file.s*, where file is specified by *file.f* in the command. Parameter –s creates a file with a non-default name, which overrides option *S*. The default is disabled.

a Aborts job after compilation if any program unit contains a fatal error. The default is disabled.

c Lists common block names and lengths in file *listfile* after each program unit. Not needed if **–e***L* is used. The default is disabled.

f **–e***f* generates flowtrace for the entire compilation unit. The option supersedes FLOW and NOFLOW directives. The default is disabled.

g **–e***g* enables listing of generated code to the output file (*file.l* or the file named by **–l**). This option is superseded by **–e***L*; it supersedes CODE and NOCODE directives. The default is disabled.

h Enables listing of first statement in each program unit and error messages. *h* is superseded by LIST; it supersedes *c, g, s,* and *x*. The default is disabled.

j Causes at least one execution of each DO loop whose DO statement is executed. The default is disabled.

o **–e***o* generates bounds checking for entire compilation unit, and also enables runtime conformance checking in array syntax expressions. **–e***o* supersedes the BOUNDS directive. The default is disabled.

p Allows double-precision. If **–d***p* is specified, the following occurs during compile two (the default is enabled):

- All double-precision declaratives are treated as real
- Double-precision functions are changed to the corresponding single-precision functions
- Double-precision constants are converted as double-precision and truncated to real
- D in FORMAT statement is changed to E

q Aborts compilation when 100 fatal error messages are counted. The default is enabled.

r Rounds the results on multiply operations. The default is enabled.

s **–e***s* enables listing of source code to the output file (*file.l* or file named by **–l**). This option supersedes LIST, NOLIST, and EJECT directives; it is not needed if **–e***L* is specified. The default is disabled

x Enables cross reference listing to the output file (file.o or the file named by -l). Not needed if **–e***L* is specified.

–i *intlen* *intlen* can be either 64 or 46, to specify 64-bit or 46-bit integer arithmetic. The default is 46.

–l *listfile* Creates file listfile to receive list output. Output is enabled by **–e** options /fIL, c, g, s, or *x*. The default, *file.l* uses file from *file.f* on the command.

–m *msglev* Level of CFT77 messages; *msglev* indicates the highest message level to be suppressed. For example, **–m2** allows Caution, Warning, and Fatal messages to appear. 0<msglev<4. For example, **–m0** allows all messages, and fatal errors are never suppressed. The default is 3 (only warning and error messages will be issued). The message levels are as follows:

Level	Severity
Comment	Comments on programming inefficiencies
Note	May cause problems with other compilers
Caution	Possible user error.
Warning	Probable user error.
Error	Fatal error (never suppressed)

–o *optim* Specifies optimization options. *optim* can be *off, full,* or *novector*; and *zeroinc* or *nozeroinc*. With full (default), compiler directives for vectorization are recognized. *nozeroinc*, the default, improves execution time by assuming that constant increment variables (CIVs) are not incremented by variables with the value 0. *zeroinc* adds runtime checks for zero increments of CIV increments in DO-loops.

–s *calfile* Creates file calfile (if it does not already exist) to receive Cray Assembly Language (CAL) output. This file can be manually modified to be input to the CAL assembler. DATA statements are not supported. The default is no file or option s

–t *trunc* Number of bits to be truncated. Range is 0<trunc<47. **–t** specifies truncation for all floating-point results; it does not truncate double-precision results, function results, or constants. Truncated bits are set to 0. The default is 0.

–C *cpu,hdw*

Specifies the mainframe type and optional characteristics of the hardware running the generated code. This parameters does not apply to CRAY-2 computer systems. The *cpu* value assumes the minimum characteristics for that mainframe. Unspecified hardware is assumed to be disabled. The *cpu* and *hdw* options are as follows (the default value is *cray–xmp, nocigs,noema,mova,movpop*):

Hardware	Description
cray-1m	CRAY-1M computer systems
cray-1	CRAY-1 computer systems
cray-1a cray-1b cray-1s	CRAY-1S computer systems
cray-xmp cray-x1 cray-x2 cray-x4	CRAY X-MP computer systems
ema˜noema	Target machine does/does not have extended memory addressing.
cigs˜nocigs	Target machine does/does not have gather-scatter hardware with compressed index hardware.
vpop˜novpop	Target machine does/does not have a vector population count.

file.f	Name of file containing source input. This option does not have an option. The file must be specified.

FILES

file.f	Fortran source file; must always be specified.
file.o	Binary object file name. The default name derived from *file.f*; parameter **-b** can be used to specify a different name.
file.l	Source listing file name. The default name derived from *file.f*; parameter **-e** can be used to specify a different name.
file.s	Pseudo CAL file; default is derived from *file.f*; parameter **-e** can be used to specify a different name.

SEE ALSO

UNICOS CFT77 Reference Card, publication SQ-0138
CFT77 Reference Manual, publication SR-0018, and UNICOS CFT77

NAME

chmod – Changes mode

SYNOPSIS

chmod *mode files*

DESCRIPTION

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

4000	Set user ID on execution
2000	Set group ID on execution
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*]

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

Op can be + to add *permission* to the file's mode, – to take away *permission*, or = to assign *permission* absolutely (the bits associated with the specified *who* 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). The **t** permission is associated with the 1000 bit; only a super user can alter the **t** permission. The **t** permission is not operable in UNICOS. Omitting *permission* is only useful with = to take away all permissions.

The letters **u, g,** and **o** are allowable as permissions. For example,

chmod g+u *file*

makes group permissions the same as user permissions.

Multiple symbolic modes separated by commas may be given. Operations are performed in the order specified. The letter **s** only works with **u** or **g**; **t** only works with **u** (**t** is not operable in UNICOS).

Only the owner of a file (or the super-user) may change its mode. 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, the second makes a file executable:

```
chmod o–w file
chmod +x file
```

SEE ALSO

ls(1)
chmod(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

chown, chgrp – Changes 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.

If either command is invoked by someone other than the super user, the set-user-ID and set-group-ID bits of the file mode, 04000 and 02000, respectively, are cleared.

FILES

/etc/passwd
/etc/group

SEE ALSO

chown(2) in the UNICOS System Calls Reference Manual, publication SR-2012
group(4F), passwd(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

chsh – Change default login shell

SYNOPSIS

chsh *name* [*shell*]

DESCRIPTION

The *chsh* command is similar to *passwd*(1) except that it changes the login shell field of your password file rather than the password entry. You can specify one of the following for *shell*: **/bin/csh**, **/bin/oldcsh**, or **/usr/new/csh**. If you do not specify *shell*, it defaults to the login shell */bin/sh*. Only the super user can specify a shell other than one of these.

Name is your login name.

An example use of this command would be

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

SEE ALSO

csh(1), passwd(1)
passwd(4F) the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

clear – Clears terminal screen

SYNOPSIS

clear

DESCRIPTION

The *clear* command clears your screen. It looks in the environment for the terminal type and then in **/usr/lib/terminfo** to figure out how to clear the screen.

FILES

/usr/lib/terminfo	Terminal capability data base

NAME

cmp – Compares two files

SYNOPSIS

cmp [**–l**] [**–s**] *file1 file2*

DESCRIPTION

File1 and *file2* 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, *cmp* displays the byte and line number at which the difference occurs. If one file is an initial subsequence of the other, that fact is noted.

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

Options for *cmp* are as follows:

–l Prints the byte number (decimal) and the differing bytes (octal) in three columns for each difference.

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

SEE ALSO

comm(1), diff(1)

NAME

comb – Combines SCCS deltas

SYNOPSIS

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

DESCRIPTION

Comb generates a shell procedure (see *sh*(1)) which, when run, reconstructs the given SCCS files. The reconstructed files are, hopefully, 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 input is taken to be the name of an SCCS file to be processed; non-SCCS files and unreadable files are silently ignored. If no keyletter arguments are specified, *comb* will preserve only leaf deltas and the minimal number of ancestors needed to preserve the tree.

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* Provides 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* Prints 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 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 file

MESSAGES

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.

SEE ALSO

admin(1), delta(1), get(1), help(1), prs(1)
sccsfile(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014
The Source Code Control System (SCCS) User Guide, publication SG-2017

NAME

comm – Selects or rejects 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 standard input is used.

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

SEE ALSO

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

NAME

cord – Invokes the core dump program

SYNOPSIS

cord [**–btvs**] [**–{w|a}** *addr1,addr2*]

DESCRIPTION

Cord is a core dump formatter. The program reads the local file **core** and formats selected portions for dumping to standard output. *Cord* dumps the exchange package by default but can dump **b**, **t**, or v registers, the stack (s), and memory by argument selection.

The options **–b**, **–t**, and **–v** select the respective saved registers for dumping. The **–s** option selects a stack dump. This is possible only if the stack is in tact. The memory dump can be invoked by either the **–w** or **–a** option. The dual option is to provide compatibility both with earlier versions of *cord* and *fdmp*(1). The parameter *addr1* is the starting dump word address, and the parameter *addr2* is the ending dump word address. The word dump mode is from program address zero, not core file address zero.

BUGS

Input can only be from the file **core**.

The stack dump often appears to be bashed.

Cord is extremely sensitive to the format of the **core** file.

There is currently no option for formatting the user structure.

NAME

cp – Copies files

SYNOPSIS

cp *file1* [*file2* ...] *target*

DESCRIPTION

File1 is copied 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 to that directory. If *target* is a file, its contents are destroyed.

If *target* is not a file, *cp* creates a new file that has the same mode as *file1* except that the sticky bit is not set unless you are super user (the sticky bit is not operable in UNICOS); 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, or group. The last modification time of *target* (and the 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.

SEE ALSO

cpio(1), ln(1), mv(1), rm(1)
chmod(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

cpio – Copies file archives in and out

SYNOPSIS

cpio –o [**acBv**]

cpio –i [**BcdmrtuvfsSb6**] [*patterns*]

cpio –p [**adlmuv**] *directory*

DESCRIPTION

The *cpio* **–o** command (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 4096-byte boundary.

The *cpio* **–i** command (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 * (that is, select all files). The extracted files are conditionally created and copied into the current directory tree based upon the options described below. The 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**.

The *cpio* **–p** command (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 *directory* based upon the options described below.

The available arguments are as follows. The hyphen preceding the following options can be omitted if the options are concatenated without spaces.

a	Reset *access* times of input files after they have been copied.
B	*Block* input/output at 5,120 bytes to the record (does not apply to the *pass* option; useful only with data that will be stored on tape).
d	Creates *directories* as needed.
c	Writes *header* information in ASCII character form for portability. This option is necessary to transfer a *cpio* archive between a CRAY mainframe and a front end machine.
r	Interactively *rename* files. If the user types a null line, the file is skipped. This option is usable only with the **–i** option.
t	Prints *table of contents* of the input. No files are created.
u	Copies *unconditionally* (normally, an older file will not replace a newer file with the same name).
v	*Verbose*: causes a list of file names to be printed. When used with the **t** option, the table of contents looks like the output of an **ls –l** command (see *ls*(1)).
l	Whenever possible, link files rather than copying them. Usable only with the **–p** option.
m	Retains previous file modification time. This is ineffective on directories that are being copied.
f	Copies in all files except those in *patterns*.
s	Swaps bytes. Use only with the **–i** option.
S	Swaps halfwords. Use only with the **–i** option.
b	Swaps both bytes and halfwords. Use only with the **–i** option.
6	Processes an old (that is, UNIX System Sixth Edition format) file. Only useful with the **–i** option (copy in).

EXAMPLES

The first example below copies the contents of a directory into an archive; the second duplicates a directory hierarchy:

```
ls | cpio –o >/archive/file

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

CAVEATS

Only the super user can copy special files.

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.

SEE ALSO

ar(1), find(1), ls(1), tar(1)
cpio(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

cpp – Invokes the C language preprocessor

SYNOPSIS

/lib/cpp [**–P**] [**–C**] [**–U***name*] [**–D** [*name=def*] [**–T**] [**–I***dir*] [*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. Thus, the output of *cpp* is in a form acceptable as input to the next pass of the C compiler. As the C language evolves, *cpp* and the rest of the C compilation package will be modified to follow these changes. Therefore, the use of *cpp* other than in this framework is not suggested. The preferred way to invoke *cpp* is through the *cc*(1) command since the functionality of *cpp* may someday be moved elsewhere.

Cpp optionally accepts two file names as arguments. *Ifile* and *ofile* are respectively the input and output for the preprocessor. They default to standard input and standard output if you do not supply these arguments.

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 Strips, by default, the C-style comments. If the **–C** option is specified, all comments (except those found on *cpp* directive lines) are passed along.

–U*name*
Removes any initial definition of *name*, where *name* is a reserved symbol that is predefined. The predefined symbols on the CRAY-2 Computer System are **CRAY**, **CRAY2**, and **unix**.

–D [*name=def*]
Defines *name* as if by a **#define** directive. If no *=def* is given, *name* is defined as 1.

–T Causes preprocessor symbols to not be restricted to 8 characters. The **–T** option forces *cpp* to use only the first 8 characters for distinguishing different preprocessor names. This behavior is the same as previous preprocessors with respect to the length of names and is included for backward compatibility.

–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 quotes (" ") will be searched for first in the directory of the *ifile* argument, then in directories named in **–I** options, and last in directories on a standard list. For **#include** files whose names are enclosed in <>, the directory of the *ifile* argument is not searched.

Two special names are understood by *cpp*. The name __LINE__ is defined as the current line number (as a decimal integer) as known by *cpp*, and __FILE__ is defined as the current file name (as a C string) as known by *cpp*. They can be used anywhere (including in macros) just as any other defined name.

All *cpp* directives start with lines begun by **#**. The directives are:

#define *name token-string*
Replace subsequent instances of *name* with *token-string*.

#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 tokens, and a) by *token-string* where each occurrence of an *arg* in the *token-string* is replaced by the corresponding token in the comma-separated list.

#undef *name*

Cause the definition of *name* (if any) to be forgotten.

#include "*filename*"
#include <*filename*>

Include at this point the contents of *filename* (which will then be run through *cpp*). When the <*filename*> notation is used, *filename* is only searched for in the standard places. See the **-I** option above for more detail.

#line *integer-constant "filename"*

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

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

#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. So if lines before 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.

NOTES

In previous versions of *cpp*, when new-line characters were found in argument lists for macros to be expanded, new-lines were put out 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.

FILES

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

MESSAGES

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.

SEE ALSO

cc(1)

NAME

crontab – Copies files into the 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 lists the crontab file for the invoking user.

A user is permitted to use *crontab* if his or her 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. The null file **cron.allow** would mean no user is allowed to use **cron**; a null file **cron.deny** would mean no user is denied the use of **cron**. 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 one 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 % character 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).** You will probably want to set the **TZ** variable.

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 is mailed to the user.

FILES

/usr/lib/cron	Main **cron** directory
/usr/spool/cron/crontabs	Spool area
/usr/lib/cron/log	Accounting information
/usr/lib/cron/cron.allow	List of allowed users
/usr/lib/cron/cron.deny	List of denied users

SEE ALSO

sh(1)
cron(1M) in the UNICOS Administrator Commands Reference Manual, publication SR-2022

NAME

crypt – Encodes/decodes files

SYNOPSIS

crypt [*password*]

DESCRIPTION

Crypt reads from the standard input and writes on the standard output. The *password* is a key that selects a particular transformation. If no *password* is given, *crypt* demands a key from the terminal and turns off printing while the key is being typed in. *Crypt* encrypts and decrypts with the same key:

crypt *key* <*clear* >cypher
crypt *key* <cypher | pr

will print the *clear*.

Files encrypted by *crypt* are compatible with those treated by the editor *ed* in encryption mode.

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	Typed key

LIMITATIONS

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.

SEE ALSO

ed(1), makekey(1)

NAME

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

SYNOPSIS

csh [**–cefinstvVxX**] [*arg* ...]

DESCRIPTION

The *csh* command is the first implementation of a command language interpreter that incorporates a history mechanism (see the History Substitutions subsection), job control facilities, and a C-like syntax.

A session with *csh* begins by executing commands from the file **.cshrc** in your home directory. If this is a login shell, *csh* also executes commands from the files **/etc/cshrc**, **.cshrc**, and **.login** (in that order).

The shell begins reading commands from the terminal after the '% ' prompt; it then repeatedly performs the following actions:

1. Reads a line of command input
2. Breaks the line of command input into *words* (described under Lexical structure)
3. Puts the sequence of words in the command history list (described under History substitution)
4. Parses the command history list
5. Executes each command on the current line

When a login shell terminates, *csh* executes commands from the **.logout** file in the users home directory.

If argument 0 to the shell is '–', then this is a login shell. The *csh* command accepts the following options:

–c Reads commands from the (single) following argument, which must be present. Any remaining arguments are placed in the *argv* variable.

–e Exits if any invoked command terminates abnormally or yields a nonzero exit status

–f Starts faster, because it neither searches for nor executes commands from the **.cshrc** file in your home directory.

–i Specifies an interactive shell and prompts for its top-level input, even if it appears to not be a terminal. Shells are interactive without this option if their inputs and outputs are terminals.

–n Commands are parsed, but not executed. This helps check for accuracy in syntax of shell scripts.

–s Takes command input from the standard input

–t Reads and executes a single line of input Use a '\' to escape the newline at the end of this line and continue onto another line.

–v Sets the **verbose** variable, so that command input is echoed after history substitution

–x Sets the **echo** variable, so that commands are echoed immediately before execution

–V Sets the **verbose** variable even before **.cshrc** is executed

–X Is to **–x** as **–V** is to **–v**.

After argument processing, if arguments remain but you did not specify any of the **–c, –i, –s,** or **–t** options, the first argument is taken as the name of a file of commands to be executed. The shell opens this file, and saves its name for possible resubstitution by '$0'. Since many systems use either the standard version 6 or version 7 shells whose shell scripts are not compatible with this shell, the shell executes such a 'standard' shell if the first character of a script is not a '#', that is, if the script does not start with a comment. Remaining arguments initialize the variable *argv.*

Lexical structure

The shell splits input lines into words at blanks and tabs with the following exceptions. The characters '&' '|' ';' '<' '>' '(' ')' form separate words. If the characters '&&', '| |', '<<' or '>>' are doubled, the pairs form single words. You can use these parser metacharacters as part of other words or override their special meaning by preceding them with '\'. (A newline preceded by a '\' is equivalent to a blank.)

In addition, strings enclosed in matched pairs of quotations, ''', '`', or '"', form parts of a word; metacharacters in these strings, including blanks and tabs, do not form separate words. Quotations have semantics to be described subsequently (see the Quotations with ' and " subsection). Within pairs of ''' or '"' characters, a newline preceded by a '\' gives a true newline character.

When the shell's input is not a terminal, the character '#' introduces a comment, which continues to the end of the input line. To override this special meaning, precede the '#' with a '\' and use '`', ''', and '"' in quotations.

Commands

A simple command is a sequence of words, the first of which specifies the command you want to execute. A simple command or a sequence of simple commands separated by '|' characters forms a pipeline. The output of each command in a pipeline is connected to the input of the next. You can separate sequences of pipelines with ';'. The piped commands are then executed sequentially. You can execute a sequence of pipelines without immediately waiting for the sequence to terminate by following it with an '&'.

You can put any of the above characters in '(' ')' to form a simple command (which can be a component of a pipeline). You can also separate pipelines with '| |' or '&&' indicating, as in the C language, that the second is to be executed only if the first fails or succeeds respectively. (See the Expressions subsection.)

Built-in commands

The *csh* command accepts the following list of **built-in** commands (execution of **nonbuilt-in** commands is described later). If a built-in command occurs as any component of a pipeline except the last then it is executed in a subshell.

alias
alias *name*
alias *name wordlist*

The first form prints all aliases. The second form prints the alias for *name*. The final form assigns the specified *wordlist* as the alias of *name; wordlist* is command and filename substituted. *Name* can not be *alias* or *unalias.*

break

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

breaksw

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

case label:

A label in a **switch** statement as discussed below (see the **default** command).

cd
cd *name*
chdir
chdir *name*
Changes the shell's working directory to directory *name*. If no argument is given, it changes to the home directory of the user. If *name* is not found as a subdirectory of the current directory (and does not begin with '/', './' or '../'), then each component of the variable **cdpath** is checked to see if it has a subdirectory *name*. Finally, if all else fails but *name* is a shell variable whose value begins with '/', then this is tried to see if it is a directory.

continue
Continues execution of the nearest enclosing **while** or **foreach.** The rest of the commands on the current line are executed.

default:
Labels the default case in a **switch** statement. The default should come after all **case** labels.

dirs
Prints the directory stack; the top of the stack is at the left, the first directory in the stack is the current directory.

echo *wordlist*
echo –n *wordlist*
Writes the specified words to the shell's standard output, separated by spaces, and terminated with a newline unless the **–n** option is specified.

else
end
endif
endsw
See the description of the **foreach, if, switch,** and **while** statements below.

eval *arg ...*
(As in *sh*(1).) The arguments are read as input to the shell and the resulting command(s) are executed in the context of the current shell. This is usually used to execute commands generated as the result of command or variable substitution, since parsing occurs before these substitutions. See *tset*(1) for an example of using **eval.**

exec *command*
Executes the specified command in place of the current shell.

exit
exit *(expr)*
Exits the shell either with the value of the **status** variable (first form) or with the value of the specified *expr* (second form).

foreach *name (wordlist)*
...
end
Successively sets the variable *name* to each member of *wordlist* and executes the sequence of commands between this command and the matching **end.** (Both **foreach** and **end** must appear alone on separate lines.)

continue may be used to continue the loop prematurely a **break** to terminate it prematurely. When this command is read from the terminal, the loop is read up once prompting with '?' before any statements in the loop are executed. If you make a mistake typing in a loop at the terminal you can rub it out.

glob *wordlist*
: Like **echo** but no '\' escapes are recognized and words are delimited by null characters in the output. **Glob** is useful for programs that wish to use the shell to filename expand a list of words.

goto *word*
: The specified *word* is a 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.

history
history *n*
history –r *n*
history –h *n*
: Displays the history event list. If *n* is given, only the *n* most recent events are printed. The **–r** option reverses the order of printout to be most recent first instead of the oldest first. The **–h** option causes the history list to be printed without leading numbers. This is used to produce files suitable to the **source** command using the **–h** option to **source**.

if *(expr) command*
: If the specified expression evaluates true, then the single *command* with arguments is executed. Variable substitution (described later) on *command* happens simultaneously with the rest of the **if** command. *Command* must be a simple command (not a pipeline), a command list, or a parenthesized command list. Input/output redirection occurs even if *expr* is false, when *command* is **not** executed (this is a bug).

if *(expr)* **then**
...
else if (expr2) **then**
...
else
...
endif
: If the specified *expr* is true, then the commands up to the first **else** are executed; else if *expr2* is true then the commands up to the second else are executed, and so on. Any number of **else-if** pairs are possible; only one **endif** is needed. The **else** part is likewise optional. (The words else and **endif** must appear at the beginning of input lines; the **if** must appear alone on its input line or after an **else**.)

jobs
jobs –l
: Lists the active jobs. The **–l** option lists the process ID's in addition to the normal information.

kill *%job*
kill *–sig % job ...*
kill *pid*
kill *–sig pid ...*
kill –l
: Sends either the TERM (terminate) signal or the specified signal to the specified jobs or processes. Signals are either given by number or by names (as given in **/usr/include/signal.h,** stripped of the prefix "SIG"). The signal names are listed by "**kill –l**". There is no default, saying just '**kill**' does not send a signal to the current job. If the signal being sent is TERM (terminate) or HUP (hangup), then the job or process will be sent a CONT (continue) signal as well.

login
Terminates a login shell, replacing it with an instance of **/bin/login.** This is one way to log off; it was included for compatibility with *sh*(1).

logout
Terminate a login shell. Especially useful if the **ignoreeof** variable is set.

newgrp

newgrp –

newgrp *group*

newgrp – *group*
Changes a user's group identifation by replacing the current shell with an instance of **/bin/newgrp.** The first form changes the group ID back to the group you specified in your password file entry. The second form changes the environment to what would be expected if the user logged in again. The third form changes the group ID to *group.* The last form combines the actions of the second and third forms. You may be prompted for a password.

nice
nice *+number*
nice *command*
nice *+number command*
The first form adds 4 to the current **nice** value for this shell. for this shell to 4. The second form adds *number* to the current **nice** value. The final two forms run *command* at priority 4 plus the current **nice** value, and *number* plus the current **nice** value, respectively. The super user may specify negative niceness by using 'nice –number ...'. The *command* is always executed in a sub-shell, and the restrictions placed on commands in simple **if** statements apply. The system imposes a maximum **nice** value of 39 and a minimum **nice** value of 0.

nohup
nohup *command*
The first form can be used in shell scripts to ignore hangups for the remainder of the script. The second form causes the specified *command* to run with hangups ignored. All processes detached with '&' are effectively ignored.

notify
notify *%job ...*
Causes the shell to notify the user asynchronously when the status of the current or specified job changes; normally notification is presented before a prompt. This is automatic if the shell variable **notify** is set.

onintr
onintr –
onintr *label*
Controls the action of the shell on interrupts. The first form restores the default action of the shell on interrupts, which is to terminate shell scripts or to return to the terminal command input level. The second form, '**onintr –**', causes all interrupts to be ignored. The final form causes the shell to execute a 'goto label' when it receives an interrupt or when a child process terminates because it was interrupted.

If the shell is running detached and interrupts are being ignored, all forms of **onintr** have no meaning, and interrupts continue to be ignored by the shell and all invoked commands.

popd
popd *+n*
Pops the directory stack, returning to the new top directory. With an argument, '*+n*', **popd** discards the *n* th entry in the stack. The elements of the directory stack are numbered from 0 starting at the top.

pushd
pushd *name*
pushd *+n*
The first form, without arguments, exchanges the top two elements of the directory stack. The second form changes to the new directory (ala **cd**) and pushes the old current working directory (as in **csw**) onto the directory stack. The last form with a numeric argument, **pushd** 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
Recomputes the internal hash table of the contents of the directories in the **path** variable. 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*
Executes the specified *command*, which is subject to the same restrictions as the *command* in the one line **if** statement above, *count* times. I/O redirections occur exactly once, even if *count*=0.

set
set *name*
set *name=word*
set *name[index]=word*
set *name=(wordlist)*
The first form of the command shows the value of all shell variables. Variables that 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.

You can repeat these arguments to set multiple values in a single set command. Note that variable expansion happens for all arguments before any setting occurs.

setenv *name value*
Sets the value of environment variable *name* to be *value*, a single string. The most commonly used environment variable LOGNAME, TERM, and PATH are automatically imported to and exported from the *csh* variables *user, term,* and *path;* there is no need to use *setenv* for these.

shift
shift *variable*
Shifts the members of *argv* to the left, discarding *argv*[1]. It is erroneous to not have *argv* set or to have less than one word as value. The second form performs the same function on the specified *variable*.

source *name*
source -h *name*
Reads commands from *name*. **Source** commands can 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 not placed on the history list; the option causes the commands to be placed in the history list without being executed.

switch *(string)*
case *str1:*
  ...
 breaksw
...
default:
  ...
 breaksw
endsw

Matches each case label successively, 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 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 no **default** exists, execution continues after **endsw.**

time
time *command*

This command is only available on CRAY X-MP and CRAY-1 computer systems. With no argument, the shell prints a summary of time used by this shell and its children. If the argument is given, the shell times the specified simple command and prints a time summary as described under the **time** variable. If necessary, an extra shell is created to print the time statistic when the command completes.

umask
umask *value*

Displays (first form) or sets the file creation mask to the specified value (second form). The mask is given in octal. Common values for the mask are 002 giving all access to the group and read and execute access to others or 022 giving all access except no write access for users in the group or others.

unalias *pattern*

Discards all aliases whose names match the specified pattern. Thus all aliases are removed by 'unalias *'. It is not an error for nothing to be *unaliased.*

unhash

Disables use of the internal hash table. The internal hash table speeds up locating executed programs.

unset *pattern*

Removes all variables whose names match the specified *pattern.* To remove all variables, use '**unset ***'; this has noticeably distasteful side-effects. It is not an error for nothing to be **unset.**

unsetenv *pattern*

Removes all variables whose name match the specified *pattern* from the environment. See also the **setenv** command above and *printenv*(1).

wait

Waits for all background jobs. If the shell is interactive, an interrupt can disrupt the wait.

which *name ...*

Takes the specified *name* and searches for the file that would be executed had this *name* been given as a command. *Name* is expanded if it is aliased, and searched for along your path.

while *(expr)*

...

end

While the specified expression evaluates nonzero, the commands between the **while** and the matching **end** are evaluated. The **break** and **continue** built-in commands may be used to terminate or continue the loop prematurely. (The **while** and **end** must appear alone on their input lines.) Prompting occurs here the first time through the loop as for the **foreach** statement if the input is a terminal.

@

@ *name=expr*

@ *name[index]=expr*

The first form prints the values of all the shell variables. The second form sets the specified *name* th the value of *expr*. If the expression contains '<', '>', '&', or '|', then at least this part of the expression must be put in parentheses (()). The third form assigns the value of *expr* to the *index'th* argument of *name*. Both *name* and its *index'th* component must already exist.

The operators '*=', '+=', etc. are available as in C. The space separating the name from the assignment operator is optional. Spaces are, however, mandatory in separating components of *expr* which would otherwise be single words.

Special postfix '++' and '--' operators increment and decrement *name* respectively (that is, '@ i++').

Nonbuilt-in command execution

When a command to be executed is found not to be a built-in command, the shell attempts to execute the command using *execv*(2). Each word in the variable **path** names a directory from which the shell will attempt to execute the command. If it is not given a **–c** or a **–t** option, the shell hashes the names in these directories into an internal table so that it only tries an *exec* in a directory if there is a possibility that the command resides there. This greatly speeds command location when a large number of directories are present in the search path. If this mechanism has been turned off (using **unhash**), or if the shell was given a **–c** or **–t** argument (and in any case, for each directory component of **path** that does not begin with a '/'), the shell concatenates with the given command name to form a path name of a file, which it then attempts to execute.

Parenthesized commands are always executed in a subshell. For example,

```
(cd ; pwd) ; pwd
```

prints the **home** directory; leaving you where you were (printing this after the home directory), while

```
cd ; pwd
```

leaves you in the **home** directory. Parenthesized commands are most often used to prevent **chdir** from affecting the current shell.

If the file has execute permissions but is not an executable binary to the system, it is assumed to be a file containing shell commands, and a new shell is spawned to read it.

If there is an *alias* for *shell*, the words of the alias are prepended to the argument list to form the shell command. The first word of the *alias* should be the full path name of the shell (for example, $shell). This is a special, late occurring, case of *alias* substitution, and only allows words to be prepended to the argument list without modification.

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.

The shell redirects standard input to **/dev/null** for a job being run in the background, this will result in the receipt of an end-of-file if such a job tries to read from the terminal. Background jobs are normally allowed to produce output.

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'. Jobs can also be named by prefixes of the string typed in to start them, if these prefixes are unambiguous. 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. The characters '%%' is a synonym for the current job.

Status reporting

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

Substitutions

The following subsections describe the various transformations the shell performs on the input. The shell performs these in the following order:

History substitution
Alias substitution
Variable substitution
Command substitution
Filename substitution

History substitution

History substitutions place words from previous command input as portions of new commands, making it easy to repeat commands, repeat arguments of a previous command in the current command, or fix spelling mistakes in the previous command with little typing and a high degree of confidence. History substitutions begin with the character '!' and may begin **anywhere** in the input stream (with the provison that they **do not** nest.) The '!' can be preceded by a '\' to override its special meaning; for convenience, a '!' is passed unchanged when it is followed by a blank, tab, newline, '=' or '('. (History substitutions also occur when an input line begins with a carat (^); which will be described later.) Any input line that contains history substitution is echoed on the terminal before it is executed as it could have been typed without history substitution.

Commands input from the terminal consisting of one or more words are saved in the history list. The history substitutions reintroduce sequences of words from these saved commands into the input stream (the size of which is controlled by the **history** variable). The previous command is always retained, regardless of its value. Commands are numbered sequentially from 1.

For definiteness, consider the following output from the **history** command:

```
 9  write michael
10  ex write.c
11  cat oldwrite.c
12  diff *write.c
```

The commands are shown with their event numbers. It is not usually necessary to use event numbers, but the current event number can be made part of the *prompt* by placing an '!' in the prompt string.

With the current event 13, we can refer to previous events by event number '!11', relatively as in '!–2' (referring to the same event), by a prefix of a command word as in '!d' for event 12 or '!wri' for event 9, or by a string contained in a word in the command as in '!?mic?', which also refers to event 9. These forms, without further modification, simply reintroduce the words of the specified events, each separated by a single blank. As a special case '!!' refers to the previous command; thus '!!' alone is essentially a *redo*.

To select words from an event, we follow the event specification by a ':' and a designator for the desired words. The words of an input line are numbered from 0, the first (usually command) word being 0, the second word (first argument) being 1, etc. The basic word designators are:

0	first (command) word
n	*n*'th argument
^	first argument, that is, '1'
$	last argument
%	word matched by (immediately preceding) ?*s*? search
x–y	range of words
–y	abbreviates '0–*y*'
*	abbreviates '^–$', or nothing if only 1 word in event
*x**	abbreviates '*x*–$'
x–	like '*x**' but omitting word '$'

The ':' separating the event specification from the word designator can be omitted if the argument selector begins with a '^', '$', '*' '–' or '%'. After the optional word designator you can place a sequence of modifiers, each preceded by a ':'. The following modifiers are defined:

h	Remove a trailing pathname component, leaving the head.
r	Remove a trailing '.xxx' component, leaving the root name.
e	Remove all but the extension '.xxx' part.
s/*l*/*r*/	Substitute *r* for *l*
t	Remove all leading pathname components, leaving the tail.
&	Repeat the previous substitution.
g	Apply the change globally, prefixing the above, for example, 'g&'.
p	Print the new command but do not execute it.
q	Quote the substituted words, preventing further substitutions.
x	Like q, but break into words at blanks, tabs and newlines.

Unless preceded by a 'g' the modification is applied only to the first modifiable word. With substitutions, it is an error for no word to be applicable.

The left hand side of substitutions are not regular expressions in the sense of the editors, but are strings. Any character may be used as the delimiter in place of '/'; a '\' quotes the delimiter into the *l* and *r* strings. The character '&' in the right hand side is replaced by the text from the left. A '\' quotes '&' also. A null *l* uses the previous string either from a *l* or from a contextual scan string *s* in '!?*s*?'. The trailing delimiter in the substitution may be omitted if a newline follows immediately as may the trailing '?' in a contextual scan.

A history reference may be given without an event specification, for example, '!$'. In this case the reference is to the previous command unless a previous history reference occurred on the same line in which case this form repeats the previous reference. For example,'!?foo?^ !$' 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 carat (^). This is equivalent to '!:s^' providing a convenient shorthand for substitutions on the text of the previous line. For example, '^lb^lib' fixes the spelling of 'lib' in the previous command. Finally, a history substitution may be surrounded with '{' and '}' if necessary to insulate it from the characters which follow. For example, after 'ls –ld ~paul' we might do '!{l}a' to do 'ls –ld ~paula', while '!la' would look for a command starting 'la'.

Quotations with ´ and "

You can put quotations around strings ('´' and '"') to override all or some of the remaining substitutions. Strings enclosed in '´' are prevented any further interpretation. Strings enclosed in '"' may be expanded as described below.

In both cases the resulting text becomes (all or part of) a single word; only in one special case (see the Command Substitition subsection) does a '"' quoted string yield parts of more than one word; '´' quoted strings never do.

Alias substitution

The shell maintains a list of aliases which can be established, displayed and modified by the **alias** and **unalias** commands. After a command line is scanned, it is parsed into distinct commands and the first word of each command, left-to-right, is checked to see if it has an alias. If it does, then the text which is the alias for that command is reread with the history mechanism available as though that command were the previous input line. The resulting words replace the command and argument list. If no reference is made to the history list, then the argument list is left unchanged.

For example, if the alias for 'ls' is 'ls –l' the command 'ls /usr' would map to 'ls –l /usr', the argument list here being undisturbed. Similarly if the alias for 'lookup' was 'grep !^ /etc/passwd' then 'lookup bill' would map to 'grep bill /etc/passwd'.

If an alias is found, the word transformation of the input text is performed and the aliasing process repeats on the reformed input line. Looping is prevented if the first word of the new text is the same as the old by flagging it to prevent further aliasing. Other loops are detected and cause an error.

The mechanism allows aliases to introduce parser metasyntax. Thus we can 'alias print ´pr \!* | lpr´' to make a command which *pr's* its arguments to the line printer.

Variable substitution

The shell maintains a set of variables, each of which has as value a list of zero or more words. Some of these variables are set by the shell or referred to by it. For instance, the *argv* variable is an image of the shell's argument list, and words of this variable's value are referred to in special ways.

The values of variables may be displayed and changed by using the **set** and **unset** commands. Of the variables referred to by the shell a number are toggles; the shell disregards what their value is, and is only concerned if they are set or not. For instance, the **verbose** variable is a toggle which causes command input to be echoed. The setting of this variable results from the **–v** command line option.

Other operations treat variables numerically. The @ command permits numeric calculations to be performed and the result assigned to a variable. Variable values are, however, always represented as (zero or more) strings. For the purposes of numeric operations, the null string is considered to be zero, and the second and subsequent words of multiword values are ignored.

After the input line is aliased and parsed, and before each command is executed, variable substitution is performed keyed by '$' character. This expansion can be prevented by preceding the '$' with a '\' except within '"'s where it **always** occurs, and within '´'s where it **never** occurs. Strings quoted by '`' are interpreted later (see *Command substitution* below); '$' substitution does not occur there until later, if at all. A '$' is passed unchanged if followed by a blank, tab, or end-of-line.

Input/output redirections are recognized before variable expansion, and are variable expanded separately. Otherwise, the command name and entire argument list are expanded together. It is possible for the first (command) word to this point to generate more than one word, the first of which becomes the command name, and the rest of which become arguments.

Unless enclosed in '"' or given the ':q' modifier, the results of variable substitution may eventually be command and filename substituted. A variable within '"', whose value consists of multiple words expands to a (portion of) a single word, with the words of the variables value separated by blanks. When the ':q' modifier is applied to a substitution, the variable expands to multiple words, with each word separated by a blank and quoted to prevent later command or filename substitution.

The following metasequences introduce variable values into the shell input. Except as noted, it is an error to reference a variable which is not set.

$name
${name}

Are replaced by the words of the value of variable **name,** each separated by a blank. Braces insulate **name** from following characters which would otherwise be part of it. Shell variables have names consisting of up to 20 letters and digits starting with a letter. The underscore character is considered a letter.
If **name** is not a shell variable, but is set in the environment, then that value is returned (but : modifiers and the other forms given below are not available in this case).

$name[selector]
${name[selector]}

May be used to select only some of the words from the value of *name*. The selector is subjected to '$' substitution and may consist of a single number or two numbers separated by a '–'. The first word of a variables value is numbered '1'. If the first number of a range is omitted, it defaults to '1'. If the last member of a range is omitted, it defaults to '$#name'. The selector '*' selects all words. It is legal for a range to be empty if the second argument is omitted or in range.

$#name
${#name}

Gives the number of words in the variable. This is useful for later use in a '[selector]'.

$0

Substitutes the name of the file from which command input is being read. An error occurs if the name is unknown.

$number
${number}

Equivalent to '$argv[number]'.

$*

Equivalent to '$argv[*]'.

The modifiers ':h', ':t', ':r', ':q' and ':x' may be applied to the substitutions above as may ':gh', ':gt' and ':gr'. If braces '{' '}' appear in the command form, then the modifiers must appear within the braces. **The current implementation allows only one ':' modifier on each '$' expansion.**

The following substitutions may not be modified with ':' modifiers.

$?name
${?name}

Substitutes the string '1' if name is set, '0' if it is not.

$?0

Substitutes '1' if the current input filename is known, '0' if it is not.

$$

Substitute the (decimal) process number of the (parent) shell.

$<

Substitutes a line from the standard input, with no further interpretation thereafter. It can be used to read from the keyboard in a shell script.

Command and filename substitution

Command and filename substitution are applied selectively to the arguments of built-in commands. This means that portions of expressions which are not evaluated are not subjected to these expansions. For commands which are not internal to the shell, the command name is substituted separately from the argument list. This occurs very late, after input-output redirection is performed, and in a child of the main shell.

Command substitution

Command substitution is indicated by a command enclosed in '`'. The output is broken into separate words at blanks, tabs and newlines, with null words being discarded, this text then replaces the original string. Within '"'s, 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 this makes it 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, the characters '~' and '{' are more akin to abbreviations.

In matching filenames, the character '.' at the beginning of a filename or immediately following a '/', as well as the character '/' must be matched explicitly. The character '*' matches any string of characters, including the null string. The character '?' matches any single character. The sequence '[...]' matches any one of the characters enclosed. Within '[...]', a pair of characters separated by '–' matches any character lexically between the two.

The character '~' at the beginning of a filename is used to refer to home directories. Standing alone, i.e. '~' it expands to the user's 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; for example, '~ken' might expand to '/usr/ken' and '~ken/chmach' to '/usr/ken/chmach'. If the character '~' is followed by a character other than a letter or '/' or does not appear 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. For example, '~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 it is created; if the file exists, it is truncated, losing its previous contents.

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 '!' forms can be used and suppress this check.

The forms involving '&' route the diagnostic output into the specified file as well as the standard output. The *name* is expanded in the same way as '<' input filenames are.

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

Uses file *name* as standard output like '>' but places output at the end of the file. If the variable **noclobber** is set, then it is an error for the file not to exist unless one of the '!' forms is given. Otherwise similar to '>'.

A command receives the environment in which the shell was invoked as modified by the input-output parameters and the presence of the command in a pipeline. Unlike some previous shells, commands run from a file of shell commands have no access to the text of the commands by default; rather they receive the original standard input of the shell. The '<<' mechanism should be used to present inline data. This permits shell command scripts to function as components of pipelines and allows the shell to block read its input. Note that the default standard input for a command run detached remains as the original standard input of the shell. If this is a terminal and if the process attempts to read from the terminal, then the process will block and the user will be notified

Diagnostic output may be directed through a pipe with the standard output. By using the form '|&' rather than just '|'.

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:

```
|| && | ^ & == != =~ !~ <= >= < > << >> + - * / % ! ~ ( )
```

Here the precedence increases to the right, '==' '!=' '=~' and '!~', '<=' '>=' '<' and '>', '<<' and '>>', '+' and '-', '*' '/' and '%' being, in groups, at the same level. The '==' '!=' '=~' and '!~' operators compare their arguments as strings; all others operate on numbers. The operators '=~' and '!~' are like '!=' and '==' except that the right hand side is a *pattern* (containing, for example, '*'s, '?'s and instances of '[...]') against which the left hand operand is matched. This reduces the need for use of the *switch* statement in shell scripts when all that is really needed is pattern matching.

Strings that 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. Note that no two components of an expression can appear in the same word; except when adjacent to components of expressions which are syntactically significant to the parser ('&' '|' '<' '>' '(' ')') they are surrounded by spaces.

Also available in expressions as primitive operands are command executions enclosed in '{' and '}' and file enquiries of the form '*–l* name', where *l* is one of the following:

r	read access
w	write access
x	execute access
e	existence
o	ownership
z	zero size
f	plain file
d	directory

The specified name is command and filename expanded and then tested to see if it has the specified relationship to the real user. If the file does not exist or is inaccessible then all enquiries return false, that is, '0'. Successful command executions return the value '1' if true or '0' if false. If more detailed status information is required then the command should be executed outside of an expression and the variable **status** examined.

Control flow

The shell contains a number of commands that can be used to regulate the flow of control in command files (shell scripts) and (in limited but useful ways) from terminal input. These commands all operate by forcing the shell to reread or skip in its input and, due to the implementation, restrict the placement of some of the commands.

The **foreach, switch,** and **while** statements, as well as the **if–then–else** form of the **if** statement require that the major keywords appear in a single simple command on an input line as shown below.

If the shell's input is searchable, the shell buffers up input whenever a loop is being read and performs seeks in this internal buffer to accomplish the rereading implied by the loop. (To the extent that this allows, backward goto's will succeed on non-seekable inputs.)

Predefined and environment variables

The following variables have special meaning to the shell. Of these, the shell always sets **argv, cwd, home, path, prompt, shell** and **status.** Except for **cwd** and **status,** these variables are only set at initialization; therefore, if you wish to modify these, you must explicitly do so.

The shell copies the environment variable **LOGNAME** into the variable **user, TERM** into **term,** and **HOME** into **home,** which are copied back into the environment whenever the normal shell variables are reset. The environment variable PATH is likewise handled; you do not need to worry about its setting other than in the **.cshrc** file as inferior *csh* processes will import the definition of **path** from the environment, and re-export it if you change it.

argv Sets the arguments to the shell, it is from this variable that positional parameters are substituted, for example, '$1' is replaced by '$argv[1]'.

cdpath Gives a list of alternate directories searched to find subdirectories in *chdir* commands.

cwd Gives the full pathname of the current directory.

echo Sets when the **–x** command line option is given. Causes each command and its arguments to be echoed just before execution. For nonbuilt-in commands all expansions occur before echoing. Built-in commands are echoed before command and filename substitution, since these substitutions are then done selectively.

histchars Can be given a string value to change the characters used in history substitution. The first character of its value is used as the history substitution character, replacing the default character !. The second character of its value replaces the character ^ in quick substitutions.

history Can be given a numeric value to control the size of the history list. Any command that has been referenced in this many events will not be discarded. Too large values of **history** may run the shell out of memory. The last executed command is saved on the history list.

home The user's home directory, initialized from the environment. The filename expansion of '~' refers to this variable.

ignoreeof If set, the shell ignores end-of-file from input devices, which are terminals. This prevents shells from accidentally being killed by CONTROL-D's.

mail The files where the shell checks for mail. This is done after each command completion, which results in a prompt, if a specified interval has elapsed. The shell says 'You have new mail.' if the file exists with an access time not greater than its modify time.

If the first word of the value of **mail** is numeric it specifies a different mail checking interval, in seconds, instead of the default, which is 10 minutes.

If multiple mail files are specified, the shell says 'New mail in *name*' when there is mail in the file *name*.

noclobber As described in the section on Input/output, restrictions are placed on output redirection to ensure that files are not accidentally destroyed, and that '>>' redirections refer to existing files.

noglob If set, filename expansion is inhibited. This is most useful in shell scripts that are not dealing with filenames, or after a list of filenames has been obtained and further expansions are not desirable.

nonomatch If set, it is not an error for a filename expansion to not match any existing files; rather the primitive pattern is returned. It is still an error, however, for the primitive pattern to be malformed, that is, 'echo [' still gives an error.

notify If set, the shell notifies asynchronously of job completions. The default is to rather present job completions just before printing a prompt.

path Each word of the path variable specifies a directory in which commands are to be sought for execution. A null word specifies the current directory. If there is no **path** variable, only full path names will execute. The usual search path is '.', '/bin' and '/usr/bin', but this may vary from system to system. For the super-user the default search path is '/etc', '/bin' and '/usr/bin'. A shell that is given neither the **–c** nor the **–t** option will normally hash the contents of the directories in the **path** variable after reading **.cshrc,** and each time the **path** variable is reset. If new commands are added to these directories while the shell is active, it may be necessary to give the **rehash** or the commands may not be found.

prompt The string that is printed before each command is read from an interactive terminal input. If a '!' appears in the string, it will be replaced by the current event number unless a preceding '\' is given. Default is '% ', or '# ' for the super user.

savehist Given a numeric value to control the number of entries of the history list that are saved in ~/.history when the user logs out. Any command that has been referenced in this many events will be saved. During start up, the shell sources ~/.history into the history list enabling history to be saved across logins. Too large values of **savehist** will slow down the shell during start up.

shell The file in which the shell resides. This is used in forking shells to interpret files that have execute bits set, but that are not executable by the system. (See the description

	of Nonbuiltin Command Execution.) Initialized to the (system-dependent) home of the shell.
status	The status returned by the last command. If it terminated abnormally, then 0200 is added to the status. Built-in commands that fail return exit status '1', all other built-in commands set status '0'.
time	Controls automatic timing of commands. If set, any command that takes more than this many cpu seconds prints a line giving user, system, and real times and a utilization percentage, which is the ratio of user plus system times to real time to be printed when it terminates.
verbose	Set by the **–v** command line option, which prints the words of each command after history substitution.

Signal handling

The shell ignores **quit** signals. Jobs running detached (by '&') are immune to signals generated from the keyboard, including hangups. Other signals have the values, which the shell inherited from its parent. **Onintr** can control the shells handling of interrupts and terminate signals in shell scripts. 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.**

FILES

~/.cshrc	Read at beginning of execution by each shell
~/.login	Read by login shell, after '.cshrc' at login
~/.logout	Read by login shell, at logout
/bin/sh	Standard shell, for shell scripts not starting with a '#'
/tmp/sh*	Temporary file for '<<'
/etc/passwd	Source of home directories for '~name'

LIMITATIONS

Words can be no longer than 1024 characters. The system limits argument lists to 5120 characters on the CRAY X-MP and CRAY-1 computer systems. The system limits argument lists to 50,000 characters on the CRAY-2 computer system. The number of arguments to a command involving filename expansion is limited to 1/6'th the number of characters allowed in an argument list. Command substitutions can 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.

BUGS

Alias substitution is most often used to clumsily simulate shell scripts; shell scripts 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.

SEE ALSO

sh(1),
access(2), execve(2), fork(2), kill(2), pipe(2), signal(2), umask(2), wait(2), in in the UNICOS System Calls Reference Manual, publication SR-2012
a.out(4F) in in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

csim – Invokes the hardware simulator for the CRAY X-MP and CRAY-1 operating systems

SYNOPSIS

csim [**–i** *ifile*] [**–l** *lfile*] [**–t** *time*] [**–s** *symfile*] [**–m** *fwi*]

DESCRIPTION

The *csim* command invokes the Cray hardware simulator for the CRAY X-MP, CRAY-1, and I/O Subsystems. The *csim* command lets you test several versions of an operating system at the same time, without affecting the production-level operating system. You control input to the simulated system, track performance, and observe reactions at selected points. See the Cray Simulator (CSIM) Reference Manual for details.

The following options are available:

–i *ifile*	Input file name. The default is standard input.
–l *lfile*	List file name. The default is standard output.
–t *time*	Time limit of simulation. The default is 100. You can override **–t** by using the T keyword on the RUN command.
–s *symfile*	Name of symbol file.
–m *fwi*	CSIM message level. **–m** is one of the following:

f	Fatal errors
w	Warning errors
i	Informative errors

EXAMPLE

In this example, CSIM is started with the default time limit. Only fatal and warning messages are printed from the interpreter.

```
csim -m fw
```

FILES

VMEMn	Virtual memory files

BUGS

The file system and system binary must be in the users current directory.

The DSU command does not default properly. You must specify the NS keyword for a new disk.

Checkpoint/reload facilities do not work.

The COS station does not work. Attempts to use it will produce unpredictable results.

Symbol tables are not implemented yet.

SEE ALSO

Cray Simulator (CSIM) Reference Manual (SR-0073)

NAME

csplit – Separates files into sections

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* option is a hyphen (-), then the 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** . . . *prefixn*. The default is **xx00** . . . **xx***n*.

The arguments (*arg1* . . . *argn*) to *csplit* can be a combination of the following:

/rexp/	A file is to be created for the section from the current line up to (but not including) the line containing the regular expression *rexp*. The current line becomes the line containing *rexp*. This 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**.

MESSAGES

Diagnostic messages are self-explanatory, except for the following:

```
            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.

SEE ALSO

ed(1), sh(1)

NAME

cut – Cuts 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, that is, character positions as on a punched card (**–c** option) or the length can vary from line to line and can 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 (either **–c** or **–f** must be specified:

list	Comma-separated list of integer field numbers (in increasing order), with optional **–** to indicate ranges such as, **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 (such as, **–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**); such as, **–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.

HINTS

Use *grep*(1) to make horizontal "cuts" (by context) through a file, or *paste*(1) to put files together column-wise (that is, horizontally). To reorder columns in a table, use *cut* and *paste*.

EXAMPLES

```
cut –d: –f1,5 /etc/passwd
```
Maps user IDs to names; a colon (:) is used as a field separator.
```
name=`who am i | cut –f1 –d" "`
```
Sets *name* to current login name.

MESSAGES

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), join(1), paste(1)

NAME

cvt – Converts files between update and scm formats

SYNOPSIS

cvt toscm
cvt toupdate

DESCRIPTION

Cvt converts *update*(1) source files to *scm*(1) **.m** format, and files in *scm* **.m** format to *update* source format. It reads standard input and writes to standard output. The argument tells *cvt* what type of conversion to perform; **cvt toscm** converts an *update* file to *scm* format, while **cvt toupdate** converts an *scm* file to *update* format.

Both *update* and *scm* tag each line with identifiers that are used to describe the locations of modifications. Each identifier is made up of a program or modification name followed by a '.', followed by a line number (for example, **init.1**, **CFT77.1086**, or **HISTORY.3**). *Update* source files have line identifiers on the right hand side after column 72; the identifiers are adjusted so that the '.' always appears in column 81. *Scm* **.m** files have line identifiers on the left hand side between columns 1 and 24; the identifiers are left justified and are generally lower case.

BUGS

If the input to *cvt* is a file not in *update* or *scm* **.m** format or a file already in the desired format, *cvt* may write nonsense in the output file or produce a core file with no error messages.

SEE ALSO

scm(1)

NAME

cxref – Generates 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 information in **#define** directives 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 *cpp*(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 will default to 80 if *num* is not specified or is less than 51.
–o *file*	Direct output to named *file*.
–s	Operate silently; does not print input file names.
–t	Format listing for 80-column width.

FILES

/usr/lib/xcpp	Special version of C-preprocessor
/usr/lib/xpass	Executable file for cross-reference pass

MESSAGES

Error messages are unusually cryptic, but usually mean that you cannot compile these files.

BUGS

Cxref considers a formal argument in a **#define** macro definition to be a declaration of that symbol. For example, a program that has **#include ctype.h** contains many declarations of the variable **c.**

SEE ALSO

cpp(1)

NAME

date – Prints and sets the date

SYNOPSIS

date [*mmddhhmm*[*yy*]] [+ *format*]

DESCRIPTION

If you do not specify an argument, or if the argument begins with +, the current date and time are printed. Otherwise, the current date is set. The first *mm* is the month number; *dd* is the day number in the month; *hh* is the hour number (24 hour system); the second *mm* is the minute number; *yy* is the last 2 digits of the year number and is optional. For example:

```
date 10080045
```

sets the date to Oct 8, 12:45 AM. The current year is the default if no year is mentioned. The system operates in GMT. *Date* takes care of the conversion to and from local standard and daylight time.

If the argument begins with +, the output of *date* is under the control of the user. The format for the output is similar to that of the first argument to *printf*(3S). All output fields are of fixed size (zero-padded if necessary). Each field descriptor is preceded by **%** and will be replaced in the output by its corresponding value. A single **%** is encoded by **%%**. All other characters are copied to the output without change. The string is always terminated with a new-line character.

Format 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 – Sun to Sat
h	Abbreviated month – Jan to Dec
r	Time in AM/PM notation

EXAMPLE

```
date '+DATE: %m/%d/%y%nTIME: %H:%M:%S'
```

would have generated as output:

```
DATE: 08/01/76
TIME: 14:45:05
```

MESSAGES

no permission	Only the super user can change the date.
bad conversion	The date specified is syntactically incorrect.
bad format character	A field descriptor is not recognizable.

WARNING

It is a bad practice to change the date while the system is running in multi-user mode.

SEE ALSO

printf(3S) in the CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual, publication SR-2013

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)). The overall structure of *dc* is a stacking (reverse Polish) calculator. If an argument is given, input is taken from that file until its end, then from the standard input. The following constructions are recognized:

number
The value of the number is pushed on the stack. A number is an unbroken string of the digits 0 through 9. It may be preceded by an underscore (_) to input a negative number. Numbers may contain decimal points.

+ – / * % ^
The top two values on the stack are added (+), subtracted (–), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. Any fractional part of an exponent is ignored.

s*x* The top of the stack is popped and stored into a register named *x*, where *x* may be any character. If the s is capitalized, *x* is treated as a stack and the value is pushed on it.

l*x* The value in register *x* is pushed on the stack. The register *x* is not altered. All registers start with 0 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 2.

Q The top value on the stack is popped and the string execution level is popped by that value.

x Treats the top element of the stack as a character string and executes it as a string of *dc* commands.

X Replaces the number on the top of the stack with its scale factor.

[...] Puts the bracketed ASCII string onto the top of the stack.

<*x* **>***x* **=***x*
The top two elements of the stack are popped and compared. Register *x* is evaluated if they obey the stated relation. The exclamation point indicates negation.

v Replaces the top element on the stack by its square root. Any existing fractional part of the argument is taken into account, but otherwise the scale factor is ignored.

! Interprets the rest of the line as a UNICOS command. Control returns to *dc* when the command terminates.

c All values on the stack are popped.

i	The top value on the stack is popped and used as the number radix for further input.
I	Pushes the input base on the top of the stack.
o	The top value on the stack is popped and used as the number radix for further output.
O	Pushes the output base on the top of the stack.
k	The top of the stack is popped, and that value is used as a non-negative scale factor: the appropriate number of decimal 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 value of the scale factor onto 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.

EXAMPLE

This example prints the first ten values of n!:

```
[la1+dsa*pla10>y]sy
0sa1
lyx
```

MESSAGES

x is unimplemented	*x* is an octal number.
stack empty	Not enough elements on the stack to do what was asked.
Out of space	Free list is exhausted (too many digits).
Out of headers	Too many numbers being kept around.
Out of pushdown	Too many items on the stack.
Nesting Depth	Too many levels of nested execution.

SEE ALSO

bc(1)

NAME

dd – Converts and copies a file to the specified output

SYNOPSIS

dd [*option=value*] ...

DESCRIPTION

The *dd* command 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*	Inputs file name; standard input is default
of=*file*	Outputs file name; standard output is default
ibs=*n*	Inputs block size *n* bytes (default 4096)
obs=*n*	Outputs block size (default 4096)
bs=*n*	Sets both input and output block size, superseding *ibs* and *obs*; also, if you do not specify a conversion, it is particularly efficient since no in-core copy need be done
cbs=*n*	Conversion buffer size
skip=*n*	Skips *n* input blocks before starting copy; th skipped blocks are actually read, so this can take a considerable amount of time.
iseek=*n*	Seeks *n* input blocks from beginning of input file before starting copy. (this option is invalid on the CRAY X-MP and CRAY-1 computer systems)
seek=*n*	Seeks *n* output blocks from beginning of output file before copying
count=*n*	Copies only *n* blocks
conv=ascii	Converts EBCDIC to ASCII
ebcdic	ConvertsASCII to EBCDIC (does not match the COS conversion exactly)
ibm	Slightly different map of ASCII to EBCDIC
lcase	Maps alphabetics to lower case
ucase	Maps alphabetics to upper case
swab	Swaps every pair of bytes
noerror	Do not stop processing on an error
sync	Pads every input block to *ibs*
. . . , . . .	Several comma-separated conversions

Where sizes are specified, a number of bytes is expected. A number may end with **b**, **s**, **k**, or w to specify multiplication by 4096, 4096, 1024, or 8, 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 trimmed and new-line 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 following command reads an EBCDIC tape blocked ten 80-byte EBCDIC card images per block into the ASCII file **x**:

```
        dd  if=/dev/rmt0  of=x  ibs=800  cbs=80  conv=ascii,lcase
```

Note the use of raw magtape. The *dd* command is especially suited to I/O on the raw physical devices because it permits reading and writing in arbitrary block sizes.

MESSAGES

f+p records in(out)	Numbers of full and partial records read (written)

BUGS

The ASCII/EBCDIC conversion tables are taken from the 256-character standard in the CACM Nov, 1968. The *ibm* conversion corresponds better to certain IBM print train conventions.

New-lines are inserted only on conversion to ASCII; padding is done only on conversion to EBCDIC. These should be separate options.

SEE ALSO

cp(1)

NAME

dda – Invokes the dynamic dump analyzer

SYNOPSIS

dda [**–i** '[s=*symfile*] [**dblog**=*logfile*] [**echo**=*echofile*].']

DESCRIPTION

The *dda* command invokes the dynamic dump analyzer. It is a programming tool for interactively debugging program memory dumps.

For specific information about *dda* directives and debugging, see the Symbolic Debugging Package Reference Manual. The *dda* command accepts the following arguments if you specify the **–i** option (surrounded by quotes and terminated with a period):

s=*symfile*	The *symfile* argument names the file containing the symbol tables. The default is **a.out**.
dblog=*logfile*	Name of the log file that receives a copy of all input to and all output from the debugger. The default is **log.db**. Your program cannot use Fortran I/O unit 99 because *dda* uses it to write **log.db.**
echo=*echofile*	Name of the echo file that receives a copy of all input to the debugger. The default is **echo.db**. Your program cannot use Fortran I/O unit 98 because *dda* uses it to write **echo.db.**

SEE ALSO

as(1), cft(1), pascal(1), segldr(1)
symdebug(3.16) in the Programmer's Library Reference Manual, publication SR-0113
Symbolic Debugging Package Reference Manual, publication SR-0112

NAME

debug – Invokes the postmortem core analyzer

SYNOPSIS

debug [**-i** *'options list.'*]

DESCRIPTION

The *debug* command provides a traceback of a program, and interprets the program memory dump in terms of source language symbols. It is normally used after a program has aborted and a **core** file has been written. The *debug* command reads a symbol file that is created by compilers and the loader. Refer to the command descriptions of the compiler and loader you are using to find out what options are necessary to produce this information.

The *debug* command optionally processes a set of input arguments. These are specified on the **-i** option as a quoted string. The string must be terminated by a dot (.). All arguments are optional and must be separated with commas; the list may consist of any subset of the following:

calls=*n* The *n* argument specifies the number of routine levels to be displayed in the symbolic dump. For each task reported, *debug* traces back through active subprograms the number of levels specified by *n*. Routines for which no symbol table information is available are not counted for purposes of the **calls** count. The default is 50.

tasks Traces back through all existing tasks; the default is to trace back only through tasks that were running when the dump file was written. Deferred implementation

s=*symfile*
The *symfile* argument names the file containing the DEBUG symbol table. The default is **a.out.** With regard to case, *symfile* must be entered exactly as it appears to the system.

syms=*sym*{*:sym*}
List of symbols to be dumped by *debug*. You can specify up to 20 symbols and there is no limit on the length of the symbol name; symbols are separated by a colon (:). The default is that all symbols are skipped. The **syms** argument applies to all blocks dumped.

notsyms=*nsym*{*:nsym*}
List of symbols to be skipped. You can specify up to 20 symbols and there is no limit on the length of the symbol's name; symbols are separated by a colon (:). The default is that no symbols are skipped. This argument takes precedence over the **syms** argument.

maxdim=*dim*{*:dim*}
Maximum number of elements from each dimension of the arrays to be dumped. The **maxdim** allows you to sample the contents of arrays without creating huge amounts of output. When the **maxdim** argument is specified, arrays are dumped in storage order (*row, column* for C and Pascal; *column, row* for CFT and CFT77). No more than 7 dimensions can be specified.

blocks=*blk*{*:blk*}
List of common blocks to be included in the symbolic dump. You can specify a maximum of 20 blocks separated by colons. All symbols (qualified by **syms** and **notsyms** arguments) in the named blocks are dumped. The default is to dump no common blocks. If you specify **blocks** with any *blk* values, all blocks are dumped.

notblks=*nblk*{*:nblk*}
List of common blocks to be excluded from the symbolic dump. You can specify a maximum of 20 blocks separated by colons. This argument takes precedence over the **blocks** argument.

rptblks Repeat blocks; when this option is used, the contents of common blocks that you specified using the **blocks** and **notblks** arguments are displayed for each subroutine in which they are declared. The default displays common blocks only once.

mtbuf=*m*
Number of entries in the multitasking history trace buffer to list. If you do not specify *m*, the whole buffer is displayed. If you do not specify this argument and the buffer is present, the last 25 entries are displayed. Deferred implementation.

pages=*np*
Under UNICOS, *debug* does not format output in pages; This argument can still be used to regulate the amount of output that *debug* generates. Every page is worth 45 lines of output from *debug*, so **page=10** limits the output to 450 lines. The default for *np* is 70.

FILES

SYMBOLS	Symbolic information
DBSYM	Temporary use by *debug*
DBBLK	Temporary use by *debug*
core	Program memory dump

BUGS

Support for multitasked programs is not yet available.

SEE ALSO

as(1), cft(1), pascal(1), segldr(1)
symdebug(3.16) in the Programmer's Library Reference Manual, publication SR-0113.
Symbolic Debugging Package Reference Manual, publication SR-0112.

NAME

delta – Makes a delta (change) to an SCCS file

SYNOPSIS

delta [**–r** *SID*] [**–s**] [**–n**] [**–g** *list*] [**–m** [*mrlist*]] [**–y** [*comment*]] [**–p**] *files*

DESCRIPTION

The *delta* command is used to permanently introduce into the named SCCS file changes that were made to the file retrieved by *get*(1) (called the *g-file*, or generated file).

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

The *delta* command may issue prompts on the standard output depending upon certain keyletters specified and flags (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 nonzero exit status is returned from MR number validation program, *delta* terminates (it is assumed that the MR numbers were not all valid).

–y [*comment*]
: Arbitrary text used to describe the reason for making the delta. A null string is considered a valid *comment.*

  If **–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 (SCCS) User's Guide.* The naming convention for these files is also described there.

g-file	Existed before the execution of *delta*; removed after completion of *delta*.
p.*file*	Existed before the execution of *delta*; may exist after completion of *delta*.
q.*file*	Created during the execution of *delta*; removed after completion of *delta*.
x.*file*	Created during the execution of *delta*; renamed to SCCS file after completion of *delta*.
z.*file*	Created during the execution of *delta*; removed during the execution of *delta*.
d.*file*	Created during the execution of *delta*; removed after completion of *delta*.
/usr/bin/bdiff	Program to compute differences between the "gotten" file and the *g-file*.

WARNINGS

Lines beginning with an **SOH ASCII** character (binary 001) cannot be placed in the SCCS file unless the **SOH** is escaped. This character has special meaning to SCCS (see *sccsfile*(4F)) 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.

MESSAGES

Use *help*(1) for explanations.

SEE ALSO

admin(1), bdiff(1), cdc(1), get(1), help(1), prs(1), rmdel(1)
sccsfile(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014
The Source Code Control System (SCCS) User Guide, publication SG-2017

NAME

df – Reports the number of free disk blocks

SYNOPSIS

df [**–t**] [**–f**] [**–p**] [*file-systems*]

DESCRIPTION

The *df* command prints out the number of free blocks and free i-nodes available for on-line file systems by examining the counts kept in the super-blocks; you can specify *file-systems* either by device name (such as **/dev/dsk/0s0** on the CRAY-2 computer system and **/dev/d4901** on the CRAY X-MP and CRAY-1 computer systems) or by mounted directory name (such as **/usr**). If you do not specify the *file-systems* option, the free space on all of the mounted file systems is printed.

The following options are available:

–t Reports the total allocated block figures.

–f Reports only an actual count of the blocks in the free list (free i-nodes are not reported). With this option, *df* reports on raw devices.

–p The **–p** option is only available on the CRAY X-MP and CRAY-1 computer systems. It displays partitions on the filesystem in the following format:

```
Part     Start     Total     Free        Frags       Device
 0        0         100      50 (50%)    10 (40%)     A131
```

Where, *part* is the relative partition number, *start* is the starting block number, *total* is the total block count of partitions, *free* is the free block count of partitions, *frags* is the count of uncontiguous areas in the partition, and *device* is the ASCII name of the device on which the partition resides.

FILES

/dev/dsk/*	Disk devices on the CRAY-2 computer system
/dev/d*	Disk devices on the CRAY X-MP and CRAY-1 computer systems
/etc/mnttab	List of currently mounted file systems

SEE ALSO

fs(4F), mnttab(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

diff – Compares files for differences

SYNOPSIS

diff [**–efbh**] *file1 file2*

DESCRIPTION

The *diff* command indicates 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 is a list of all lines that are affected in the first file flagged by <, then all the lines that are affected in the second file flagged by >.

Options for *diff* follow.

–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 option **–e** but in the opposite order. This script is not useful with *ed*.

–h Does a fast, half-hearted job. It works only when changed stretches are short and well separated, but does work on files of unlimited length. Options **–e** and **–f** are not available with **–h**.

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** option

MESSAGES

Exit status is 0 for no differences, 1 for some differences, 2 for trouble.

Missing newline at end of file *x*
The last line of file *x* did not have a newline character. If the lines are different, they are flagged and output; however the output will indicate they are the same.

BUGS

Editing scripts produced under the **-e** or **-f** option are naive about creating lines consisting of a single period (.).

SEE ALSO

bdiff(1), cmp(1), comm(1), diff3(1) (CRAY-2 only), ed(1), scm(1)
The Source Code Control System (SCCS) User Guide, publication SG-2017

NAME

diff3 – Makes a 3-way differential file comparison

SYNOPSIS

diff3 [**–ex3**] *file1 file2 file3*

DESCRIPTION

The *diff3* command 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* = *n*2, the range may be abbreviated to *n1*.

The original contents of the range follow immediately after a **c** indication. When the contents of two files are identical, the contents of the lower-numbered file are suppressed.

Under the **–e** option, *diff3* publishes a script for the editor *ed* that will incorporate into *file1* all changes between *file2* and *file3*; that is, 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

BUGS

Text lines that consist of a single **.** will defeat the **–e** option.

There is an arbitrary limit of 200 on the total number of disagreeing ranges of text.

SEE ALSO

diff(1)

NAME

dircmp – Compares directories

SYNOPSIS

dircmp [**–d**] [**–s**] [**–w***n*] *dir1 dir2*

DESCRIPTION

The *dircmp* command 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 you do not specify an option, a list is output indicating whether the files common to both directories have the same contents. Options are:

–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)

NAME

dispose – Disposes a file from the Cray computer system to a front-end station

SYNOPSIS

dispose *localpath* [**–n***SFN*] [**–i***TERMID*] [**–m***MF*] [**–d***DC*] [**–f***FM*] [**–t**'*TEXT*'] [**–s***SPECIAL*] [**–u***USER*]

DESCRIPTION

The *dispose* command creates a request file for USCP (UNICOS Station Call Processor). If any slot information is associated with the requesting user, it is also copied into the request file. USCP uses station protocol to negotiate the file transfer from the Cray computer system to the designated station (specified by the *–m* option). The *dispose* command then waits for the transfer to complete.

localpath The path name (either full or relative to the current working directory) of the file to be disposed. This must be a path name from which the requesting user has permission to read.

–n*SFN* The name to be associated with the file when it is received by the front end. Only 15 characters are significant. If you do not specify *SFN*, the field is filled with the filename from the *localpath*.

–i*TERMID* The terminal ID to be associated with the file on the specified front end. If you do not specify *TERMID*, the default is the stored terminal ID associated with the requesting user on the front-end station from which the user originated.

–m*MF* A two character front-end ID for the station that is to handle the file transfer. If you do not specify the mainframe, then the stored ID of the station from which the requesting user originated is used.

–f*FM* A two character file format code. Valid formats are:

CB	Character blocked; the default
CD	Character deblocked
BB	Binary blocked
BD	Binary deblocked
TR	Transparent
UD	UNICOS Data

For further descriptions of the valid format codes, see the Front End Protocol Internal Reference Manual, CRI publication SM-0042.

–d*DC* A two character disposition code. Valid codes are:

IN	File is executed as a job.
ST	File is saved.
MT	File is disposed to a magnetic tape.
PR	File is disposed to a printer.
PU	File is disposed to a card punch.
PT	File is disposed to a plotter.
IT	File is flagged as intertask data and handled by the receiving station.

–t'*TEXT*'	Text to be interpreted by the specified station for dispose processing. The field can contain label information, routing, etc., possibly in the form of control statements for the station. Text field information should be enclosed by single quotes ('). If you do not specify this option, the dispose text field is filled with binary 0's.
–s*SPECIAL*	The station-defined special forms option. If you do not specify this option, the special forms field is filled with binary 0's.
–u*USER*	The user ID associated with the requested file on the specified front end. If you do not specify *USER*, the field is left blank.

LIMITATIONS

If you are not accessing the Cray computer system through USCP, defaults for *TERMID* and *MF* do not exist. The request is queued without regard to whether the mainframe ID you specified belongs to a currently active station. If the associated station is not active or does not have streams assigned (that is, interactive only station), then the user process wait indefinitely.

SEE ALSO

fetch(1), dispose(1)
Front End Protocol Internal Reference Manual, publication SM-0042

NAME

drd – Invokes the dynamic runtime debugger

SYNOPSIS

drd [**-i** '[s=*symfile*], [**dblog**=*logfile*], [**echo**=*echofile*], [**maxbp**=*n*],
[**prog**="*command*"].']

DESCRIPTION

The *drd* command invokes the dynamic runtime debugger to use as a programming tool for debugging executing programs. You can use the *drd* command in either interactive or batch mode. The *drd* command accepts the following options:

s=*symfile*	*Symfile* names the file containing the symbol table. The default is **a.out.** With regard to case, *symfile* must be entered in exactly as it appears to the system.
dblog=*logfile*	*logfile* names the debug log file; the default is **log.db.** Your program cannot use Fortran I/O unit 99 because *drd* uses it to write the debug log file.
echo=*echofile*	*echofile* names the echo file; the default is **echo.db.** Your program cannot use Fortran I/O unit 98 because *drd* uses it to write the echo file.
maxbp=*n*	*n* names the maximum breakpoint limit; by default, there is no breakpoint limit.
prog=*command*	*command* invokes the user program, including all options. The default is **a.out.** Enclose all input entered in the *command* portion in double quotes (").

SEE ALSO

as(1), cft(1), pascal(1), segldr(1)
symdebug(3.16) in the Programmer's Library Reference Manual, publication SR-0113
Symbolic Debugging Package Reference Manual, publication SR-0112

NAME

du – Summarizes disk usage

SYNOPSIS

du [**–ars**] [*names*]

DESCRIPTION

The *Du* command gives the number of blocks contained in all files and (recursively) directories within each directory and file specified by the *names* argument. The block count includes the indirect blocks of the file. If *names* is not supplied, the current directory is used.

The *du* command accepts the following arguments:

–s	Causes only the grand total (for each of the specified *names*) to be given
–a	Causes an entry to be generated for each file. Absence of either this argument or **–s** causes an entry to be generated for each directory only.
–r	Causes *du* to generate messages about directories that cannot be read, files that cannot be opened, and the like. The *du* command is normally silent about such things.

A file with two or more links is only counted once.

BUGS

If you do not use the **–a** option, nondirectories given as arguments are not listed.
If there are too many distinct linked files, *du* counts the excess files more than once.
Files with holes in them will have an incorrect block count.

NAME

echo – Echos arguments

SYNOPSIS

echo [*arg*] ...

DESCRIPTION

The *echo* command writes its arguments separated by blanks and terminated by a new-line on the standard output. It also understands C-like escape conventions; beware of conflicts with the shell's use of \:

\b	Backspace
\c	Print line without new-line
\f	Form-feed
\n	New-line
\r	Carriage return
\t	Tab
\v	Vertical tab
\\	Backslash
\*n*	The 8-bit character whose ASCII code is the 1-, 2- or 3-digit octal number *n*, which must start with a 0.

The *echo* command is useful for producing diagnostics in command files and for sending known data into a pipe.

NOTES

csh(1) has a built-in echo with slightly different characteristics. See csh(1)

SEE ALSO

sh(1)

NAME

ed, red – Invokes the ed text editor

SYNOPSIS

ed [**–**] [**–p** *string*] [**–d** *dir*] [**–x**] [*file*]

red [**–**] [**–p** *string*] [**–d** *dir*] [**–x**] [*file*]

DESCRIPTION

Ed is the standard text editor. If the *file* argument is given, *ed* simulates an *e* command (described later) on the named file; that is to say, the file is read into *ed*'s buffer so that it can be edited. The optional – (hyphen) suppresses the printing of character counts by *e*, *r*, and *w* commands, of diagnostics from *e* and *q* commands, and of the ! prompt after a !*shell command*. The **–p** option allows the user to specify a prompt string. If **–x** is present, an *x* command is simulated first to handle an encrypted file. The **–d** option allows temporary files to be created in file systems with large free space. *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. A 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*).

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.

Regular Expressions

Ed supports a limited form of *regular expression* notation; regular expressions are used in addresses to specify lines and in some commands (such as *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) is a one-character RE that matches itself.

1.2 A backslash (\) followed by any special character is a one-character RE that matches the special character itself. The special characters are:

   a. ., *, [, and \ (period, asterisk, left square bracket, and backslash, respectively), which are always special, *except* when they appear within square brackets ([ ]; see 1.4).

   b. ^ (caret or circumflex), which is special at the *beginning* of an *entire* RE (see 3.1 and 3.2), or when it immediately follows the left of a pair of square brackets ([ ]) (see 1.4).

c. **$** (currency symbol), which is special at the *end* of an entire RE (see 3.2).

d. The character used to bound (that is, delimit) an entire RE, which is special for that RE (for example, see how slash (/) is used in the *g* command, described later.)

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); for instance, **[ ]a–f]** matches either a right square bracket (]) or one of the letters **a** through **f** inclusive. The four characters listed in 1.2.a above stand for themselves within such a string of characters.

The following rules may be used to construct REs from one-character REs:

2.1 A one-character RE is a RE that matches whatever the one-character RE matches.

2.2 A one-character RE followed by an asterisk (*) is a RE that matches *zero* or more occurrences of the one-character RE. If there is any choice, the longest leftmost string that permits a match is chosen.

2.3 A one-character RE followed by **\{***m***\}**, **\{***m***,\}**, or **\{***m,n***\}** is a RE that matches a *range* of occurrences of the one-character RE. The values of *m* and *n* must be 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 (such as, //) is equivalent to the last RE encountered.

Addressing

To understand addressing in *ed*, it is necessary to know that at any time there is a *current line*. 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 **$** addresses the last line of the buffer.
3. A decimal number *n* addresses the *n*-th line of the buffer.
4. *'x* addresses the line marked with the mark name character *x*, which must be a lower-case letter. Lines are marked with the *k* command described below.
5. A RE enclosed by slashes (/) addresses the first line found by searching *forward* from the line *following* the current line toward the end of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the beginning of the buffer and continues up to and including the current line, so that the entire buffer is searched. See also the last paragraph before *FILES* below.
6. A RE enclosed in question marks (?) addresses the first line found by searching *backward* from the line *preceding* the current line toward the beginning of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the end of the buffer and continues up to and including the current line. See also the last paragraph before *FILES* below.
7. An address followed by a plus sign (+) or a minus sign (–) followed by a decimal number specifies that address plus (or minus) the indicated number of lines. The plus sign may be omitted.
8. If an address begins with + or –, the addition or subtraction is taken with respect to the current line; such as, **–5** is understood to mean **.–5**.
9. If an address ends with + or –, then 1 is added to or subtracted from the address, respectively. As a consequence of this rule and of rule 8 immediately above, the address – refers to the line preceding the current line. (To maintain compatibility with earlier versions of the editor, the character ^ in addresses is entirely equivalent to –.) Moreover, trailing + and – characters have a cumulative effect, so — refers to the current line less 2.
10. For convenience, a comma (,) stands for the address pair **1,$**, 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 ones 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 displayed; *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 *MESSAGES*.

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 *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 message.

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 was 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 (such as *tab* and *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 lines after the line addressed by *a*. Address 0 is legal for *a* and causes the addressed lines 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 *MESSAGES* 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 displayed; **.** is set to the last line read in. If *file* is replaced by **!**, the rest of the line is taken to be a shell command whose output is to be read. (Refer to **!***shell command* below.) 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) and *umask*(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 displayed. If *file* is replaced by !, the rest of the line is taken to be a shell command whose standard input is the addressed lines. (Refer to !*shell command* below.) 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 UNICOS system shell (*sh*(1)) by default or to the value of the SHELL environment variable, if set and exported, 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.

LIMITATIONS

Size limitations: Large files generate larger editor temporary files and cost many processor cycles on entry to *ed*. The buffer is limited to approximately 17 gigabytes (on the CRAY-2 Computer System). Reasonable editing sessions should be kept under 10 megabytes. Lines are limited to 4096 characters.

When reading a file, *ed* discards ASCII NUL characters and all characters after the last new-line. Files (such as **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 (such as, /) would be the last character before a new-line, that delimiter may be omitted, in which case the addressed line is displayed. The following pairs of commands are equivalent:

s/s1/s2	s/s1/s2/p
g/s1	g/s1/p
?s1	?s1?

FILES

/tmp/e#	Temporary; # is the process number.
ed.hup	Work is saved here if the editor is killed with signal 1; see *signal*(2).

MESSAGES

?	Command errors.
?file	An inaccessible file.

If changes have been made in the buffer since the last *w* command that wrote the entire buffer, *ed* warns the user if an attempt is made to destroy *ed*'s buffer via the *e* or *q* commands. It prints ? and allows one to continue editing. A second *e* or *q* command at this point will take effect. The – command-line option inhibits this feature.

BUGS

A *!* command cannot be subject to a *g* or a *v* command.

The *!* command and the ! escape from the *e*, *r*, and *w* commands cannot be used if the the editor is invoked from a restricted shell (see *sh*(1)).

The sequence \n in a RE does not match a new-line character.
The *l* command mishandles DEL.

Files encrypted directly with the *crypt*(1) command with the null key cannot be edited.

Characters are masked to 7 bits on input.

If the editor input is coming from a command file (that is, **ed** *file* < *ed-cmd-file*), the editor will exit at the first failure of a command that is in the command file.

For UNICOS running on CRAY X-MP or CRAY-1 mainframes, *ed* truncates large files without warning.

SEE ALSO

crypt(1), grep(1), sed(1), sh(1)
regexp(3C) in the CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual, publication SR-2013
The UNICOS Primer, publication SR-2010

NAME

env – Sets 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 (hyphen) 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) in the UNICOS System Calls Reference Manual, publication SR-2012
profile(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

ex – Invokes the ex text editor

SYNOPSIS

ex [**–**] [**–v**] [**–t** *tag*] [**–r** [*file*]] [**–R**] [*+command*] [**–l**] [**–x**] *name* ...

DESCRIPTION

The *ex* command invokes the *ex* editor. *Ex* is a line-based editor that you can use alone or as an extension of the screen-based visual editor, *vi*. If you are not familiar with these editors, please refer to *ed*(1), and *vi*(1).

Ex can perform functions that *vi* alone cannot. *Ex* is useful for making large-scale changes to more than one part of a file, such as performing global text changes, moving text between files, or other advanced editing procedures. You can also perform functions like writing your file without having to quit the file. (However, many commands in *ex* duplicate commands in vi.)

You can access *ex* in one of two ways: on the command line, or from within *vi*.

Ex can be invoked on the command line like any other command. You will want invoke *ex* in this way if you are using it alone. **–** Suppresses all interactive-user feedback. This is useful in processing editor procedures.

–v	Invokes the *vi* editor
–t *tag*	Allows you to edit the file containing the *tag* and position the editor at its definition
–r [*file*]	Recovers *file* after an editor or system crash. If *file* is not specified, a list of all saved files is printed.
–R	Sets *read-only* mode and prevents you from accidentally overwriting the file.
+command	Allows you to begin editing by executing the specified editor search or positioning *command*.
–l	Invokes LISP mode. Indents appropriately for lisp code, the () {} [[and]] commands in *ex* are modified to have meaning for *lisp*.
–x	Invokes encryption mode; a key is prompted for allowing creation or editing of an encrypted file.
name	Indicates files to be edited

Once you have opened a file with *ex*, you will get a colon (:) prompt, which indicates you are in command mode. At this point, you must give *ex* a command (see the Commands subsection).

Ex can be in one of the following modes at a time:

Command	Normal and initial state. Input prompted for by :. Your kill character cancels a partial command.
Insert	Entered by **a i** and **c**. You can enter arbitrary text once in this mode. Insert is normally terminated by entering a **.** on a line by itself, or abnormally terminated with an interrupt.
Visual	Invokes *vi*by entering **vi**, terminates with **Q** or **e**.

To invoke *ex* once you've opened a file with *vi* (you must be within *vi*'s command mode), type a colon (:). At this point, a colon appears in the lower right-hand corner of the screen. Now you are in *ex* and can type any *ex* command (see the Commands subsection). *Ex* returns control to *vi* after every command has completed. The syntax of an *ex* command is as follows:

: [*m*],[*n*]*command* [*argument*] newline where, *m* and *n* are the optional line numbers on which to perform the command, *command* is the command name from the list in the next subsection, *argument* specifies a file or text for some commands, and newline is the newline character you need to enter to invoke the command. If you do not specify line numbers on the commands that require them, the command performs the action on the current line.

Commands

Following is a list of command names and their aliases that *ex* accepts.

abbrev *word string*
alias: **ab**
Uses the abbreviation *word* for the character string *string*. When in visual mode, if *word* is typed as a complete word, it is changed to *string*.

append
alias: **a**
Appends text and places it after the specified line. Terminated by a **.** (dot).

args
alias: **ar**
Prints the members of the argument list. The current argument is delimited by brackets ([]).

change
alias: **c**
Replaces specified lines with text. Terminated by **.** (dot).

copy
alias: **co**
Copies the specified lines. For example, **:1,10co20** copies lines 1 through 10 after line 20.

delete
alias: **d**
Deletes the specified line numbers. For example, **:1,10d** deletes lines 1 through 10.

edit
alias: **e**
Edits either the current file, disregarding any changes made so far (**:e!**), or edits a new file (**:e** *file*). Changes are not lost in the current file and control returns to the shell after editing the new file.

file
alias: **f**
Prints the current file and line number.

global
alias: **g**
Prints certain lines; makes global searches and changes. For example, **:g/***text* moves the cursor to the last line in the file that contains *text*. You can also use **g** to print certain lines of the file: **:g/***text***/p** or **nu** prints all lines containing *text* (if you specify **nu** instead, it prints the line numbers also.

The following command can be used to list, one at a time, each line containing *text* and change to *newtext* as required (the editor prompts you if you want the change made. Respond with a **y** or **n**): **:g/***text***/s//***newtext***/c**

insert
alias: **i**
Places text before the specified line. Terminated with **.** (dot).

join
alias: **j**
Places text from specified lines on one line (that is, joins lines of text).

list
alias: **l**
Prints the specified lines with tabs shown as ^I and the end of the line marked with a trailing $. This does not change the contents of the edit buffer.

map *lhs rhs*
alias: **map**
Defines macros for use in visual mode. *lhs* is a single character; *rhs* is a sequence of ex commands. When *lhs* is typed, it behaves as if *rhs* has been typed.

mark *x*
alias: **ma**
Marks the specified line with character *x*.

move
alias: **m**
Moves the specified line numbers. For example, **:1,10m32** moves line numbers 1 through 10 after line 32.

number
alias: **nu**
Prints the specified lines; each line is preceded by its (buffer) line number.

preserve
alias: **pre**
Preserves the buffer.

print
alias: **p**
Prints the specified line numbers. For example, **:1,10p** prints lines 1 through 10. **:p** with no line specifications prints the current line.

put
alias: **pu**
Puts back previously deleted or yanked lines (with **delete** and **yank**, respectively.

quit
alias: **q**
Exits the editor. Use **:q!** to quit without saving your changes.

recover
alias: **rec**
Recovers the buffer after a system crash, **:pre**, or disconnect.

rewind
alias: **rew**
Rewinds the argument list and edits the first file in the list.

set
alias: **se**
Sets editing initialization options. Options set with **set** last only while you are in the editor. Set has the following syntax:

:set [*argument*] [*option*]

Set with no arguments shows the options you have changed. **:set all** shows the state of all options. **:set** *x* enables the *x* option. **:set no***x* disables the *x* option. **:set** *x=val* gives the *x* option the value of *val*. **:set** *x*? shows the value of the *x* option.

A list of options follows (the alias, if one exists is in parentheses):

autoindent(ai)	Supply indent
autowrite(aw)	Write before changing files
ignorecase(ic)	Ignore case when scanning
lisp	() { } are s-expressions
list	Print ^I for tab, $ at end of line
magic	Turn on the normal metacharacter meaning of ., [, *.
number(nu)	Number lines
paragraphs(para)	Option's value is the name of the macros that start paragraphs
redraw	Redraw the screen
scroll	Command mode lines
sections(sect)	Specifies section macro names ...
shiftwidth(sw)	Gives the width of a software tab stop used in reverse tabbing
showmatch(sm)	Shows the matching to) and { as typed
showmode(smd)	Show insert mode in *vi*
slowopen(slow)	Stop updates during insert
window	Visual mode lines
wrapscan(ws)	Searches using regular expressions will wrap around past eof
wrapmargin(wm)	Automatically splits line n characters from right

shell
alias: **sh**

Escapse to the shell without writing your file. Which shell you get is specified by the $SHELL environment variable.

stop
alias: **st**

Suspends ex and returns control to the calling shell.

substitute
alias: **s**

Substitutes one string for another. For example, **:s/***y***/***x***/[gcp]** substitutes string *y* for string *x*. **g**,**c**, and**p** respectively, change every occurrence in the line, confirm each change before it's made, and print changed lines.

unabbrev *word*
alias: **una**

Deletes *word* from the abbreviation list.

read
alias: **r**

version
alias: **ve**

visual
alias: **vi**

write
alias: **w**

xit
alias: **x**
Writes changes if any have been made and not written, then quits.

yank *buffer*
alias: **ya**
Places specified lines in a buffer named *buffer*. These lines can be retrieved with the **put** command.

window
alias: **(**

escape
alias: **!**

undo
alias: **u**

print next
alias: **<CR>**

source
alias: **so**

rshift
alias: **>**

scroll
alias: **CONTROL-D**

Line Addressing Symbols

Ex accepts the following command addresses:

n
Line *n*

.(dot)
Specifies the current line

/pat
Goes to the next line containing the string *pat*

?pat
Goes to the previous line containing the string *pat*

FILES

/usr/lib/ex?.?strings	Error messages
/usr/lib/ex?.?recover	Recover command
/usr/lib/ex?.?preserve	Preserve command
/usr/lib/*/*	Describes capabilities of terminals
$HOME/.exrc	Editor startup file
./.exrc	Editor startup file
/tmp/Ex*nnnnn*	Editor temporary
/tmp/Rx*nnnnn*	Temporary files for ex
/usr/preserve	Preservation directory

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 screenfull 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 you if you put text in named buffers and do not use the text before exiting the editor.

Null characters are discarded in input files and cannot appear in resultant files.

SEE ALSO

awk(1), ed(1), edit(1), grep(1), sed(1), vi(1).
term(4F), terminfo(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

exdf – Transfers files to and from the IOS partition of the expander disk

SYNOPSIS

exdf [**–{i|o}** *dirname/filename*] [**–r**]

DESCRIPTION

Exdf can read or write a file on the IOS partition of the expander disk. The program determines from the command line whether it is reading or writing an expander disk file. The **–i** option indicates input from the expander disk, and the **–o** option indicates output to a disk file. The *dirname/filename* is the directory/file name entry as known to the IOS. All IOS directory file name combinations are automatically forced to upper case letters. Standard input or output is used for the UNICOS file. UNICOS directory and file names can be specified through the normal redirect methods. *Exdf* can easily be used as a link in a pipe to move files onto or off of the expander disk.

The **–r** option indicates whether a file being output to the expander disk can replace an existing file. The default is to abort a transfer when the file already exists. The option is ignored for files being read from the expander disk.

WARNING

The IOS has its own conventions for directory names, and file names. Please observe the conventions.

BUGS

The IOS and Cray computer systems force full words to be transferred. This results in null characters added to the end of files. This will be a problem with text files that you want to compile. The C preprocessor blows up when it runs into the nulls. *Exdf* cannot filter the data because the last bytes in a file are not always accompanied by the end of file status.

Character files on the IOS do not have carriage return and line feed as end of lines, but apparently just a carriage return. Develop your own filter if this is a problem.

If *exdf* cannot write a file to the expander disk because the disk is write-protected, no error messages are displayed.

NAME

exlp – Prints files on the expander line printer

SYNOPSIS

exlp [**–p**] [**–r**] [**–s**] [**–w**] [*file* ...]

DESCRIPTION

The *exlp* command takes standard input, a file, or a file list and prints the files on the expander printer, **/dev/lp**. If a list of files is specified, the files are printed in the order specified with a blank page between each file.

The **–p** option switches the expander printer into plot mode. The **–r** option prints the files rotated 90 degrees on the paper. The **–s** option is the silent switch, and will turn off warning messages. The **–w** option is useful when the rotated option, **–r**, is also specified. The **–w** option specifies that the files are to be printed in 132 column mode. The default setting when only the rotate option is set is 80 columns. The maximum number of columns is 132 for all files printed without the rotate option set.

WARNING

The *exlp* command ignores attempts to redirect output to other files or devices.

BUGS

The *exlp* command can be terminated by the normal interrupt character (control-C) or the eof character (control-D), but it takes a little while for the printer to be closed.

NAME

expand, unexpand – Expands tabs to spaces, and vice versa

SYNOPSIS

expand [*–tabstop*] [–tab1,tab2,...,tabn] [*file* ...]
unexpand [**–a**] [*file* ...]

DESCRIPTION

The *expand* command changes tabs into blanks in the named files (or the standard input if you do not specify any files) and writes the files to standard output. *Expand* preserves backspace characters into the output and decrements the column count for tab calculations. *Expand* is useful for pre-processing character files (for example, before sorting, looking at specific columns,) that contain tabs.

If you specify a single *tabstop* argument, tabs are set *tabstop* spaces apart instead of the default 8. If you specify multiple tabstops, then the tabs are set at those specific columns.

The *unexpand* command puts tabs back into the data from the standard input or the named files and writes the result to the standard output. By default, only leading blanks and tabs are reconverted to maximal strings of tabs. If you specify the **–a** option, tabs are inserted whenever they would compress the resultant file by replacing two or more characters.

NAME

expr – Evaluates 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 64-bit, twos complement numbers.

The operators and keywords are listed below. Characters that need to be escaped are preceded by \. The list is in order of increasing precedence, with equal precedence operators grouped within { } symbols.

expr **\\|** *expr* Returns the first *expr* if it is neither null nor **0**, otherwise returns the second *expr*.

expr **\\&** *expr*
Returns the first *expr* if neither *expr* is null or **0**, otherwise returns **0**.

expr { =, \\>, \\>=, \\<, \\<=, != } *expr*
Returns the result of an integer comparison if both arguments are integers, otherwise returns the result of a lexical comparison.

expr { +, – } *expr*
Addition or subtraction of integer-valued arguments.

expr { \\*, /, % } *expr*
Multiplication, division, or remainder of the integer-valued arguments.

expr **:** *expr*
The matching operator **:** compares the first argument with the second argument, which must be a regular expression. Regular expression syntax is the same as that of *ed*(1), except that all patterns are "anchored" (that is, 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.

EXAMPLES

a=\`expr $a + 1\` Adds 1 to the shell variable **a**.

# ´For $a equal to either "/usr/abc/file" or just "file"´

expr $a : ´.*/\\(.*\\) \\| $a
Returns the last segment of a path name (that is, the *filename*). Watch out for / alone as an argument; *expr* will take it as the division operator (see BUGS).

# A better representation of example 2.

expr //$a : ´.*/\\(.*\\)´ The addition of the // characters eliminates any ambiguity about the division operator and simplifies the whole expression.

expr $VAR : ´.*´ Returns the number of characters in **$VAR**.

MESSAGES

syntax error	Operator/operand errors
non–numeric argument	Arithmetic is attempted on a non-integer string

As a side effect of expression evaluation, *expr* returns the following exit values:

0	The expression is neither null nor **0**
1	The expression *is* null or **0**
2	Invalid expressions

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=
```

SEE ALSO

ed(1), sh(1)

NAME

extd – Transfers files to and from the IOS expander tape drive

SYNOPSIS

extd [**–**{**i**|**o**} *message*] [**–r**] [**–n** *filenumber*]

DESCRIPTION

The *Extd* command can read or write a file on an expander tape. The program determines from the command line whether it is reading or writing an expander tape. The **–i** option indicates input from an expander tape, and the **–o** option indicates output to a tape. *Message* is the mount message to be displayed on the IOS console. The mount message can be up to 7 characters long. All IOS mount messages are automatically forced to upper case letters. Standard input or output is used for the UNICOS file. UNICOS directory and file names can be specified through the normal redirection methods. The *extd* command can easily be used as a link in a pipe to move files onto or off of an expander tape.

The **–r** option is the "remain in position" option; it forces the drive to not rewind on close. This option can be used to read or write multiple files to a tape. The option should be used in conjunction with the **–n** *filenumber* to specify which file is being read or written. The **–r** option should not be used on the last invocation of *extd* because the drive remains reserved in the IOS. The next user of the drive will be denied access to the IOS.

The **–n** option specifies the number of the file to be read or written on the tape. The first file is number 0, the second is 1, etc.

WARNING

The IOS has its own conventions for mount messages. Please observe the conventions.

Be aware that the drive does stay reserved for the *pid* of the user who invokes *extd* with the **–r** option until the drive has been opened with rewind specified by that *pid*, or aborted at the IOS console. This does provide some interlocking for the tape, but should be used with caution.

NAME

factor – Factors a number

SYNOPSIS

factor [*number*]

DESCRIPTION

When *factor* is invoked without an argument, it waits for a number to be typed in. If you type in a positive number less than or equal to 1.0×10^{14}, it will factor the number and print its prime factors; each one is printed the proper number of times. Then it waits for another number. It exits if it encounters a 0 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.

MESSAGES

Ouch!	Input out of range or garbage input

NAME

fetch - Requests a file from a front-end station

SYNOPSIS

fetch *localpath* [**–n***SFN*] [**–i***TERMID*] [**–m***MF*] [**–d***DC*] [**–f***FM*] [**–t**'*TEXT*']
[**–u***USER*]

DESCRIPTION

The *fetch* command creates a request file for USCP (UNICOS Station Call Processor). If any slot information is associated with the requesting user, it is also copied into the request file. USCP uses station protocol to make the request for a file from the designated station (specified by the *–m* option). The fetch process then waits until the transfer status has been determined. The transfer status is returned when a negative reply is received from the station (requested file did not transfer) or when a positive reply is received from the station (requested file has been saved on the Cray computer system). It is possible for the station to return a postpone status, in which case the fetch process resets the request for USCP to find and again waits for the transfer status.

localpath A path name (either full or relative to current working directory) where the requested file is to reside when the transfer is complete. The *localpath* must be a location where the requesting user has permission to write. This is a required argument.

–n*SFN* Name associated with requested file on the specified front end. This argument is stored in the request record PDN field. Only 15 characters are significant. If you do not specify *SFN*, this field is filled the filename from the *localpath*.

–i*TERMID* The terminal ID associated with the requested file on the specified front end. The size limit is 8 characters. If you do not specify *TERMID*, the default is the stored terminal ID associated with the requesting user on the front-end station from which the user originated.

–m*MF* *MF* is a two character front-end ID for a station that has access to the requested file. If you do not specify the mainframe, the stored ID of the station from which the requesting user originated is used.

–f*FM* *FM* is a two character file format code. Valid formats are :

CB	Character blocked; the default
CD	Character deblocked
BB	Binary blocked
BD	Binary deblocked
TR	Transparent
UD	UNICOS Data

For further descriptions of the valid format codes, see the Front End Protocol Internal Reference Manual, CRI publication SM-0042.

–d*DC* A two character disposition code interpreted by the receiving system. Valid codes are:

IN	File is executed as a job.
ST	File is saved.

-t'*TEXT*' Text to be interpreted by the specified station for processing of the request. The field can contain label information, routing, etc., possibly in the form of control statements for the station. Text field information should be enclosed by single quotes ('). If you do not specify this option, the request text field is filled with binary 0's.

-u*USER* User ID associated with the requested file on the specified front end. If you do not specify *USER*, this field is left blank for the request.

LIMITATIONS

If you are not accessing the Cray computer system through USCP, defaults for *TERMID* and *MF* do not exist. The request is queued without regard to whether the mainframe ID specified belongs to a currently active station. If the associated station is not active or has no streams assigned (that is, interactive only station), the user process waits indefinitely.

SEE ALSO

dispose(1), acquire(1), uscpintro(1)
Front End Protocol Internal Reference Manual, publication SM-0042.

NAME

file – Determines file type

SYNOPSIS

file [**–c**] [**–f** *ffile*] [**–m** *mfile*] *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* will print the version stamp, provided it is greater than 0 (see *ld*(1)).

File uses the file **/etc/magic** to identify files that have some sort of *magic number*, that is, any file containing a numeric or string constant that indicates its type. Commentary at the beginning of /etc/magic explains its format.

Options for *file* are:

–f The next argument is taken to be a file containing the names of the files to be examined.

–m Instructs *file* to use an alternate *magic* file.

–c Causes *file* to check the *magic* file for format errors. This validation is not normally carried out for reasons of efficiency. No file typing is done under **–c**.

FILES

/etc/magic

SEE ALSO

ld(1)

NAME

find – Finds files

SYNOPSIS

find *path-name-list* (*expression*)

DESCRIPTION

Find recursively descends the directory hierarchy for each path name in the *path-name-list* (that is, 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 (07777, see *stat*(2)) become significant and the flags are compared.
–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 (named pipe), or regular 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 file **/etc/passwd**, 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 file **/etc/group**, it is taken as a group ID.
–size *n*[**c**]	True if the file is *n* blocks long (512 bytes per block). If *n* is followed by a **c**, the size is in characters.
–atime *n*	True if the file has been accessed in *n* days. The access time of directories in *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 0 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* (4F) ascii format (5120-byte records). *Device* can be a file.
–newer *file*	True if the current file has been modified more recently than the argument *file*.
–depth	Always true; causes descent of the directory hierarchy to be done so that all entries in a directory are acted on before the directory itself. This can be useful when *find* is used with *cpio*(1) to transfer files that are contained in directories without write permission.
(*expression*)	True if the parenthesized expression is true (parentheses are special to the shell and must be escaped).

The primaries may be combined using the following operators (in order of decreasing precedence):

1. The negation of a primary (! is the unary *not* operator).
2. Concatenation of primaries (the *and* operation is implied by the juxtaposition of two primaries).
3. Alternation of primaries (**–o** is the *or* operator).

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 {} \;
```

FILES

/etc/group
/etc/passwd

SEE ALSO

chmod(1), cpio(1), sh(1), test(1)
stat(2) in the UNICOS System Calls Reference Manual, publication SR-2012
cpio(4F), fs(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

finger – Provides user information

SYNOPSIS

finger [*options*] *name* ...
finger *@host*

DESCRIPTION

By default, *finger* lists the login name, full name, terminal name, and write status (as an asterisk (*) before the terminal name if write permission is denied), idle time, login time, and office location and phone number (if they are known) for each current user. (Idle time is represented in minutes if it is a single integer, hours and minutes if a colon (:) is present, or days and hours if a "d" is present in the date field.)

The *finger* command accepts the following options:

–b	Briefer long form list of users
–f	Suppress heading in the short and quick output format
–h	Suppress printing of the *.project* file
–i	Same as quick list but includes idle time
–l	Force long output format
–m	Match arguments only on user name
–p	Suppress printing of the *.plan* file
–q	Quick list with only login name, terminal name and login time
–s	Short list of users
–w	Suppress printing of the full name in the short list format

A longer list format also exists; .I finger uses it whenever a list of names is given. (Account names as well as first and last names of users are accepted.) This format is multi-line, and includes all the information described above as well as the user's home directory and login shell, any plan that the person has placed in the file *.plan* in their home directory, and the project on which they are working from the file *.project* (note that text must begin on line 1 of the file or /fIfinger/fR will not read it) also in the home directory.

The *finger* command can also be used as a network protocol by specifying a hostname or address or a specific user at a host. The *finger* command displays information about a user or users on that host. Instead of the username, any of the above options can be specified at a host.

FILES

/etc/utmp	Who file
/etc/passwd	For user names, phone numbers, ...
~/.plan	Plan file
~/.project	Projecct file

NAME

flodump – Displays flowtrace data in 132-column format

SYNOPSIS

flodump [**–rATC**] [*flowdata_file*]

DESCRIPTION

The *flodump* command interprets the *flowdata_file* file that is produced by executing a program with flowtrace turned on. If you do not specify a *flowdata_file* file, *flodump* uses **flow.data** by default (if you want to specify a different default file, you can set the environment variable **FLOWDATA** to the name of the file to which the data should be written). For each function that is flowtraced, the following statistics are kept: name, entry point, number of times called, who called it and how many times (parents), who it called and how many times (children), time spent in the function exclusive of children, and time spent including children. In addition, *flodump* records the time spent executing FLOWENTER and FLOWEXIT code.

The *flodump* command prints the output in the same format as that provided under COS. It expects a 132-column display that understands form control in column 1.

The following options are available:

–r Reverses the order of the sort

–A Sorts alphabetically; the default.

–T Sorts by time spent in function

–C Sorts by time spent in function and children

Enable the *flodump* command by supplying the flowtrace option at compile time. Use the **–ef** option with *cft*(1) and the **–F** option with *cc*(1).

SEE ALSO

cc(1), cft(1), flow(1)

NAME

flodump – Displays flowtrace data from a file named **core**

SYNOPSIS

flodump

DESCRIPTION

Flodump automatically prepares a flow trace report, on standard output, from a file named core (which must be in the current directory) produced by abnormal program termination. *Flodump* only produces the report under the following conditions:

- If your program was compiled with the flow trace compiler option turned on
- If the report was generated with the same version of the flow trace subroutines as were the programs in the **core** dump file

The report will be less accurate than if the programs terminated normally. *Flodump* assumes that the program termination occurred at the last subprogram entry or exit it processed. The distortion introduced by this assumption can occasionally produce bizarre results, such as apparently negative runtimes.

FILES

core	Program memory dump

BUGS

Support for multitasked programs is not yet available.

SEE ALSO

cft(1), debug(1), pascal(1)

NAME

flow – Displays flowtrace data in 80-column format

SYNOPSIS

flow [**–rATCpctis**] [*flowdata_file*]

DESCRIPTION

The *flow* command interprets the specified *flowdata_file* file that is produced by executing a program with flowtrace turned on. The data from the flowtrace is written in the **flow.data** file by default; however, you can specify a *flowdata_file* on the command. You can also override the default file name by setting the environment variable **FLOWDATA** to the name of the file to which the data should be written.

For each function that is flowtraced, the following statistics are kept: name, entry point, number of times called, who called it and how many times (parents), who it called and how many times (children), time spent in the function exclusive of children, and time spent including children. In addition, flowtrace records the time spent executing FLOWENTER and FLOWEXIT code.

The *flow* command prints the output so it may be viewed on an 80-column screen. The *flow* command also provides information about its children that is not provided by *flodump*(1).

The following options are available:

–r	Reverses the order of the sort
–A	Sorts alphabetically
–T	Sorts by time spent in function; the default.
–C	Sorts by time spent in function and children
–p	Does not print parents
–c	Does not print children
–t	Does not print call tree
–i	Does not print data about functions
–s	Does not print totals

Enable the *flow* command by supplying the flowtrace option at compile time. Use the **–ef** option with *cft*(1) and the **–F** option with *cc*(1).

SEE ALSO

cc(1), cft(1), flodump(1)

NAME

fold – Folds long lines of files for finite width output device

SYNOPSIS

fold [*–width*] [*file* ...]

DESCRIPTION

The *fold* command acts as a filter that folds the contents of the specified files, or the standard input (if you do not specify files), breaking the lines to have maximum width *width*. The default for *width* is 80. If tabs are present, or if they need to be expanded (using *expand*(1) before coming to **fold**), the *width* should be a multiple of 8.

BUGS

If underlining is present, it may be messed up by the *fold* command.

SEE ALSO

expand(1)

NAME

fsplit – Splits Fortran files

SYNOPSIS

fsplit *options files*

DESCRIPTION

Fsplit splits the named *files* into separate files, with one procedure per file. A procedure includes the following program segments: **blockdata**, **function**, **main**, **program**, and **subroutine**. Procedure *X* is put in file *X*.**f**, *X*.**r**, or *X*.**e** depending on the language option chosen, with the following exceptions: *main* is put in the file *MAIN*.**[efr]** and unnamed **blockdata** segments in the files *blockdataN*.**[efr]** where *N* is a unique integer value for each file.

The following *options* are available:

–f Input files are *f77* (default)

–r Input files are *ratfor*

–e Input files are *efl*

–s Strip *f77* input lines to 72 or fewer characters with trailing blanks removed

LIMITATIONS

Ratfor and *efl* are not supported on UNICOS.
F77 is supported on UNICOS running on a CRAY X-MP or CRAY-1 Computer System. However, the *f77* option is useful for any Fortran program.

SEE ALSO

csplit(1), split(1)

NAME

ftp – Transfers files to and from a remote retwork site

SYNOPSIS

ftp [**–v**] [**–d**] [**–i**] [**–n**] [**–g**] [*host*]

DESCRIPTION

Ftp is the user interface to the ARPANET standard File Transfer Protocol (FTP). The program allows a user to transfer files to and from a remote network site.

The client host with which *ftp* is to communicate may be specified on the command line. If this is done, *ftp* will immediately attempt to establish a connection to an FTP server on that host; otherwise, *ftp* will enter its command interpreter and await instructions from the user. When *ftp* is awaiting commands from the user the prompt **ftp>** is provided the user. If insufficient command arguments are supplied by the user, *ftp* will prompt for them.

Ftp may be interrupted, typically by striking the DELETE key or control-C key. If this is done while *ftp* is attempting to carry out a command, *ftp* will revert to the command interpreter and display the "ftp>" prompt. Otherwise, *ftp* will be terminated.

Options may be specified at the command line or to the command interpreter.

The –v (verbose on) option forces *ftp* to show all responses from the remote server, as well as report on data transfer statistics.

The **–n** option restrains *ftp* from attempting autologin upon initial connection. If auto-login is enabled, *ftp* will check the *.netrc(4F)* file in the user's home directory for an entry describing an account on the remote machine. If no entry exists, *ftp* will use the login name on the local machine as the user identity on the remote machine, and prompt for a password and, optionally, an account with which to login.

The –i option turns off interactive prompting during multiple file transfers.

The **–d** option enables debugging.

The **–g** option disables file name globbing.

The following commands are recognized by *ftp*. They may be given aliases as long as they remain unique.

! Invoke a shell on the local machine.

append *local-file* [*remote-file*]
: Append *local-file* to a file on the remote machine. If *remote-file* is left unspecified, the name of *local-file* is used in naming *remote-file*. File transfer uses the current settings for *type*, *format*, *mode*, and *structure*.

ascii Set the file transfer *type* to network ASCII. This is the default *type*.

bell Sound a bell after each file transfer command is completed.

binary Set the file transfer *type* to support *binary* image transfer.

bye Terminate the FTP session with the remote server and exit *ftp*.

cd *remote-directory*
: Change the working directory on the remote machine to *remote-directory*.

close Terminate the FTP session with the remote server and return to the command interpreter.

delete *remote-file*
: Delete the file *remote-file* on the remote machine.

debug Toggle debugging mode. When debugging is on, *ftp* prints each command sent to the remote machine, preceded by the string -->. For a list of the commands, see *ftpd(8)*.

dir [*remote-directory*] [*local-file*]
Print a listing of the contents of *remote-directory* and, optionally, place the output in *local-file*. If no directory is specified, the current working directory on the remote machine is used. If *local-file* is not specified, output comes to the terminal.

form *format*
Set the file transfer *form* to *format*. The default format is "non-print". At this printing, only the default form is supported.

get *remote-file* [*local-file*]
Retrieve *remote-file* and store it on the local machine. If the name of *local-file* is not specified, it is given the same name it has on the remote machine. The current settings for *type*, *form*, *mode*, and *structure* are used while transferring the file.

hash Toggle hash-sign (``#'') printing for each data block transferred. The size of a data block is 4096 bytes.

glob Toggle local file name globbing. With file name globbing enabled, each local file or path name is processed for the *sh(1)* metacharacters *, ?, [, An additional pair of metacharacters, {}, is also processed. This pair may enclose several comma-separated strings for each of which a match is sought. With globbing disabled all local files and path names are treated literally. Globbing is always on with reference to remote files.

help [*command*]
Print an informative message about the meaning of *command*. If no argument is given, *ftp* prints a list of the known commands.

lcd [*directory*]
Change the working directory on the local machine. If *directory* is not specified, the user's home directory is used.

ls [*remote-directory*] [*local-file*]
Print an abbreviated listing of the contents of a directory on the remote machine. If *remote-directory* is left unspecified, the current working directory is used. If *local-file* is not specified, the output is sent to the terminal.

mdelete *remote-files*
Delete the specified files on the remote machine. If globbing is enabled, the specification of *remote-files* will first be expanded using *ls*.

mdir *remote-files local-file*
Obtain a directory listing of multiple files on the remote machine and place the result in *local-file*.

mget *remote-files*
Retrieve the specified files from the remote machine and place them in the current local directory. If globbing is enabled, the specification of remote files will first be expanded using *ls*.

mkdir *directory-name*
Make a directory on the remote machine.

mls *remote-files local-file*
Obtain an abbreviated listing of multiple files on the remote machine and place the result in *local-file*.

mode [*mode-name*]
Set the file transfer *mode* to *mode-name*. The default mode is stream mode. At this printing, only the default is supported.

mput *local-files*
: Transfer multiple *local-files* from the current local directory to the current working directory on the remote machine.

open *host* [*port*]
: Establish a connection to the specified *host* FTP server. An optional *port* number may be supplied, in which case, *ftp* will attempt to contact an FTP server at that port. If autologin is enabled (default), *ftp* will also attempt to automatically log the user in to the FTP server (see below).

prompt Toggle interactive prompting. Interactive prompting occurs during multiple file transfers to allow the user to selectively retrieve or store files. If prompting is turned off (default), a *mget* or *mput* command will transfer all files.

put *local-file* [*remote-file*]
: Store *local-file* on the remote machine. If *remote-file* is left unspecified, the name of *local-file* is used in naming *remote-file*. File transfer uses the current settings for *type*, *format*, *mode*, and *structure*.

pwd Print the name of the current working directory on the remote machine.

quit A synonym for bye.

quote *arg1 arg2 ...*
: The arguments specified are sent, verbatim, to the remote FTP server. A single FTP reply code is expected in return.

recv *remote-file* [*local-file*]
: A synonym for **get**.

remotehelp [*command-name*]
: Request help from the remote FTP server. If *command-name* is specified it is supplied to the server as well.

rename [*from*] [*to*]
: Rename the file *from* on the remote machine to the file *to*.

rmdir *directory-name*
: Delete a directory on the remote machine.

send *local-file* [*remote-file*]
: A synonym for **put**.

sendport
: Toggle the use of PORT commands. By default, *ftp* will attempt to use a PORT command when establishing a connection for each data transfer. If the PORT command fails, *ftp* will use the default data port. When the use of PORT commands is disabled, no attempt will be made to use PORT commands for each data transfer. This is useful for certain FTP implementations which do ignore PORT commands but, incorrectly, indicate they've been accepted.

status Show the current status of *ftp*.

struct [*struct-name*]
: Set the file transfer *structure* to *struct-name*. By default, *file* structure is used. At this printing, only the default is supported.

tenex Set the file transfer *type* to that needed to talk to TENEX machines.

trace Toggle packet tracing.

type [*type-name*]
: Set the file transfer *type* to *type-name*. If *type* is not specified, the current type is printed. The three valid types are "tenex", "binary" and "ascii", which is the default.

user *user-name* [*password*] [*account*]
Identify yourself to the remote FTP server. If *password* is not specified and the server requires it, *ftp* will prompt you for it (after disabling local echo). If *account* is not specified, and the FTP server requires it, you will be prompted for it. Unless *ftp* is invoked with autologin disabled, this process is done automatically on initial connection to the FTP server.

verbose Toggle *verbose* mode. In *verbose* mode, all responses from the FTP server are displayed to the user. In addition, if *verbose* is on, when a file transfer completes, statistics regarding the efficiency of the transfer are reported. By default, *verbose* is off.

? [*command*]
A synonym for help.

Command arguments which have embedded spaces may be quoted with quote (") marks.

File Naming Conventions

Files specified as arguments to *ftp* commands are processed according to the following rules.

1. If the file name **–** is specified, *stdin* (for reading) or *stdout* (for writing) is used.
2. If the first character of the file name is **|**, the remainder of the argument is interpreted as a shell command. *Ftp* then forks a shell with the argument supplied, and reads (or writes) from *stdout* (or *stdin*). If the shell command includes spaces, the argument must be quoted; for example, **"| ls -lt"**. A particularly useful example of this mechanism is **dir < directory name> |pg**.

3) Failing the above checks, if "globbing" is enabled, local file names are expanded as per the *glob* command.

BUGS

Many FTP server implementations do not support operations such as "print working directory".

The *mget* and *mdelete* commands should be used with caution. Specifying a directory where a plain file name is expected could produce unexpected results.

SEE ALSO

ftpusers(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

ftref – Generates Fortran reference listing

SYNOPSIS

ftref [**–c***cb*][**–t***tree*][**–r***root*][**–e***end*][**–dnm**] *filename*

DESCRIPTION

The *ftref* command generates a listing that contains several forms of information about a Fortram program. The *ftref* command reports on the common block variables used in the subroutines within the program. It provides tabular information that consists of entry names, calling routines, and called routines for each subroutine. This information is displayed as a static calling tree. For multitasked programs, *ftref* summarizes the use of multitasking subroutines and reports whether a common variable or a subroutine is locked when it is referenced or redefined.

The *ftref* command requires the output produced when **–e** is specified in a previous CFT statement. The dataset to be processed by *ftref* may contain any number of modules used by the program. The more program modules included in the dataset, the more complete the information output by *ftref*.

The following arguments are available:

–c*cb* Global common block cross references. The default is *part.* The *cb* argument can use the following routines:

part Identifies the routine names using a common block

full Details the use of the variables of a common block in a routine

none No output information

–t*tree* Produces information about the routines called and the static calling tree for the program. The value '**LOOP**' indicates that an apparently recursive program exists. The *tree* argument can use the following routines:

part Reports entry names, external calls, other routines that call the routine, and common block names from the input dataset; the default.

full Reports the information that the *part* option provides plus the static calling tree

none No output information

–r*root* If you specified **–t***full,* the **–r** directs *root* to be the root of the tree. The **–r** option can be used to get a subtree for the program; it can also be used to request multiple subtrees, each beginning at a different root. If you do not specify the **–r** argument, a routine that has not been called by another routine is chosen by default. If there is more than one uncalled routine, the first routine (by alphabetic order) is chosen as the root.

–e*end* If you specified **–t***full,* the **–e** argument directs routine *end* to terminate any branch of the tree in which *end* is encountered. The value '**STOP**' is printed whenever the routine is found, and that branch of the tree is terminated. By default, *ftref* generates a tree containing all subroutines in the program.

–l*n* If you specified **–t***full,* the **–l** argument indicates that the maximum length of any branch is *n* levels deep. The default is the entire program. If both **–l** and **–e** are specified, *ftref* terminates a branch of the tree at which ever state is encountered first.

–d Selects modules to process or common blocks for *ftref* to check to determine whether a variable is in a locked area. The standard input contains a set of directives that controls the processing or check. The directives are taken from standard input. The default is no directives to be read.

–n Lists the subroutines in input order instead of alphabetic order. The default is alphabetic order.

–m Examines the source for uses of the multitasking subroutines and generates tables summarizing the subroutine's use within the program. Refer to the CRAY X-MP Multitasking Programmer's Manual for more information.

NAME

get – Gets a version of an SCCS file

SYNOPSIS

get [**–r***SID*] [**–c***cutoff*] [**–i***list*] [**–x***list*] [**–w***string*] [**–a***seq-no*] [**–k**] [**–e**] [**–l**[**p**]] [**–p**] [**–m**] [**–n**] [**–s**] [**–b**] [**–g**] [**–t**] *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 a dash (–). The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, *get* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with s.) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The generated text is normally written into a file called the *g-file* whose name is derived from the SCCS file name by simply removing the leading s.; (see also *FILES*, below).

Each of the keyletter arguments is explained below as though only one SCCS file is to be processed, but the effects of any keyletter argument applies independently to each named file.

–r*SID* The *S*CCS *ID*entification string (SID) of the version (delta) of an SCCS file to be retrieved. Table 1 below shows, for the most useful cases, what version of an SCCS file is retrieved (as well as the SID of the version to be eventually created by *delta*(1) if the **–e** keyletter is also used), as a function of the SID specified.

–c*cutoff* *Cutoff* date-time, in the form:

YY[*MM*[*DD*[*HH*[*MM*[*SS*]]]]]

No changes (deltas) to the SCCS file which were created after the specified *cutoff* date-time are included in the generated ASCII text file. Units omitted from the date-time default to their maximum possible values; that is, **–c7502** is equivalent to **–c750228235959**. Any number of non-numeric characters may separate the various 2-digit pieces of the *cutoff* date-time. This feature allows one to specify a *cutoff* date in the form: "**–c77/2/2 9:22:25**". Note that this implies that one may use the %E% and %U% identification keywords (see below) for nested *gets* within, say the input to a *send*(1C) 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 *gets* 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*.

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

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

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

–n Causes each generated text line to be preceded with the %M% identification keyword value (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 (such as, **–r1**), or release and level (such as, **–r1.2**).

–w *string* Substitute *string* for all occurrences of "**% w%**" when using the *get* command on a 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	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 (that is, maximum branch number plus one) of level L within release R". Note that if the SID specified is of the form "R.L", "R.L.B", or "R.L.B.S", each of the specified components *must* exist.

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

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

# Successor.

† The **–b** 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 CX-OS system program files. %W% = %Z%%M%<horizontal-tab>%I%
%A%	Another shorthand notation for constructing *what*(1) strings for non-CX-OS 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 process ID of the command (that is, *get*) that created it. The *z-file* is created in the directory containing the SCCS file for the duration of *get*. The same protection restrictions as those for the *p-file* apply for the *z-file*. The *z-file* is created mode 444.

MESSAGES

Use *help*(1) for explanations.

BUGS

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

SEE ALSO

admin(1), delta(1), help(1), prs(1), what(1)
sccsfile(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

getopt – Parses command options

SYNOPSIS

set — getopt *optstring* **$***

DESCRIPTION

Getopt breaks up options in command lines for easy parsing by shell procedures and checks 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 of the shell ($1 $2 ...) 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.

EXAMPLE

The following shell procedure 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 –ao*arg file file*
cmd –a –o *arg file file*
cmd –o*arg* –a *file file*
cmd –a –o*arg* — *file file*

MESSAGES

Getopt prints an error message on the standard error when it encounters an option letter not included in *optstring*.

SEE ALSO

sh(1)
getopt(3C) in the CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual, publication SR-2013

NAME

grep, egrep, fgrep – Searches a file for a pattern

SYNOPSIS

grep [*options*] *expression* [*files*]

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); *grep* uses a compact non-deterministic algorithm. *Egrep* patterns are full regular *expressions*; *egrep* uses a fast deterministic algorithm that sometimes needs exponential space. *Fgrep* patterns are fixed *strings*. 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-line characters.
–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 *file*.

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

Fgrep searches for lines that contain one of the *strings* separated by new-lines.

Egrep accepts regular expressions as in *ed*(1), except for **\\(** and **\\)**, with the addition of the following:

- A regular expression followed by **+** matches one or more occurrences of the regular expression
- A regular expression followed by **?** matches zero or one occurrences of the regular expression
- Two regular expressions separated by **|** or by a new-line match strings that are matched by either
- A regular expression may be enclosed in parentheses **()** for grouping

The order of precedence of operators is **[ ]**, then *** ? +**, then concatenation, then **|** and new-line.

NOTES

Contrary to expectation, *fgrep* is the slowest of the three commands. *Grep* is the fastest.

MESSAGES

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 there is not a single algorithm that spans a wide enough range of space-time tradeoffs.

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

Egrep does not recognize ranges, such as **[a–z]**, in character classes.

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

SEE ALSO

ed(1), sed(1), sh(1)

NAME

head – Prints the first few lines of a file

SYNOPSIS

head [– *count*] [*file* ...]

DESCRIPTION

The *head* command writes the first *count* lines of each of the specified files, or of the standard input if you do not specify any files. If you omit *count*, *head* defaults to 10.

SEE ALSO

tail(1)

NAME

help – Provides explanation of messages and commands

SYNOPSIS

help [*arg* ...]

DESCRIPTION

The *help* command 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 (for example, **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 (for example, **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**.

SEE ALSO

man(1)

NAME

hostid – Sets or prints identifier of current host system

SYNOPSIS

hostid [*identifier*]

DESCRIPTION

The *hostid* command sets or prints the *identifier* of the current host in hexadecimal. By default, the value of *identifier* is 0. This numeric value is expected to be unique across all hosts and is normally set to the host's Internet address. The super user can set the *identifier* by giving a hexadecimal argument; this is usually done in the startup script /etc/rc2*.net*.

SEE ALSO

gethostid(3W), sethostid(3W) in the CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual, publication SR-2013

NAME

hostname – Prints the name of current host system

SYNOPSIS

hostname [*nameofhost*]

DESCRIPTION

The *hostname* command prints the name of the current host. A user with super user privileges can set the host name by giving an argument to *hostname*.

SEE ALSO

gethostname(2), sethostname(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

id – Prints user and group IDs

SYNOPSIS

id

DESCRIPTION

The *id* command 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.

FILES

/etc/passwd
/etc/group

SEE ALSO

logname(1)
getuid(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

jad – Job accounting daemon

SYNOPSIS

jad

DESCRIPTION

The *jad* daemon collects selected accounting records from the system accounting file (/usr/adm/pacct) and writes them to a job/session–related accounting file. The selection is done upon the *job* ID of the calling process. Data collection is done in the background.

The *jad* daemon creates three files in the users HOME directory. Their names are generated using the *job* ID, thus they are unique. The *jad* command creates the following files :

.<job id>.prot: job accounting file
.<job id>.prec: *jad* input pipe (for synchronization)
.<job id>.psnd: *jad* output pipe (for synchronization)

Upon normal termination (**SIGTERM** or **SIGHUP**) *jad* removes all its files. However, files may not get deleted immediately upon non-normal termination of *jad*.

During initialization *jad* removes old *jad* files (see above) in the HOME directory of the calling user.

BUGS

If *jad* is started from a batch process, it is not notified about the termination of the batch job, thus it is the user's responsibility to terminate *jad* (see *jar*(1)).

Until the *job* ID concept has been implemented in UNICOS, *jad* will use the *process group* ID instead of the *job* ID.

SEE ALSO

jar(1), kill(1), HOME variable in sh(1)
signal(2)

NAME

jar – Provides job accounting information

SYNOPSIS

jar [**–acdfhlmst**] [**–I** *file* [**–G** *gid*] [**–U** *uid*] [**–J** *job id*]]

DESCRIPTION

The *jar* command provides information about job/session-related accounting information. Upon normal operation (–I not specified) input is taken from the job accounting file provided by *jad*(1). Due to the asynchronous operation of the *jad* command, a synchronization has to take place to ensure that the job/session accounting file holds current information. If you specify **–I** , *jar* takes input from the given file and no synchronization with *jad(1)* is performed.

Ther *jar* command can produce three kinds of reports by specifying different options (described below):

–a	Same as specifying **–chs** together.
–c	Produces the *command statistics* report.
–d	Provides information about device-specific I/0 (if available); this results in multiple lines per command, if **–c** is set also.
–f	Produces the *process flow chart* report
–h	Precedes each report with a header.
–l	Provides addition information for the *command statistics* and the *process flow chart* reports.
–m	If you specify this in addition to **–c**, the output includes a user CPU time breakdown for 1, 2, 3, and 4 that are running parallel.
–s	Produces the *summary statistics* report. This is default (with a header) if you do not specify a report type.
–t	Sends a termination request to *jad*. Other options are ignored and *jad* does not produce a report.
–I *file*	Takes input from the given *file* and performs no synchronization with *jad*. If you specify this option, see the **–GUJ** options.
–G *gid*	Provides the group ID.
–U *uid*	Provides the user ID.
–J *job id*	Provides th job ID. During normal operation **–GUJ** are ignored.

The following information may be output from *jar*:

Command statistics per process (–c):

Command name (first 8 characters)
Starting time as *hh:mm:ss*
CPU time spent in user mode in seconds (4 fields if –m)
CPU time spent in system mode in seconds
I/O wait time in seconds
Elapsed time in seconds
Average execution-memory size in MegaWords (–l only)
Average I/O-wait-memory size in MegaWords (–l only)
No. of MegaBytes transfered (–l only)
No. of logical I/O requests (–l only)

No. of real I/O requests (–l only)
Exit status (–l only)
Nice value (–l only)
Accounting Flags (–l only)
System Billing Units (SBUs)
Device I/O breakdown (–d only); one line per major device number; each line gives the no. of bytes transfered and the the number of logical I/O requests.

Process Flow Chart (–f)

Relations between processes
SBUs (–l only)

Summary statistics (–s):

Input file name (–I only)
Selective *gid* (–G only)
Selective *uid* (–U only)
System identification
Job name (from NQS–supplied SUBMIT_REQNAME environment variable)
User ID
Group ID
Accounting ID Name
Job ID
Least recent command starting time
Most recent command ending time
Accumulated CPU time in user mode
Accumulated CPU time in system mode
Accumulated CPU time for 1,2,3 and 4 CPUs (for multitasked processes)
Average CPU usage (for multitasked processes)
Accumulated I/O wait time
Total elapsed time
CPU time memory integral
I/O wait time memory integral
Accumulated MegaBytes transferred
Accumulated logical I/O requests
Accumulated real I/O requests
Number of commands
System Billing Units

BUGS

Until the *job id* concept has been implemented in UNICOS, *jad* will use the *process group id* instead of the *job id.*

SEE ALSO

acctcom(1),jad(1),sh(1)

NAME

join – Joins specified lines of files

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 joins 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
```

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*, and *uniq* are wildly incongruous.

File names that are numeric may cause conflict when the **-o** option is used right before listing file names.

SEE ALSO

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

NAME

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

The process to be killed must belong to the current user; the super user can kill any process.

If a signal number preceded by – is given as first argument, that signal is sent instead of the terminate signal (see *signal*(2)). In particular, "kill –9 *PID*" is a kill signal that cannot be caught or ignored.

NOTE

The csh(1) command has a built-in kill command with slightly different characteristics. See csh(1).

SEE ALSO

ps(1), sh(1)
kill(2), signal(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

last – Indicates the last logins of users and teletypes

SYNOPSIS

last [*num*] [*name* ...] [*tty* ...]

DESCRIPTION

The *last* command looks in the **wtmp** file, which records all logins and logouts, for information about a user, a teletype, or any group of users and teletypes. Arguments specify names of users or teletypes of interest. Names of teletypes can be given fully or abbreviated. For example, *last 0* is the same as *last tty0*. If you specify multiple arguments, *last* prints the information applying to any of the arguments. The *last* command prints the sessions of the specified users and teletypes, most recent first, indicating the times at which the session began, the duration of the session, and the teletype on which the session took place. If the session is still continuing or was cut short by a reboot, *last* so indicates.

If you specify *num*, the *last* command limits the report to *N* lines.

The pseudo-user **reboot** logs in at reboots of the system, thus

```
last reboot
```

gives an indication of mean time between reboots.

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.

EXAMPLES

To list all of "root's" sessions as well as all sessions on the console terminal:

```
last root console
```

To print a record of all logins and logouts in (in reverse order), use

```
last
```

with no arguments.

FILES

/etc/wtmp	Login data base

SEE ALSO

wtmp(4F)

NAME

lastcomm – Shows last commands executed in reverse order

SYNOPSIS

lastcomm [*command name*] ... [*user name*] ... [*terminal name*] ...

DESCRIPTION

The *lastcomm* command gives information on previously executed commands. With no arguments, *lastcomm* prints information about all the commands recorded during the current accounting file's lifetime. If called with arguments, only accounting entries with a matching command name, user name, or terminal name are printed. So, for example,

```
lastcomm a.out root tty00
```

would produce a listing of all the executions of commands named *a.out* by user *root* on the terminal *tty00*, and

```
lastcomm root
```

would produce a listing of all the commands executed by user *root*.

For each process entry, the following are printed (in the order given).

The command name under which the process was called
The name of the user who ran the process
Flags, as accumulated by the accounting facilitites in the system
The tty name from which the command was executed
The amount of cpu time used by the process (in seconds)
The time the process exited

The flags are encoded as follows: ''S'' indicates the command was executed by the super-user, ''F'' indicates the command ran after a fork, but without a following *exec*, and ''M'' indicates additional accounting records were written for this process.

SEE ALSO

last(1)
acct(4F), core(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

ld, fpld – Invokes the link editor for relocatable files

SYNOPSIS

ld [*options*] [*files*] [*options with files*]
fpld [*options*] [*files*] [*options with files*]

DESCRIPTION

The *ld* or *fpld* commands combine several relocatable files into one executable file. The difference between their output *files* is that with *fpld*, the output *file* is intended for execution by the foreground processor. Specifically, the foreground processor has smaller memory sizes, initialized data in a separate memory, and has separated code and data areas. The services performed include relocating code and data blocks, resolving external references, and writing symbol information for debugging.

With the exception of the **–q** option, all input files, libraries and options are processed in the order listed. The distinction between hard and soft external references (see *relo*(4F)) is supported; a module is loaded from a library only if it resolves an outstanding hard external reference.

There are six character strings built into *ld*. First, **"$start"** is the default starting point for the resulting executable. Second, any entry point with the primary entry bit set (see *relo*(4F)) will be entered twice; once with the specified entry name and once with the current primary entry name, which defaults to **main**. The remaining four generate entry points that permit the program and the debugger to locate various areas within the program. These entry points are as follows:

0 thru the value of "zzzzzztx"	Program text
"zzzzzztx" thru "zzzzzzdt"	Initialized data blocks
"zzzzzzdt" thru "zzzzzzcm"	Uninitialized data blocks
0 thru the value of "zzzzzzlm"	All allocations of local memory

Because zzzzzz symbols are added after the libraries have been searched, any user attempt to define one of them fails.

The following options are recognized by *ld* family.

–a Calls the module align subroutine to cause the local code blocks from the next module to be loaded on an instruction buffer boundary.

–A Toggles the module align flag. If the flag is one, the module align subroutine is called before each relocatable module is processed.

–d *x* Uses the character string *x* to define a global symbol. The following four formats are possible.

–d =*n2*	The name *n2* is entered as the second name for all symbols with the primary entry bit set.
–d *n1*	The name *n1* is entered as defined variable with value zero.
–d *n1=nnn*	The name *n1* is entered as a defined variable with integer value *nnn*.
–d *n1=n2*	The name *n1* is entered as a variable aliased to *n2*.

–e *name* Sets the entry point address for the output file to be the address of the symbol name.

-g	Produces a global variable for each local block in each module. For local block "xxx" in module "yyy", the name of the global variable is "#yyy#xxx". The size of the block is in the **pdtecl** field of the **nlis_h** structure where it may be accessed by future debuggers or by the namelist library routine.
-l*x*	Identifies a library to be conditionally loaded. Symbol *x* represents a character string. If *x* begins with a . or /, the library is the file named *x*. If *x* begins with any other character, the library **/lib/libx.a** is searched for the library file. If it is not found, the library **/usr/lib/libx.a** is then searched for the file.
-L*x*	Identifies a library to be conditionally loaded. The processing is as detailed above except that the library is scanned repeatedly until no further modules in the library resolve currently outstanding hard externals.
-m	Toggles the map request flag. This flag is off at the beginning of pass 1 and pass 2. Placing a single **-m** as the first option gives a complete map listing on **stdout**. Placing a single **-m** as the last option gives only the symbol listing. Using repeated **-m** options can produce partial maps.
-o *fn*	Using *fn* as the name of the output file overrides the default of **a.out**.
-q *fn*	Identifies an initial definition file. This file will be processed first. It may be used to supply base addresses and global locations for the load step.
-r *fn*	Identifies a file name to contain a summary of the load. The **-r** file from one *ld* run may be used as the **-q** file for the next.
-s	Indicates that the debug symbol information is to be stripped from the end of the output module.
-u *nl*	Enters name *nl* in the symbol table as an outstanding hard external.
-6	Fortran programs can initialize common blocks more than once and can use common block names that are the same as subroutine names. Both of these are direct violations of the Fortran 77 standard. However, if multiple initializations are permitted for common blocks, no common blocks will be loaded into the *bss* space. Thus, it is important to have the **-6** option off if disk space is important. The **-6** option is implemented to permit Fortran 66 programs to run.

Argument names without a preceding dash character are taken to be names of input files to be included unconditionally.

SEE ALSO

as(1), cc(1)
relo(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

ld – Invokes the CRAY X-MP and CRAY-1 link editor

SYNOPSIS

ld [**–o***outfile*] [**-L** *dir*] [**-l***x*] [**-m**] *files* ...

DESCRIPTION

Ld combines several object programs into one, resolves external references, and searches libraries. The object file names must end in **.o**. In the simplest case, several object files are given; *ld* combines them, producing an executable program.

Segldr(1) is used to link the object files into one executable program. A load map summary is produced in a file whose name has **.map** appended to *outfile*.

The options to *ld* are as follows:

–o*outfile*
Override the default output file name, *a.out*, to be *outfile*.

–L *dir* Change the algorithm of searching for **lib***x***.a** to look in the directory *dir* before looking in **/lib** and **/usr/lib**. This option is only effective if it preceeds the **–l** option on the command line.

–l*x* Search a library **lib***x***.o**; *x* is up to nine characters. A library is searched when its name is encountered, so the placement of the **–l** option is significant. By default, libraries are located in **/lib** and **/usr/lib**.

–m Produce a load map in the file *outfile***.map** (**a.out.map** by default)

FILES

file.o	Input object file
a.out	Executable output file
/lib/fortlib.o	FORTRAN library
/lib/libp.o	Pascal library
/lib/libc.o	C library
/lib/libf.o	FORTRAN library
/lib/libm.o	FORTRAN math library
/lib/libu.o	FORTRAN utility library
/lib/libio.o	FORTRAN I/O library

MESSAGES

The error messages produced by *segldr* are self-explanatory. If further explanation is required, refer to the Segment Loader (SEGLDR) Reference Manual, publication SR-0066.

SEE ALSO

as(1), cc(1), cft(1), segldr(1)

NAME

lex – Generates programs for simple lexical tasks

SYNOPSIS

lex [**–rctvn**] [*file*] ...

DESCRIPTION

Lex generates programs to be used in simple lexical analysis of text.

The input *files* (standard input by default) contain strings and expressions to be searched for, and C text to be executed when strings are found.

A file **lex.yy.c** is generated which, when loaded with the library, copies the input to the output except when a string specified in the file is found; then the corresponding program text is executed. The actual string matched is left in *yytext*, an external character array. Matching is done in order of the strings in the file. The strings may contain square brackets to indicate character classes, as in **[abx–z]** to indicate **a**, **b**, **x**, **y**, and **z**; and the operators *, +, and ? mean respectively any non-negative number of, any positive number of, and either zero or one occurrences of, the previous character or character class. The character **.** is the class of all ASCII characters except new-line. Parentheses for grouping and vertical bar for alternation are also supported. The notation *r*{*d*,*e*} in a rule indicates between *d* and *e* instances of regular expression *r*. It has higher precedence than |, but lower than *, ?, +, and concatenation. The character ^ at the beginning of an expression permits a successful match only immediately after a new-line, and the character **$** at the end of an expression requires a trailing new-line. The character / in an expression indicates trailing context; only the part of the expression up to the slash is returned in *yytext*, but the remainder of the expression must follow in the input stream. An operator character may be used as an ordinary symbol if it is within " symbols or preceded by \. Thus [a–zA–Z]+ matches a string of letters.

Three subroutines defined as macros are expected: **input()** to read a character; **unput(***c***)** to replace a character read; and **output(***c***)** to place an output character. They are defined in terms of the standard streams, but you can override them. The program generated is named **yylex()**, and the library contains a **main()** which calls it. The action REJECT on the right side of the rule causes this match to be rejected and the next suitable match executed; the function **yymore()** accumulates additional characters into the same *yytext*; and the function **yyless(***p***)** pushes back the portion of the string matched beginning at *p*, which should be between *yytext* and *yytext+yyleng*. The macros *input* and *output* use files **yyin** and **yyout** to read from and write to, defaulted to **stdin** and **stdout**, respectively.

Any line beginning with a blank is assumed to contain only C text and is copied; if it precedes **%%** it is copied into the external definition area of the **lex.yy.c** file. All rules should follow a **%%**, as in YACC. Lines preceding **%%** which begin with a non-blank character define the string on the left to be the remainder of the line; it can be called out later by surrounding it with **{}**. Note that curly brackets do not imply parentheses; only string substitution is done.

The following options must appear before any files:

–r Indicates RATFOR actions.

–c Indicates C actions and is the default.

–t Causes the **lex.yy.c** program to be written instead to standard output.

–v Provides a one-line summary of statistics of the machine generated.

–n Will not print out the – summary.

Multiple files are treated as a single file. If no files are specified, standard input is used.

Certain table sizes for the resulting finite state machine can be set in the definitions section:

%p *n*	number of positions is *n* (default 2000)
%n *n*	number of states is *n* (500)
%t *n*	number of parse tree nodes is *n* (1000)
%a *n*	number of transitions is *n* (3000)

The use of one or more of the above automatically implies the **–v** option, unless the **–n** option is used.

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;
                                }
                      }
```

The external names generated by *lex* all begin with the prefix **yy** or **YY**.

LIMITATIONS

RATFOR is not supported on Cray systems.

SEE ALSO

yacc(1)
The UNICOS Support Tools Guide, publication SG-2016.

NAME

line – Reads one line

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 character. It is often used within shell files to read from the user's terminal.

SEE ALSO

sh(1)
read(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

lint – Invokes a C program checker

SYNOPSIS

lint [**–abhlnpuvx**] *file* ...

DESCRIPTION

Lint attempts to detect features of the C program *files* that are likely to be bugs, non-portable, or wasteful. It also checks type usage more strictly than the compilers. The following are currently detected: unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, the usage of functions is checked to find functions which return values in some places and not in others, functions called with varying numbers of arguments, and functions whose values are not used.

It is assumed that all the *files* are to be loaded together; they are checked for mutual compatibility. By default, *lint* uses function definitions from the standard lint library **llib-lc.ln**; function definitions from the portable lint library **llib-port.ln** are used when *lint* is invoked with the **–p** option.

Any number of *lint* options may be used, in any order. The following options are used to suppress certain kinds of complaints:

–a	Suppress complaints about assignments of long values to variables that are not long.
–b	Suppress complaints about **break** statements that cannot be reached. (Programs produced by *lex* or *yacc* will often result in a large number of such complaints.)
–h	Do not apply heuristic tests that attempt to intuit 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.

–lx	Include additional lint library **llib-lx.ln**. For example, you can include a lint version of the Math library **llib-lm.ln** by inserting **–lm** on the command line. This argument does not suppress the default use of **llib-lc.ln**. This option can be used to keep local lint libraries and is useful in the development of multi-file projects.
–n	Do not check compatibility against either the standard or the portable lint library.
–p	Attempt to check portability to other dialects (IBM and GCOS) of C.

The **–D**, **–U**, and **–I** options of *cc*(1) are also recognized as separate arguments.

Certain conventional comments in the C source will change the behavior of *lint*:

/*NOTREACHED*/	At appropriate points, stops comments about unreachable code.
/*VARARGS*n**/	Suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0.
/*ARGSUSED*/	Turns on the –v option for the next function.
/*LINTLIBRARY*/	At the beginning of a file, shuts off complaints about unused functions in this file.

Lint produces its first output on a per source file basis. Complaints regarding included files are collected and printed after all source files have been processed. Finally, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a complaint stems from a given source file or from one of its included files, the source file name will be printed followed by a question mark.

FILES

/usr/lib/lint1	Programs
/usr/lib/lint2	Programs
/usr/lib/llib-lc.ln	Declarations for standard functions (binary format; source is in **/usr/lib/llib-lc**)
/usr/lib/llib-port.ln	Declarations for portable functions (binary format; source is in **/usr/lib/llib-port**)
/usr/lib/llib-lm.ln	Declarations for standard Math Library functions (binary format; source is in **/usr/lib/llib-lm**)
/usr/tmp/*lint*	Temporary files

BUGS

Exit(2) and other functions that do not return are not understood; this sometime causes incorrect analysis.

SEE ALSO

cc(1), cpp(1)

NAME

ln – Links files

SYNOPSIS

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

DESCRIPTION

The file *file1* is linked 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 linked to that directory. If *target* is a file, its contents are destroyed.

If *ln* determines that the mode of *target* forbids writing, it will print the mode (see *chmod*(2)), ask for a response, and read the standard input for one line. If the line begins with **y**, the *ln* occurs, if permissible; if not, the command exits. No questions are asked and the *ln* is done when the **–f** option is used or if the standard input is not a terminal.

CAVEAT

The *ln* command does not link across file systems.

SEE ALSO

cp(1), cpio(1), mv(1), rm(1)
chmod(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

login – Signs on

SYNOPSIS

login [*name* [*env-var* ...]]

DESCRIPTION

The *login* command is used at the beginning of each terminal session and lets you identify yourself to the system. It may be invoked as a command or by the system when a connection is first established. Also, it is invoked by the system when a previous user has terminated the initial shell by typing a CONTROL-D to indicate an end-of-file.

If *login* is invoked as a command it must replace the initial command interpreter. This is accomplished by typing:

```
        exec login
```

from the initial shell.

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.

After a successful login, accounting files are updated, the procedure **/etc/profile** is performed, the message-of-the-day, if any, is printed, the user-ID, the group-ID, the working directory, and the command interpreter (usually *sh*(1)) is initialized, and the file **.profile** in the working directory is executed, if it exists. These specifications are found in the **/etc/passwd** file entry for the user. The name of the command interpreter is – followed by the last component of the interpreter's path name (that is, **–sh**). If this field in the password file is empty, then the default command interpreter, */bin/sh*, is used. If this field is *, then a *chroot*(2) is performed 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 **/bin/login** and **/etc/passwd**.

The basic environment (see *sh*(1)) is initialized to:

HOME=*your-login-directory*
PATH=:/bin:/usr/bin
SHELL=*last-field-of-passwd-entry*
MAIL=/usr/mail/*your-login-name*
LOGNAME=*your-login-name*

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. People logging into restricted shell environments are thus prevented from spawning secondary shells that are not restricted.

Login understands simple single-character quoting conventions. Typing a backslash in front of a character quotes it and allows the inclusion of such characters as spaces and tabs.

FILES

/bin/passwd	Program to change passwords
/bin/sh	Standard shell
/dev/tty*	Login devices
/etc/dialups	List of devices that need a dialup password
/etc/d_passwd	Dialup passwords for **/etc/dialups**
/etc/passwd	Password file
/etc/utmp	Accounting file
/etc/wtmp	Accounting file
/usr/mail/$LOGNAME	Mailbox for account $LOGNAME

MESSAGES

Login incorrect	The user name or the password cannot be matched.
No shell or Cannot open password file or No directory	Account may not be set up correctly; consult a CRI site analyst.
No utmp entry.	You must execute *login* from the lowest level *sh.* You attempted to execute *login* as a command without using the shell's *exec* internal command or from other than the initial shell.

SEE ALSO

mail(1), passwd(1), sh(1), su(1)
chroot(1M), getty(1M) in the UNICOS Administrator Commands Reference Manual, publication SR-2022
passwd(4F), profile(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

logname – Gets login name

SYNOPSIS

logname

DESCRIPTION

The *logname* command uses *cuserid*(3S) to find the login name of the user and prints that name on the standard output.

SEE ALSO

env(1), login(1), sh(1)
cuserid(3S) in the CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual, publication SR-2013

NAME

lorder – Finds ordering relation for an object library

SYNOPSIS

lorder *file* ...

DESCRIPTION

The input is one or more object files. The standard output is a list of pairs of object file names. 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). The *lorder*(1) command may allow for a more efficient access of the archive during the link edit process.

EXAMPLES

The following example builds a new library from existing **.o** files.

```
ar cr library `lorder *.o | tsort`
```

FILES

*symdef	Temporary file
*symref	Temporary file

SEE ALSO

ar(1), ld(1), tsort(1)

NAME

ls – Lists contents of a directory

SYNOPSIS

ls [**–RadCxmlnogrtucpFbqisf**] [*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. When no arguments are given, files are listed one per line.

There are three major listing formats. 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 are assumed.

The following options are available:

–R Recursively lists subdirectories encountered.

–a Lists all entries; including entries whose names begin with a period.

–d If an argument is a directory, lists only its name (not its contents); often used with **–l** to get the status of a directory.

–C Multicolumn output with entries sorted down the columns.

–x Multicolumn output with entries sorted across rather than down the page.

–m Stream output format.

–l Lists in long format, giving mode, number of links, owner, group, size in bytes, and time of last modification for each file (see below). If the file is a special file, the size field will 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 the group is not printed.

–g The same as **–l**, except that the owner is not printed.

–r Reverses the order of sort to get reverse alphabetic or oldest first as appropriate.

–t Sorts by time modified (latest first) instead of by name.

–u Uses time of last access instead of last modification for sorting (with the **–t** option) or printing (with the **–l** option).

–c Uses time of last modification of the i-node (file created, mode changed, etc.) for sorting (**–t**) or printing (**–l**).

–p Puts a slash (/) after each filename if that file is a directory.

–F Puts a slash (/) after each filename if that file is a directory and put an asterisk (*) after each filename if that file is executable.

–b	Forces printing of nongraphic characters to be in the octal \ddd notation.
–q	Forces printing of nongraphic characters in file names as the character (?).
–i	For each file, print the i-number structure in the first column of the report.
–s	For CRAY X-MP computer systems, give size in blocks, including indirect blocks (an approximation), for each entry. For CRAY-2 computer systems, give size in sectors for each entry.
–f	Forces 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.

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 (named pipe) special file;
–	If the entry is an ordinary file.

The next 9 characters are interpreted as three sets of 3 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 indication of set-ID is capitalized (S) 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.

FILES

/etc/passwd	Contains user IDs for **ls –l** and **ls –o** .
/etc/group	Contains group IDs for **ls –l** and **ls –g** .

BUGS

Unprintable characters in file names may confuse the columnar output options.

SEE ALSO

chmod(1), find(1)

NAME

m4 – Invokes a macro processor

SYNOPSIS

m4 [*options*] [*files*]

DESCRIPTION

The *M4* command invokes a macro processor intended for use as a general-purpose front end for any programming language. 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 requires a special state of mind.

–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 single quotes (grave accent, ascii 96) 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 arguments. It is useful for renaming macros, especially built-ins.
pushdef	Like *define*, but saves any previous definition.
popdef	Removes current definition of its arguments, 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 words **unix** and **CRAY** are predefined on all UNICOS systems. Additionally, one of the words **CRAY2**, **CRAY1**, or **CRAYXMP** is predefined.
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 (that is, ` ').
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	Identical to *include*, except that it says nothing if the file is inaccessible.
syscmd	Executes the UNICOS command given in the first argument. No value is returned.
sysval	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)
UNICOS Support Tools Guide, publication SG-2016

NAME

cray1, cray2, crayxmp, pdp11, u370, u3b, u3b5, vax – Provides truth value about processor type

SYNOPSIS

cray1

cray2

crayxmp

pdp11

u370

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.

pdp11	True if you are on a PDP-11/45 or PDP-11/70.
u3b	True if you are on a 3B20S.
vax	True if you are on a VAX-11/750 or VAX-11/780.
u370	True if you are on a UNIX/370 System.
u3b5	True if you are on a 3B5 System.
cray1	True if you are on a CRAY-1 Computer System with an I/O Subsystem.
cray2	True if you are on a CRAY-2 Computer System.
crayxmp	True if you are on a CRAY X-MP Computer System.

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

sh(1), test(1), true(1)

NAME

mail, rmail – Lets you send or read mail

SYNOPSIS

mail [**–epqr**] [**–f** *file*]

mail [**–t**] *persons*

rmail [**–t**] *persons*

DESCRIPTION

The *mail* command 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. The following commands are recognized:

<new-line>	Go on to next message.
+	Same as <new-line>.
d	Delete message and go on to next message.
p	Print message again.
–	Go back to previous message.
s [*files*]	Save message in the named *files* (**mbox** is default).
w [*files*]	Save message, without its header, in the named *files* (**mbox** is default).
m [*persons*]	Mail the message to the named *persons* (yourself is default).
q	Put undeleted mail back in the mail file (**/usr/spool/$LOGNAME**) and stop.
EOT <control-D>	Same as **q**.
x	Put all mail back unchanged in the mail file (**/usr/spool/$LOGNAME**) and stop.
!*command*	Escape to the shell to execute *command*.
*	Print a command summary.

The optional arguments alter the printing of the mail:

–e	causes mail not to be printed. An exit value of 0 is returned if the user has mail; otherwise, an exit value of 1 is returned.
–p	causes all mail to be printed without prompting for disposition.
–q	causes *mail* to terminate after interrupts. Normally an interrupt only causes the termination of the message being printed.
–r	causes messages to be printed in first-in, first-out order.
–f*file*	causes *mail* to use *file* (such as **mbox**) instead of the default *mail file*.
–t	Causes the message to be preceded by all *persons* the mail is sent to. *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 **$HOME/dead.letter** will be saved to allow editing and resending. This is a temporary file in that it is recreated each time it is needed, erasing the previous contents of **dead.letter** . 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, (that is, "From ...") are preceded with a >.

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.

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

/etc/passwd	To identify sender and locate home directory of user
/usr/mail/*user*	Incoming mail for *user*; that is, the *mailfile*
$HOME/mbox	Saved mail
/tmp/ma*	Temporary file
/usr/mail/*.lock	Lock for mail directory
$HOME/dead.letter	Unmailable text

BUGS

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

After an interrupt, the next message may not be printed; printing may be forced by typing a **p**.

SEE ALSO

login(1), write(1)

NAME

mailx – Electronic message processing system

SYNOPSIS

mailx [**–deFHinNU**] [**–f** *filename*] [**–h** *number*] [**–r** *address*] [**–s** *subject*]
[**–u** *user*] [*name...*]

DESCRIPTION

The *mailx* command invokes an electronic mail system. When reading mail, *mailx* lets you save, delete, and respond to messages. When sending mail, *mailx* lets you edit, review, and perform other modifications to the message as you enter them.

Mailx stores incoming mail in a standard file for each user, called the system *mailbox* for that user. When you call *mailx* to read messages, the *mailbox* is the default place to find them. As *mailx* reads messages, it marks them to be moved to a secondary file for storage, unless you specify that you want something else done with them. This secondary file is called the *mbox* and is normally located in the user's HOME directory (see MBOX in the (ENVIRONMENT VARIABLES section) for a description of this file). Messages remain in this file until you remove them.

Any arguments to options are assumed to be destinations (or recipients). If you do not specify recipients, *mailx* attempts to read messages from the system *mailbox.* The following command line options are available:

–d	Turns on debugging output. Neither particularly interesting nor recommended.
–e	Tests for presence of mail. *Mailx* prints nothing and exits with a successful return code if there is mail to read.
–f [*filename*]	Reads messages from *filename* instead of *mailbox.* If no *filename* is specified, the *mbox* is used.
–F	Records the message in a file named after the first recipient. Overrides the **record** variable, if set (see the ENVIRONMENT VARIABLES subsection).
–h *number*	The number of network "hops" made so far. This is provided for network software to avoid infinite delivery loops.
–H	Prints header summary only.
–i	Ignores interrupts. See also **ignore** in the ENVIRONMENT VARIABLES subsection.
–n	Does not initialize from the system default *Mailx.rc* file.
–N	Does not print initial header summary.
–r *address*	Passes *address* to network delivery software. All tilde commands are disabled.
–s *subject*	Sets the Subject header field to *subject*.
–u *user*	Reads *user*'s *mailbox.* This is only effective if *user*'s *mailbox* is not read protected.
–U	Converts *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 the COMMANDS subsection). When sending mail, *mailx* is in *input mode.* If you do not specify a subject on the command line, a prompt for the subject is printed. As you type the message, *mailx* reads the message and stores it in a temporary file. You can enter commands by beginning a line with the tilde (~) escape character followed by a single command letter and optional arguments. See TILDE ESCAPES subsection for a summary of these commands.

At any time, *mailx*'s behavior is governed by a set of *environment variables*. These are flags and valued parameters that are set and cleared using the **set** and **unset** commands. See the ENVIRONMENT VARIABLES subsection 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 the COMMANDS subsection) and are lists of recipients of any type.

Regular commands are of the form:

> [**command**] [*msglist*] [*arguments*]

If you do not specify a command 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 at any time the notion exists 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:

!*shell-command*
: Escapes to the shell. See **SHELL** in the ENVIRONMENT VARIABLES subsection.

# *comment*
: Null command (comment). This may be useful in *.mailrc* files.

=
: Prints the current message number.

?
: Prints a summary of commands.

alias *alias name* ...
group *alias name* ...
: Declares 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** in the ENVIRONMENT VARIABLES subsection.

cd [*directory*]
chdir [*directory*]
: Changes directory. If *directory* is not specified, $HOME is used.

copy [*filename*]
copy [*msglist*] *filename*
: Copies messages to the file without marking the messages as saved. Otherwise equivalent to the save command.

Copy [*msglist*]
: Saves the specified messages in a file whose name is derived from the author of the message to be saved, without marking the messages as saved. Otherwise equivalent to the Save command.

delete [*msglist*]
: Deletes messages from the *mailbox.* If **autoprint** is set, the next message after the last one deleted is printed (see the ENVIRONMENT VARIABLES subsection).

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*]
Deletes the specified messages from the *mailbox* and prints the next message after the last one deleted. Roughly equivalent to a **delete** command followed by a **print** command.

echo *string ...*
Echos the given strings (like *echo*(1)).

edit [*msglist*]
Edits the given messages. The messages are placed in a temporary file and the **EDITOR** variable is used to get the name of the editor (see the ENVIRONMENT VARIABLES subsection). Default editor is *ed*(1).

exit
xit
Exits from *mailx*, without changing the *mailbox*. No messages are saved in the *mbox* (see also **quit**).

file [*filename*]
folder [*filename*]
Quits from the current file of messages and reads 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*.

The default file is the current *mailbox*.

folders
Prints the names of the files in the directory set by the **folder** variable (see the ENVIRONMENT VARIABLES subsection).

followup [*message*]
Responds 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** in the ENVIRONMENT VARIABLES subsection.

Followup [*msglist*]
Responds 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** in the ENVIRONMENT VARIABLES subsection.

from [*msglist*]
Prints the header summary for the specified messages.

group *alias name ...*
alias *alias name ...*
Declares 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 that includes the message specified. The **screen** variable sets the number of headers per page (see the ENVIRONMENT VARIABLES subsection). 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* executes 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* ...
: Mails a message to the specified users.

mbox [*msglist*]
: Arranges for the given messages to end up in the standard *mbox* save file when *mailx* terminates normally. See **MBOX** in the ENVIRONMENT VARIABLES subsection for a description of this file. See also the **exit** and **quit** commands.

next [*message*]
: Goes 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*] [*shell-command*]
| [*msglist*] [*shell-command*]
: Pipes the message through the given *shell-command*. The message is treated as if it were read. If no arguments are given, the current message is piped through the command specified by the value of the **cmd** variable. If the **page** variable is set, a form feed character is inserted after each message (see the ENVIRONMENT VARIABLES subsection).

preserve [*msglist*]
hold [*msglist*]

Preserves the specified messages in the *mailbox.*

Print [*msglist*]
Type [*msglist*]

Prints the specified messages on the screen, including all header fields. Overrides suppression of fields by the **ignore** command.

print [*msglist*]
type [*msglist*]

Prints 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) in the ENVIRONMENT VARIABLES subsection.

quit

Exits 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*]

Sends 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 filename, the response is saved at the end of that file (see the ENVIRONMENT VARIABLES subsection).

reply [*message*]
respond [*message*]

Replies to the specified message, including all other recipients of the message. If **record** is set to a filename, the response is saved at the end of that file (see the ENVIRONMENT VARIABLES subsection).

Save [*msglist*]

Saves the specified messages in a file whose name is derived from the author of the first message. The name of the file is taken to be the author's name with all network addressing stripped off. See also the **Copy**, **followup**, and **Followup** commands and **outfolder** in the ENVIRONMENT VARIABLES subsection.

save [*filename*]
save [*msglist*] *filename*

Saves 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 in the ENVIRONMENT VARIABLES subsection and the **exit** and **quit** commands).

set
set *name*
set *name=string*
set *name=number*

Defines 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 the ENVIRONMENT VARIABLES subsection for detailed descriptions of the *mailx* variables.

shell
Invokes an interactive shell (see **SHELL** in the ENVIRONMENT VARIABLES subsection).

size [*msglist*]
Prints the size in characters of the specified messages.

source *filename*
Reads commands from the given file and returns to command mode.

top [*msglist*]
Prints the top few lines of the specified messages. If the **toplines** variable is set, it is taken as the number of lines to print (see in the ENVIRONMENT VARIABLES subsection). The default is 5.

touch [*msglist*]
Touches the specified messages. If any message in *msglist* is not specifically saved in a file, it will be placed in the *mbox* upon normal termination. See **exit** and **quit**.

Type [*msglist*]
Print [*msglist*]
Prints the specified messages on the screen, including all header fields. Overrides suppression of fields by the **ignore** command.

type [*msglist*]
print [*msglist*]
Prints 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) in the ENVIRONMENT VARIABLES subsection).

undelete [*msglist*]
Restores 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 in the ENVIRONMENT VARIABLES subsection).

unset *name* ...
Causes the specified variables to be erased. If the variable was imported from the execution environment (that is, a shell variable) then it cannot be erased.

version
Prints the current version and release date.

visual [*msglist*]
Edits 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 in the ENVIRONMENT VARIABLES subsection).

write [*msglist*] *filename*
Writes the given messages on the specified file, minus the header and trailing blank line. Otherwise equivalent to the save command.

xit
exit

Exits from *mailx*, without changing the *mailbox*. No messages are saved in the *mbox* (see also **quit**).

z[+|-]

Scrolls the header display forward or backward one screen–full. The number of headers displayed is set by the **screen** variable (see the ENVIRONMENT VARIABLES subsection).

TILDE ESCAPES

The following commands may be entered only from *input mode*, by beginning a line with the tilde escape character (~). See **escape** (in the ENVIRONMENT VARIABLES subsection) for changing this special character.

~! *shell-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 the ENVIRONMENT VARIABLES subsection).

~a

Insert the autograph string **sign** into the message (see the ENVIRONMENT VARIABLES subsection).

~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** in the ENVIRONMENT VARIABLES subsection for a description of this file.

~e

Invoke the editor on the partial message. See also **EDITOR** in the ENVIRONMENT VARIABLES subsection.

~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** in the ENVIRONMENT VARIABLES subsection for a description of this file.

~r *filename*

~< *filename*

~< !*shell-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** in the ENVIRONMENT VARIABLES subsection.

~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.**

~| *shell-command*
: Pipe the body of the message through the given *shell-command*. If the *shell-command* returns a successful exit status, the output of the command replaces the message.

ENVIRONMENT VARIABLES

The following are environment variables taken from the execution environment and 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 to *mailx*. You can import them from the execution environment or set them via the **set** command at any time. Use the **unset** command 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 **alternates** 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 **delete** and **undelete** 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=*shell-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=*shell-command*
The command to run when the edit 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 filename with this directory name to obtain the real filename. 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=*shell-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 (that is, 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=*shell-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=*shell-command*
Alternate command for delivering messages. Default is *mail*(1).

sendwait
Wait for background mailer to finish before returning. Default is **nosendwait**.

SHELL=
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=*shell-command*
The name of a preferred screen editor. Default is *vi*(1).

FILES

$HOME/.mailrc	Personal start-up file
$HOME/mbox	Secondary storage file
/usr/mail/*	Post office directory
/usr/lib/mailx/mailx.help*	Help message files
/usr/lib/mailx/mailx.rc	Global start-up file
/tmp/R[emqsx]*	Temporary files

BUGS

Where *shell-command* is shown as valid, arguments are not always allowed. Experimentation is recommended.

Internal variables imported from the execution environment cannot be **unset**.

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).

SEE ALSO

mail(1), pg(1), ls(1).

NAME

make – Maintains, updates, and regenerates groups of programs

SYNOPSIS

make [**–f** *makefile*] [**–p**] [**–i**] [**–k**] [**–s**] [**–r**] [**–n**] [**–b**] [**–e**] [**–t**] [**–d**] [**–q**] [*names*]

DESCRIPTION

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.

The following is a brief description of all options and some special names:

–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.
–p	Print out the complete set of macro definitions and target descriptions.
–i	Ignore error codes returned by invoked commands. This mode is entered if the fake target name **.IGNORE** appears in the description file.
–k	Abandon work on the current entry, but continue on other branches that do not depend on that entry.
–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 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.
–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.
.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.

Makefile contains a sequence of entries that specify dependencies. The first line of an entry is a blank-separated, non-null list of targets, then a :, then a list (possibly null) of prerequisite files or dependencies. Text following a ; and all following lines that begin with a tab are shell commands to be executed to update the target. The first line that does not begin with a tab or # begins a new dependency or macro definition. Shell commands may be continued across lines with the <backslash><new-line>

sequence. Everything printed by *make* (except the initial tab) is passed directly to the shell as is. Thus,

```
echo a\
b
```

produces:

```
ab
```

exactly the same as the shell.

The # symbol starts a comment and a new-line ends a comment.

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 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 of the **MAKEFLAGS** macro under *Environment*). The **–t** (touch) option updates the modified date of target files 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 command (possibly null or implicit) associated with them. The previous version of *make* assumed if no command was specified explicitly that the command was null.

An interrupt or quit signal received during execution of a command line causes the associated target to be deleted unless the target is a dependent of the special name **.PRECIOUS**.

Environment

The environment is read by *make*. All variables are assumed to be macro definitions and are 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 **–r**) 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 executing anything.

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*. (A $$ is a dollar sign.) 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.

Internal Macros

There are four macros maintained internally that are useful in writing rules for building targets.

$*** The **$* macro stands for the filename part of the current dependent with the suffix deleted. It is evaluated only for inference rules.

$@** The **$@ macro stands for the full target name of the current target. It is evaluated only for explicitly named dependencies.

$<** The **$< macro is only evaluated for inference rules or the **.DEFAULT** rule. It is the module which is out of date with respect to the target (that is, the "manufactured" dependent file name). Thus, in the **.c.o** rule, the **$<** macro would evaluate to the **.c** file. An example for making optimized **.o** files from **.c** files is:

```
.c.o:
        cc -c -O $*.c
```

or:

```
.c.o:
        cc -c -O $<
```

$?** The **$? macro is evaluated when explicit rules from the makefile are evaluated. It is the list of prerequisites that are out of date with respect to the target; essentially, those modules which must be rebuilt.

The **$***, **$@**, and **$<** macros can have alternative forms. When an upper case **D** or **F** is appended to any of these macros, the meaning is changed to "directory part" for **D** and "file part" for **F**. Thus, **$(@D)** refers to the directory part of the string **$@**. If there is no directory part, ./ is generated.

Suffixes

Certain names (for instance, those ending with **.o**) have inferable prerequisites such as **.c**, **.s**, etc. If no update commands for such a file appear in *makefile*, and if an inferable prerequisite exists, that prerequisite is compiled to make the target. In this case, *make* has inference rules which allow building files from other files by examining the suffixes and determining an appropriate inference rule to use.

The internal rules for *make* are contained in the source file **rules.c** for the *make* program. These rules can be modified locally. To print out the rules compiled into the *make* on any machine in a form suitable for recompilation, the following command is used:

```
make -fp - 2>/dev/null </dev/null
```

The only peculiarity in this output is the **(null)** string which *printf*(3S) prints when handed a null string.

A rule with only one suffix (that is, **.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 (such as shell procedures and 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 .h .sh
```

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** and **YFLAGS** are used for compiler options to *cc*(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.

FILES

Makefile
makefile
s.Makefile
s.makefile

BUGS

Some commands return nonzero status inappropriately; use **–i** to overcome the difficulty.
File names with the characters =, :, or @ do not work.
Commands that are directly executed by the shell, notably *cd*(1), are ineffectual across new-lines in *make*.
This release of **make** does not have library support features built in for UNICOS.

SEE ALSO

cc(1), cd(1), lex(1), sh(1), yacc(1)
printf(3S) in the CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual, publication SR-2013
UNICOS Support Tools Guide, publication SG-2016.

NAME

makekey – Generates encryption key

SYNOPSIS

/usr/lib/makekey

DESCRIPTION

The *makekey* command 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.

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, ., /, upper-case letters, 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 (such as *ed*(1) and *crypt*(1)). Usually, its input and output will be pipes.

SEE ALSO

crypt(1), ed(1)
passwd(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

man – Prints entries in this manual

SYNOPSIS

man [**-oulbpr**] [*section*] *titles*

DESCRIPTION

The *man* command locates and prints the entry of this manual named *title* in the specified *section*. (The word "page" is often used as a synonym for "entry" in this context.) The *title* is entered in lower case. The *section* number may have a letter suffix or, for UNICOS running on a CRAY X-MP or CRAY-1 computer system, a two-digit suffix. If no *section* is specified, the whole manual is searched for *title* and all occurrences of it are printed. The *section* may be changed before each *title*. On a CRAY-2 computer system, the available sections are: 1, 1m, 2, 3c, 3f, 3m, 3n, 3s, 3sci, 3q, 3w, 3z, 4d, 4f, and 4n. On a CRAY X-MP or CRAY-1 computer system, the available sections are: 1, 1m, 2, 3c, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 3.10, 3.11, 3.12, 3.13, 3.14, 3.15, 3.16, 3.17, 3.18, 3.19, 4d, 4f, and 4n. Sections 3n, 3w, 3.18, and 4n are available only if your site is licensed for the TCP/IP networking software.

Pre-formatted manual entries are stored in the **/usr/man** directory hierarchy. The format style is appropriate for terminals and line printers. The *man* command further processes the entries to make the output more suitable for video displays. *Options* provide control of the processing done by *man*:

–o Removes overstrikes intended to produce an emboldened effect.

–u Removes overstrikes intended to produce underlining.

–b Removes all *character-backspace* sequences. The **–o** and **–u** options remove most of these sequences, but occasionally overstrike sequences are used to produce symbols that are not in the ascii character set. The **–b** option is useful for devices that do not handle the backspace character.

–l Reduce all occurrences of two or more blank lines in sequence to a single blank line.

–p Paginate; print 18 lines of output, print a ':' prompt character, and wait for a RETURN or CONTROL-d to be typed. Backspace-character sequences are used (unless option **b** is set), to remove the ':' prompt when the CONTROL-d response is typed.

-r Reset all **oulp** options. These options are set by default when output is being sent to a terminal file. If the output of *man* is not directed to a terminal, no options are set by default. This type of output is useful for printers.

As an example:

```
man man
```

reproduces this entry on the standard output. It also reproduces any other entries named *man* that may exist in other sections of the manual.

FILES

/usr/man/man1/*
in the UNICOS User Commands Reference Manual, publication SR-2011

/usr/man/man1m/*
in the UNICOS Administrator Commands Reference Manual, publication SR-2022

/usr/man/man2/*
in the UNICOS System Calls Reference Manual, publication SR-2012

/usr/man/man3c/*
in the CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual, publication SR-2013, for UNICOS running on a CRAY-2 computer system or the CRAY X-MP and CRAY-1 C Library Reference Manual, publication SR-0136 for UNICOS running on a CRAY X-MP or CRAY-1 computer system

/usr/man/man3f/*
/usr/man/man3m/*
/usr/man/man3n/*
/usr/man/man3p/*
/usr/man/man3q/*
/usr/man/man3s/*
/usr/man/man3sci/*
/usr/man/man3uc/*
/usr/man/man3uf/*
/usr/man/man3w/*
/usr/man/man3z/*
in the CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual, publication SR-2013, for UNICOS running on a CRAY-2 computer system

/usr/man/man3.1/*
/usr/man/man3.2/*
/usr/man/man3.3/*
/usr/man/man3.4/*
/usr/man/man3.5/*
/usr/man/man3.6/*
/usr/man/man3.7/*
/usr/man/man3.8/*
/usr/man/man3.9/*
/usr/man/man3.10/*
/usr/man/man3.11/*
/usr/man/man3.12/*
/usr/man/man3.13/*
/usr/man/man3.14/*
/usr/man/man3.15/*
/usr/man/man3.16/*
/usr/man/man3.17/*
/usr/man/man3.18/*
/usr/man/man3.19/*
in the Programmer's Library Reference Manual, publication SR-0113, for UNICOS running on a CRAY X-MP or CRAY-1 computer system

/usr/man/man4d/*
/usr/man/man4f/*
/usr/man/man4n/*
in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NOTES

All entries are formatted for display on a terminal. Some information is necessarily lost because of the display limitations.

NAME

mesg – Permits or denies messages

SYNOPSIS

mesg [n or –n]
mesg [y or –y]

DESCRIPTION

The *mesg* command with argument **n** forbids messages being written to the user's terminal via *write*(1) by revoking non-user write permission. The *mesg* command with argument **y** reinstates permission. If no arguments are given, *mesg* reports the current state without changing it.

FILES

/dev/tty*

MESSAGES

Exit status is 0 if messages are receivable, 1 if not, and 2 on an error.

SEE ALSO

write(1)

NAME

mkdir – Creates a directory

SYNOPSIS

mkdir *dirname* ...

DESCRIPTION

The *mkdir* command creates specified directories in mode 777 (unless altered by *umask*(1)). The standard entries **.** (for the directory itself) and **..** (for its parent) are made automatically.

Mkdir requires write permission in the parent directory.

MESSAGES

Mkdir returns exit code 0 if all directories were successfully made; otherwise, it prints a diagnostic message and returns a nonzero exit code.

SEE ALSO

sh(1), rm(1), umask(1)

NAME

more, page – Lets you peruse text one screenful at a time

SYNOPSIS

more [**–cdflsu**] [*–n*] [*+linenumber*] [*+/pattern*] [*name* ...]

page *more options*

DESCRIPTION

The *more* command is a filter that lets you examine a continuous text one screenful at a time. It normally pauses after each screenful, printing --More-- at the bottom of the screen. If you then type a carriage return, one more line is displayed. If you hit the spacebar, another screenful is displayed. Other possibilities are enumerated later.

The command line options are:

–n Specifies an integer that is the size (in lines) of the window, which *more* uses instead of the default.

–c Draws each page by beginning at the top of the screen and erasing each line just before drawing on it. This avoids scrolling the screen, making it easier to read while *more* is writing. This option is ignored if the terminal does not have the ability to clear to the end of a line.

–d Prompts the user with the message "Hit space to continue, Rubout to abort" at the end of each screenful. This is useful if you are using *more* as a filter in some setting, such as a class, where many users may be unsophisticated.

–f Counts logical, rather than physical 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 that would ordinarily occupy screen positions, but that 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 Does not treat CONTROL-L (form feed) specially. If you do not specify this option, *more* pauses after any line that contains a CONTROL-L, as if the end of a screenful had been reached. Also, if a file begins with a form feed, the screen is cleared before the file is printed.

–s Squeezes multiple blank lines from the output, producing only one blank line. Especially helpful when viewing *nroff* output, this option maximizes the useful information present on the screen.

–u Suppresses underline processing. Normally, *more* handles underlining such as produced by *nroff* in a manner appropriate to the particular terminal: if the terminal can perform underlining or has a stand-out mode, *more* outputs appropriate escape sequences to enable underlining or stand-out mode for underlined information in the source file.

+linenumber
Starts up at *linenumber.*

+/pattern
Starts up two lines before the line containing the regular expression *pattern.*

If *more* 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.

The *more* commands looks in the **/usr/lib/terminfo** directiory to determine terminal characteristics and the default window size. On a terminal capable of displaying 24 lines, the default window size is 22 lines.

The *more* command looks in the environment variable **MORE** to preset any desired flags. For example, if you prefer to view files using the *–c* mode of operation, the *csh* command

```
"setenv MORE -c"
```

or the *sh* command sequence

```
"MORE='-c' ; export MORE"
```

would cause all invocations of *more* , including invocations by programs such as *man* and *msgs,* to use this mode. Normally, you place the command sequence that sets up the **MORE** environment variable in the **.cshrc** or **.profile** file.

If *more* is reading from a file rather than a pipe, 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 that may be typed when *more* pauses, and their effects, are as follows (*i* is an optional integer argument, defaulting to 1) :

i <space>
: Displays *i* more lines, (or another screenful if no argument is given)

^D	Displays 11 more lines (a "scroll"). If *i* is given, then the scroll size is set to *i* .
d	Same as CONTROL-D
i z	Same as typing a space except that *i* , if present, becomes the new window size.
i s	Skips *i* lines and print a screenful of lines
i f	Skips *i* screenfuls and print a screenful of lines
q or Q	Exits from *more*
=	Displays the current line number
v	Starts up the editor *vi* at the current line
h	Help command; gives a description of all the *more* commands.
i /expr	Searches for the *i* -th occurrence of the regular expression *expr.* If less than *i* occurrences of *expr* exist, 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. You can use the erase and kill characters to edit the regular expression. Erasing back past the first column cancels the search command.
i n	Searches for the *i* -th occurrence of the last regular expression entered.
'	(Single quote) Goes to the point from which the last search started. If no search has been performed in the current file, this command goes back to the beginning of the file.

!command
: Invokes a shell with *command*. The characters '%' and '!' in "command" are replaced with the current file name and the previous shell command respectively. If no current file name exists, '%' is not expanded. The sequences "\%" and "\!" are replaced by "%" and "!" respectively.

i :n
: Skips to the *i* -th next file given in the command line (skips to last file if n doesn't make sense).

i:p Skips to the *i*-th previous file given in the command line. If this command is given in the middle of printing out a file, then *more* goes back to the beginning of the file. If *i* doesn't make sense, *more* skips back to the first file. If *more* is not reading from a file, the bell is rung and nothing else happens.

:f Displays the current file name and line number.

:q or :Q Exits from *more* (same as q or Q).

. (Dot) repeats the previous command.

The commands take effect immediately; that is, 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%) message.

At any time when output is being sent to the terminal, the user can hit the quit key (normally Control-\). *More* stops sending output and displays the usual --**More**-- prompt. You 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. Thus, what you type does 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/terminfo	Terminal data base
/usr/lib/more.help	Help file

SEE ALSO

csh(1), man(1), script(1), sh(1)

NAME

mv – Moves files

SYNOPSIS

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

DESCRIPTION

The file *file1* is 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 moved to that directory. If *target* is a file, its contents are destroyed.

If *mv* determines that the mode of *target* forbids writing, it will print the mode (see *chmod*(2)), ask for a response, and read the standard input for one line. If the line begins with **y**, the *mv* occurs, if permissible; if not, the command exits. No questions are asked and the *mv* is done when the **–f** option is used or if the standard input is not a terminal.

The *mv* command allows *file1* to be a directory, in which case the directory rename will occur only if the two directories have the same parent; *file1* is renamed *target*. 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.

FILES

/usr/lib/mv_dir mv assist program If *file1* and *target* lie on different file systems, *mv* must copy the file and delete the original. In this case any linking relationship with other files is lost.

NOTES

If *file1* and *target* lie on different file systems, *mv* must copy the file and delete the original. In this case any linking relationship with other files is lost.

SEE ALSO

cp(1), cpio(1), ln(1), rm(1)
chmod(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

mxm – Invokes the mod creation program

SYNOPSIS

mxm [**–i** *filenames*] [**–I** *indirect*] [**–b** *basedir*] [**–p** *paths*] **–m** *modname*

DESCRIPTION

Mxm is a source control program used in conjunction with *scm(1). Mxm* is similar to the MODECKS utility available with the Cray operating system COS. It takes one or more edited source files and compares them with the original *source control modules* created by *scm.* (See *scm*(1) for details of the source control module.) *Mxm* creates a single output file in the current working directory. This file is a *mod* that contains *scm* directives. This mod should contain everything needed to implement one bugfix or add one new feature to the previous level of software.

The created mod can be edited to add the *mod header,* which is simply the *scm* comment directives that document the bugfix or feature. Refer to the *scm* documentation for details of *scm* directives.

The keyletters and their meanings are:

–i *filenames* Specify one or more input source filenames. If more than one filename is specified, they must be separated by commas with no intervening white space. This keyletter is required if the **–I** keyletter is not specified.

–I *indirect* Read a list of input source filenames from the file *indirect.* This keyletter is useful if modifying many source files in a single run. This keyletter is required if the **–i** keyletter is not specified.

–b *basedir*

–p *paths* These two keyletters are used together to specify the directory or directories *mxm* will search to find the source control modules that will be compared to the input source files named by the **–i** and **–I** keyletters. If only one directory is to be searched, it may be specified with the **–b** keyletter alone. The default for **–b** is the current working directory. If more than one directory is to be searched, suffixes may be specified with the **–p** keyletter. The path name suffixes *paths* must be separated by commas with no intervening white space. The directory path names to search are derived by appending the **–p** keyletter path names to the **–b** keyletter path name. The default for **–p** is "/".

–m *modname* The name of the created mod, and the name to use in the generated ***m** directive. This keyletter is required.

Up to twenty-six input files may be specified; this limitation is due to the naming convention used by *scm,* where a letter *a–z* is appended to the mod name.

Input files are specified with the **–i** and **–I** keyletter. Each input file must have a corresponding *source control module,* which is searched for in the directory or directories derived from the **–b** and **–p** keyletter. For example, input file **foo.c** must have a corresponding source control module **foo.c.m.**

The output (mod) file will not be created if *mxm* detected any errors. Output files are always placed in the current working directory. *Mxm* will never silently replace a file; an error will be reported if an output file name would conflict with an existing file name.

EXAMPLE

Contents of source control module **/usr/src/cmd/foo.c.m**:

```
foo.1       /* foo.c – program to print something. */
foo.2
foo.3       #include <stdio.h>
foo.4
foo.5       main()
foo.6       {
foo.7               printf( "This is the future.\n" );
foo.8       }
```

Contents of edited source file **foo.c**:

```
/* foo.c – program to print something on stderr. */

#include <stdio.h>

main()
{
        fprintf( stderr, "This is the future.\n" );
}
```

Using mxm to create a mod:

```
mxm –i foo.c –b /usr/src/cmd –m foomod
```

After *scm* has extracted a source file from the source control module, the file can be edited and a mod created. This run of *mxm* takes edited source file **foo.c** and searches in directory **/usr/src/cmd** for the corresponding source control module, **foo.c.m.** The mod is called *foomod.*

Contents of mxm output file foomod:

```
*m foomod
*f foo.c
*d foo.1
/* foo.c – program to print something on stderr. */
*d foo.7
        fprintf( stderr, "This is the future.\n" );
```

A mod header (additional ***c** directives) should be added after the ***m** directive. A mod trailer (more ***c** directives) would be nice too. The mod file can then be added to the collection of mods targeted for the next software release.

SEE ALSO

scm(1)

NAME

nasa – Adds ASA carriage control characters for printing

SYNOPSIS

nasa [**–t** *tabspace*] [*file*]

DESCRIPTION

The *nasa* command provides the opposite function of the *asa(1)* command. It transforms typical UNICOS command text output to make it suitable for output to line printers that require ASA carriage control characters. It processes either the *file* whose name is given as an argument or the standard input if you do not specify a file name. Tabs are expanded to the appropriate number of spaces to position a subsequent character at the next tab stop. Tab stops are every eigth character position by default or every *tabspace* characters if you specify the –t option. Other transformations performed are as follows:

line feed	becomes line feed, space
carriage return	becomes line feed, +
form feed	becomes line feed, 1

The *nasa* command forces the first line to start on a new page by starting its output with a **1**. Backspace characters (ASCII code 8), are properly compensated for to preserve the column position of subsequent tab characters, but the destination printer may not accept them.

Below is one suggestion for preparing all the output of a job for printing on a line printer requiring carriage control:

```
(
         set -v        # print command names as they are executed
         UNICOS commands
              .
              .
              .
         application | asa
         echo '\f'     # form feed
              .
              .
              .
) 2>&1 | nasa > tracefile
```

SEE ALSO

asa(1), expand(1), unexpand(1)

NAME

netstat – Displays network status

SYNOPSIS

netstat [**–Aaimnrst**] [*interval*] [*system*] [*core*]

DESCRIPTION

The *netstat* command symbolically displays the contents of various network-related data structures. The options have the following meaning:

–A	Shows the address of any associated protocol control blocks; used for debugging
–a	Shows the state of all sockets; normally sockets used by server processes are not shown
–i	Shows the state of interfaces that have been auto-configured (those interfaces statically configured into a system but not located at boot time are not shown)
–m	Shows statistics recorded by the memory management routines (the network manages a "private share" of memory)
–n	Shows network addresses as numbers (normally *netstat* interprets addresses and attempts to display them symbolically)
–r	Shows the routing tables
–s	Shows per-protocol statistics
–t	Shows the timer value (when used with *–i* option)

The arguments, *system* and *core* allow substitutes for the defaults **/unicos** and **/dev/kmem** (not recognized in UNICOS running on CRAY X-MP or CRAY-1 systems).

If *interval* is specified, *netstat* will continuously display the information regarding packet traffic on the configured network interfaces, pausing *interval* seconds before refreshing the screen.

There are a number of display formats depending on the information presented. The default display, which is for active sockets, shows the local and remote addresses, send and receive queue sizes (in bytes), protocol, and, optionally, the internal state of the protocol.

Address formats are of the form *host.port* or *network.port* if a socket address specifies a network but no specific host address. When known the host and network addresses are displayed symbolically according to the files **/etc/hosts** (see *hosts*(4F)) and **/etc/networks** (see *networks*(4F)), respectively. If a symbolic name for an address is unknown, or if the **–n** option is specified, the address is printed in the Internet "dot format"; refer to *inet*(3N) for more information regarding this format. Unspecified, or *wildcard*, addresses and ports appear as an asterisk (*).

The interface display provides a table of cumulative statistics regarding packets transferred, errors, and collisions. The network address (currently Internet specific) of the interface and the maximum transmission unit in bytes (*mtu*) are also displayed.

The routing table display indicates the available routes and their status. Each route consists of a destination host or network and a gateway to use in forwarding packets. The flags field shows the state of the route (U if "up"), and whether the route is to a gateway (G). Direct routes are created for each interface attached to the local host. The **refcnt** field gives the current number of active uses of the route. Connection-oriented protocols normally hold on to a single route for the duration of a connection, while connectionless protocols obtain a route then discard it. The **use** field provides a count of the number of packets sent using that route. The interface entry indicates the network interface utilized for the route.

When *netstat* is invoked with an *interval* argument, it displays a running count of statistics related to network interfaces. This display consists of a column summarizing information for all interfaces, and a column for the interface with the most traffic since the system was last rebooted. The first line of each screen of information contains a summary since the system was last rebooted. Subsequent lines of output show values accumulated over the preceding interval.

BUGS

The notion of errors is ill-defined. Collisions mean something else to an IMP.

SEE ALSO

hosts(4F), networks(4F), protocols(4F), services(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

newacct – Changes account ID

SYNOPSIS

newacct [*account name*]
newacct -l

DESCRIPTION

The *newacct* command changes the account ID of the calling shell. If you do not specify *account name*, it defaults to your account ID. If you specify an *account name*, *newacct* validates whether you have permission to change the account ID to the requested value. If -l is specified, the current account name is printed.

The *newacct* command is only supported if UNICOS is generated to use the UNICOS user context database.

FILES

/etc/uentry
/etc/UID.map
/etc/ACID.map

SEE ALSO

acctid(2) in the UNICOS System Calls Reference Manual, publication SR-2012
getpwent(3C), getpwnam(3C), getpwuid(3C), id2nam(3C), putpwent(3C) in the CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual, publication SR-2013, for UNICOS running on a CRAY-2 computer system
passwd(4F), uentry(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

newgrp – Logs in to a new group

SYNOPSIS

newgrp [**–**] [*group*]

DESCRIPTION

The *newgrp* command changes your group identification. You remain 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. You are always given a new shell, replacing the current shell, by *newgrp*, regardless of whether it terminated successfully or due to an error condition (that is, an unknown group).

Exported variables retain their values after invoking *newgrp*; however, all unexported variables are either reset to their default value or set to null. System variables (such as PS1, PS2, PATH, MAIL, and HOME), unless exported by the system or explicitly exported by you, 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 your 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 you do not, or if the group has a password and you are 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

BUGS

There is no convenient way to enter a password into **/etc/group**. Use of group passwords is not recommended, because they encourage poor security practices. Group passwords may disappear in the future.

SEE ALSO

login(1), sh(1)
group(4), passwd(4) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

news – Prints news items

SYNOPSIS

news [–a] [–n] [–s] [*items* **]**

DESCRIPTION

The *news* command 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." Valid options are as follows:

–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** to report if there is news.

All other arguments are assumed to be specific news items that are to be printed.

If an interrupt signal is received during the printing of a news item, printing stops and the next item is started. Another interrupt within one second of the first causes the program to terminate.

FILES

/etc/profile
/usr/news/*
$HOME/.news_time

SEE ALSO

sh(1)
profile(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

nice – Runs a command at low priority

SYNOPSIS

nice [– *increment*] *command* [*arguments*]

DESCRIPTION

The *nice* command 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. An *increment* larger than 19 is equivalent to 19.

The super user may run commands with priority higher than normal by using a negative increment, such as **—10**.

MESSAGES

The *nice* command returns the exit status of the subject command.

NOTES

The csh command has a built-in nice command with slightly different characteristics. See csh(1).

SEE ALSO

nice(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

nm – Prints name list

SYNOPSIS

nm **–g** [*options*] *file*

DESCRIPTION

The *nm* command prints the name list (entries and externals) of each object *file* in the argument list.

Each symbol name is preceded by a value character (or by blanks if undefined) and either the letter **U** (undefined, an external) or **T** (entry point, text segment symbol). The **–g** argument is required. It prints only global (external) symbols; that is, entry names and external names.

The *nm* command accepts the following options:

–a Uses the sspecified filename instead of the PDT name.

–o Prints the module name at the beginning of each line. This is in addition to the announcement at the beginning of the module.

–u Prints only undefined symbols. This option causes entry symbols to be ignored.

–x Prints the filename after opening.

–c Includes common blocks in the output list. The character next to the name is 'C' if there is no data initialization in the common block. If there is initialization, a 'D' is printed next to the name. The number printed is the number of words in the common block (in octal). hk

SEE ALSO

ar(1), lorder(1)
relo(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

BUGS

This implementation was designed to make *lorder*(1) function so that some structure could be imposed upon relocatable libraries.

NAME

nm – Prints name list

SYNOPSIS

nm –g *filenames*

DESCRIPTION

Nm prints the name list (entries and externals) of each object *file* in the argument list.

Each symbol name is preceded by its value (blanks if undefined) and either the letter **U** (undefined, an external) or **T** (entry point, text segment symbol).

The **–g** option is required. It prints only global (external) symbols; that is, entry names and external names. The entries are printed with a value character for identification. The external names have no value indicated by "U"; the text area symbol is indicated by "T".

BUGS

This implementation is minimal at best, but it at least gives the user some idea what is in an object file.

SEE ALSO

reloc(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

nmab – Produces a list of names and addresses from executable file

SYNOPSIS

nmab [-lnwac] [*-number*] [*files*]

DESCRIPTION

The *nmab* command produces a listing of the global names inside an executable file along with their values and addresses. It may be used to generate load maps. If not named, the input file defaults to **a.out**. The meanings of the options are as follows:

-l	Produce a long listing. This includes file names and address space headers.
-n	Produce a listing sorted numerically. This turns off the default of producing both alphabetically and numerically sorted lists.
-a	Produce a listing sorted alphabetically. This turns off the default of producing both alphabetically and numerically sorted lists.
-w	Use word addresses with parcel modifiers rather than byte addresses.
-0	Produce a listing that includes the names from address space zero, common memory. This turns off the default of producing lists for each address space.
-number	As above for address space *number*. (*nnumber=1, ... 7*) For the CRAY-2 computer system, address space 1 is local memory. The other address spaces are currently unassigned.
-c	Produce a listing that includes the names and values of constants. This turns off the default of producing a list for each address space. The constants may be considered address space -1.

BUGS

Constants are printed as addresses with an octal point. Thus, the constant 0314 would be printed as 031.4 if the **-w** option was not specified and as 03.14 if it was.

NAME

nohup – Runs a command immune to hangups and quits

SYNOPSIS

nohup *command* [*arguments*]

DESCRIPTION

The *nohup* command executes *command* with hangups and quits ignored. If you do not redirect the output, it will be sent to **nohup.out**. If **nohup.out** is not writable in the current directory, output is redirected to **$HOME/nohup:out**.

NOTES

The *csh* command has a built-in **nohup** command with slightly different characteristics. See *csh*(1)

SEE ALSO

nice(1)
signal(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

nqsintro – Introduction to the Network Queuing System (NQS)

DESCRIPTION

This introduction provides an overview of NQS capabilities and describes how to start using NQS. If you are an NQS system administrator, see *qmgr*(1M) for a discussion of the internal operations and management of NQS.

This discussion refers to individual NQS commands which provide more detailed information.

NQS Overview

The UNICOS batch job processing capability is based on NQS, which was developed by Sterling Software for the National Aeronautics Space Administration (NASA) Numerical Aerodynamic Simulation (NAS) project. NQS lets you submit, terminate, monitor, and, within limits, control batch requests submitted to the batch system. You can send batch requests to your own system or to other appropriately configured computer systems in your facility's network.

NQS lets you perform the following activities:

- Submit requests to a batch queue with *qsub*(1). *Qsub* lets you specify numerous qualifications for your batch request, including start time, memory and CPU resource limits, exporting of environment variables, and the queue to which the request is submitted.
- Display the status of NQS queues with *qstat*(1). *Qstat* displays the ordering of NQS requests and provides information about the requests in their respective queues.
- Delete or signal NQS requests with *qdel*(1).
- Display the status of NQS devices with *qdev*(1).
- Display supported batch limits and shell strategies for each host with *qlimit*(1).
- Submit a hard copy print request to NQS with *qpr*(1).
- Function as an NQS system administrator with *qmgr*(1M), if you have the required privileges.

GETTING STARTED

The first step in using NQS is to create a shell script of commands to execute a sequence of actions the batch request performs. Within this file you can also include flags that qualify the *qsub*(1) command, as long as the flags appear before the shell commands and are preceded by the #, @, and $ characters. For example, the following batch request file specifies that the request is sent after 11 p.m. on Tuesday, and a list of current system users is produced:

```
# @$-a 11pm Tuesday
who
```

The following is a slightly more complex example of a batch request file:

```
#
# Slightly more complex
#
# @$–q queue1 # Queues request to queue1
# @$–lt 00:01:00 # Specifies a per-process CPU limit of 1 minute
#
qstat –l
cd junk
ls –l
cc –o test.homer ralph.c
test.homer
ls –l
rm test.homer
ls –l
```

You can use two methods to include data on which the shell commands must act. The first method is to create a separate data file to which the command is linked, as follows:

```
sort < sortinput
```

In this example, **sortinput** contains lines of text that you want sorted. The second method of including data within your batch request is to use the double-redirect operator, as follows:

```
sort << EOF
Robert Cohn was
once middleweight boxing
champion of Princeton.
EOF
```

This example sorts the specified lines of text. In contrast, the following shell script will not sort these lines:

```
sort
Robert Cohn was
once middleweight boxing
champion of Princeton.
```

In this example, the **sort** command encounters an immediate end-of-file indicator when reading the standard input file **stdin**, which defaults to **dev/null.**

After creating a batch request file, use *qsub*(1) to send the batch request for execution. *Qsub* lets you specify several controlling factors, including per-process CPU time limits, time the batch request begins execution, and the queue to which the batch request is submitted. For example, the following command submits the file **request1** to queue1 at 11 p.m. on the following day, and exports all environment variables:

```
qsub –a "11pm Tom." –q queue1 –x request1
```

Specify limits no larger than those required to execute your request, because NQS uses these limits for batch request scheduling. For example, batch requests requesting large amounts of CPU time are generally run less often than batch requests requesting small amounts of CPU time.

Qsub also lets you interactively enter the commands to be executed by the batch request. Exclude the *script-file* from the *qsub* command line and press the carriage return. All lines that you enter in the standard input buffer are then executed as the batch request. Signal the end of the standard input file with a CONTROL-d, as follows:

```
$ qsub –a "11pm Tom." –q queue1
ls
who
(control-d)
$
```

If your batch request is successfully submitted, NQS returns a message that displays the request-id and destination queue of your batch request. For example, the following message indicates that NQS assigned your request a sequence number of 125, you are working on machine MH-Vax, and your request was sent to queue1.

```
Request 125.MH-Vax submitted to queue: queue1.
```

By default, NQS also assigns a request-name to your batch request. The default NQS request name is equivalent to the name of the script file you specified on the command line. The request name and the request-id are associated with your request throughout the network.

After submitting the request, use the *qstat*(1) command to display the queue and batch request status, as follows:

```
qstat queue1
```

If this command is entered soon after the previously submitted batch request is sent, it produces the following output:

```
A_little@cray2;  type=BATCH;  [ENABLED, RUNNING];  PIPEONLY;  pri=50
 0 exit;   1 run;   0 stage;   0 queued;   0 wait;   0 hold;   0 arrive;
 Run_limit = 10;
 Queue Complex Membership:  com1;

REQUEST NAME  JID   STATE    NICE  SIZE  CPU  CPLIM  OWNER  NQSID

j             5420  RUNNING  0     0     0    0      cda    4.cray2
```

See *qstat*(1) for more information on displaying the status of NQS batch requests.

Use the *qdel*(1) command to delete a running NQS batch request. To delete a batch request, you must be the owner of that request, unless you have super user privileges or are an NQS manager. To delete a batch request, specify the request-id, as follows:

```
qdel 4.cray2
```

This command deletes the batch request with request-id 4.cray2.

You can also use the *qdev*(1), *qpr*(1), and *qlimit*(1) commands to display the status of NQS devices, submit a hard-copy request, and display the NQS resource limits, respectively.

After the batch request completes processing on the executing machine, the **stdout** and **stderr** files are returned, by default, to your home directory on the originating machine. By default, the name of the standard output file contains the first 7 characters of the request-name, followed by the characters **.o**, followed by the request sequence number of the request-id. The standard error file has the same naming convention, except that the second set of characters is **.e**. For example, the standard output file of the batch request submitted above is named request1.o4, and the standard error file is named request1.e4.

SEE ALSO

qdel(1), qdev(1), qlimit(1), qpr(1), qstat(1), qsub(1)
qmgr(1M) in the UNICOS Administrator Commands Reference Manual, publication SR-2022

NAME

od – Produces an octal dump

SYNOPSIS

od [–bcdosxpBW] [***file*** **] [[+]*offset*[.][b]]**

DESCRIPTION

The *od* command dumps *file* in one or more formats as selected by the first argument. The *file* argument specifies which file is to be dumped. If no file argument is specified, the standard input is used. If the first argument is missing, **–o** is default. The meanings of the format options are:

–b	Interpret bytes in octal.
–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 words in unsigned decimal.
–o	Interpret words in octal.
–s	Interpret 16-bit words in signed decimal.
–x	Interpret words in hexadecimal.
–p	Print parcels as opposed to words.
–B	Print address in bytes (default for **–b** and **–c**)
–W	Print address in words (default for all except **–b** and **–c**)

The *offset* argument specifies the offset in the file where dumping is to commence. This argument is normally interpreted as octal bytes. If **.** is appended to *offset*, the offset is interpreted in decimal. If **b** is appended to *offset*, the offset is interpreted in blocks of 512 bytes. If the *file* argument is omitted, the *offset* argument must be preceded by **+**.

Dumping continues until the end-of-file is reached.

NAME

pack, pcat, unpack – Compresses and expands 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
- 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* can not 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

pascal – Invokes Pascal compiler

SYNOPSIS

pascal [**–i** *idn*] [**–l** *ldn*] [**–b** *bdn*] [**–o** *list*] [**–V**] [**–C** *list*]

DESCRIPTION

The *pascal* command line invokes the Pascal compiler under UNICOS. You select compiler parameters either explicitly by listing them on the command line or implicitly by accepting the default values. All parameters are optional and have default values. Options are:

–i *idn*	Specifies the file containing the Pascal source code. When *idn* is not a complete path name, the input file defaults to a working directory. The default is **stdin.**
–l *ldn*	Specifies the file receiving the job's list output. If **–l 0** is specified, all list output is suppressed. The default is **stdout.**
–b *bdn*	Specifies the file receiving the binary load modules generated by the compiler; the default is **a.o.**
–o *list*	Specifies compiler options, separated by commas, in effect at the beginning of the compilation. The compiler options available under UNICOS are generally the same as the compiler options available under COS.
–C *list*	Specifies the characteristics of the Cray computer system for which Pascal is to generate code. The format of the **–c** *list* is as follows:

–c [*primary*] { ","*characteristic* }
primary can be one of the following:

Target Machine	Description
CRAY-X4	Generates code for a four-processor CRAY X-MP
CRAY-X2	Generates code for a dual-processor CRAY X-MP
CRAY-X1	Generates code for a single-processor CRAY X-MP
CRAY X-MP	Generates code for a single-processor CRAY X-MP that also runs on four-processor and dual processor CRAY X-MPs
CRAY-1M	Generates code for a CRAY-1 Model M
CRAY-1S	Generates code for a CRAY-1 Model S
CRAY-1B	Generates code for a CRAY-1 Model B
CRAY-1A	Generates code for a CRAY-1 Model A
CRAY-1	Generates code for a CRAY-1 Model A that also runs on CRAY-1 Models B, S, and M

characteristic can be one of the following traits:

Trait	Description
EMA	Causes Pascal to generate 24-bit A register immediate load instructions, where necessary, and allows the use of common blocks larger than 4 million words
NOEMA	Disables the generation of 24-bit A register immediate load instructions and disallows the use of common blocks larger than 4 million words
CIGS	Enables compressed index and gather/scatter
NOCIGS	Disables compressed index and gather/scatter
VPOP	Enables vector population and parity
NOVPOP	Disables vector length read instructions
READVL	Enables vector length read instructions
NOREADVL	Disables vector length read instructions
MEMSIZE	The format of the MEMSIZE option is as follows: MEMSIZE = *n* ["K" I "M"] MEMSIZE is *n* * 1024 words for *n*K and *n* * 1048575 words for *n*M
BDM	Enables bidirectional memory
NOBDM	Disables bidirectional memory

The –C *list* option cannot be specified with a CRAY-2 computer system.

–V If present, this parameter writes the version number of the compiler being run and other statistical information (such as compilation time, memory use, and the size of the relocatable output) to the standard error file, stderr.

COMPILER OPTIONS

Compiler directives placed inside comments in the Pascal program override the initial settings. The defaults on a CRAY-1 or a CRAY X-MP computer system running UNICOS are as follows:

```
A-,BP-,BREG=8,BT-,C-,DM0,E+,G-,L+,O+,P-,P24,R+,RV-,ST-,T+,TREG=8,U-,V+,W-,X-,Z+
```

The defaults on a CRAY-2 computer system are as follows:

```
A-,BP-,BREG=8,BT-,C-,DM0,E+,G-,L+,M2,O+,P-,P32,R+,RV-,ST-,T+,TREG=8,U-
,V+,W-,X-,Z+
```

SEE ALSO

The Pascal Reference Manual, publication SR-0060

NAME

passwd – Changes login password

SYNOPSIS

passwd [*name*]

DESCRIPTION

The *passwd* command changes or installs a password associated with the login *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. If the system administrator has entered the necessary encrypt string, the first time the new password is entered *passwd* checks to see if the old password has "aged" sufficiently, which means the password cannot be changed until a specific length of time has passed. If "aging" is insufficient, the new password is rejected and *passwd* terminates; see *passwd*(4F).

Assuming "aging" is sufficient, a check is made to ensure that the new password meets the construction requirements below. 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 at most two more times.

Passwords must meet the following requirements:

- Each password must have at least six characters. Only the first eight characters are significant.

- Each password must contain at least two alphabetic characters; and on CRAY-2 systems, each passwd must contain 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.

- The new password 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. NOTE: This is not recommended.

FILES

/etc/passwd

SEE ALSO

login(1), id(1), su(1)
crypt(3C) in the CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual, publication SR-2013
passwd(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

paste – Merges same lines of several files or subsequent lines of one file

SYNOPSIS

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). It can be considered the counterpart of *cat*(1) which concatenates vertically, that is, one file after the other. In the last form above, *paste* combines 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 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, that is, when exhausted, it is reused. In parallel merging (that is, 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 (for example,to get one backslash, use **–d "\\\\"**).

–s Merge subsequent lines rather than one from each input file. Use *tab* for concatenation, unless *list* is specified with **–d** option. Regardless of the *list*, the last character of the file is forced to be a new-line character.

– 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

MESSAGES

Line too long	Output lines are restricted to 511 characters.
Too many files	Except for the **–s** option, no more than 12 input files may be specified.

NOTES

pr –t –m... works similarly, but creates extra blanks, tabs, and new-lines for a nice page layout.

SEE ALSO

cut(1), grep(1), pr(1)

NAME

pg – Lets you control scrolling of files while perusing them

SYNOPSIS

pg [*files*...]

DESCRIPTION

The *pg* command is a filter that allows the examination of *files* one screenful at a time on a terminal. (If *file* is – or no arguments are specified, the *pg* command reads from the standard input.) The *pg* command prints 20 lines and then waits for a carriage return to be typed.

SEE ALSO

cat(1), pr(1), more(1)

NAME

plcopy – Converts COS PLs into UNICOS PLs

SYNOPSIS

plcopy [**–i** *inpath*] [**-o** *outpath*]

DESCRIPTION

Plcopy is a utility that converts COS PLs into UNICOS PLs under UNICOS. The COS PL must have the new *update*(1) format, which is any PL created or modified with update version 1.06 or newer.

The keyletters for *plcopy* are as follows:

–i *inpath*	Path and file name of the COS PL to be converted. This is a required keyletter.
–o *outpath*	Path and name of the file to receive the converted PL. This is a required keyletter.

NOTES

If the COS blank compression character in the PL is not octal 33, *plcopy* will not work.

To convert a PL in the old UNICOS PL format use the following:

PLTMP [–i *inpath*] [–o *outpath*]

–i *inpath*	Path and directory of UNICOS PL to be converted
–o *outpath*	Path and name of the file to receive the converted PL.

SEE ALSO

update(1)
fortfiles(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

pr – Prints files

SYNOPSIS

pr [*options*] [*files*]

DESCRIPTION

The *pr* command displays 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, only one column is displayed. If multiple columns are specified (using the **–k** option), the columns are by default 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 Suppress printing of diagnostic reports on failure to open files.

–t Suppress printing of the five-line identifying header and 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* Suspends messages

SEE ALSO

cat(1), pg(1)

NAME

premult – Invokes the premult preprocessor

SYNOPSIS

premult [**–m** *mdn*] [**–s** *sdn*] [**–c** *cdn*] [**–l**] *filename*

DESCRIPTION

The *premult* command invokes the premult microtasking preprocessor. The *premult* command interprets preprocessing directives and rewrites your program. The following options are available:

–m *mdn*	Specifies Fortran code that contains inserted microtasking primitives. If you do not specify the routine, *mdn* defaults to **multf.f.**
–s *sdn*	Specifies Fortran code containing no microtasking primitives. Microtasked subroutines also exist in a nonmicrotasked version so they can be called from a microtasked routine.
–c *cdn*	Specifies a CAL master routine for each microtasked subroutine. If you do not specify a routine, *cdn* defaults to **multc.s**
–l	Replaces the last character of an 8-character subroutine name with *s* or *m*. By default, the preprocessor creates two subroutines from each routine you microtask and appends to their names as much of **mult** and **sngl** as it can without making their names longer than 8 characters. For example, *joe* becomes *joemult* and *joesngl*; and *longjoe* becomes *longjoem* and *longjoes*. The *premult* command aborts if it finds an 8-character subroutine name in a program. It is your responsibility to ensure that subroutine names thus created are unique. *Premult* aborts if the original 8-character routine name ends in *s* or *m*.
filename	Specifies the file to be preprocessed. The filename must be supplied.

SEE ALSO

CRAY X-MP Multitasking Programmer's Manual, publication SN-0222.

NAME

printenv – Prints out the environment

SYNOPSIS

printenv [*name*]

DESCRIPTION

The *printenv* commands prints out the values of the variables in the environment. If you specify a *name*, *printenv* prints only its value.

If you specify a *name* and it is not defined in the environment, *printenv* returns exit status 1, otherwise it returns status 0.

SEE ALSO

sh(1), csh(1)

NAME

prof – Displays profile data

SYNOPSIS

prof [**–stz**] [**–m** *pdata*] [**–n** *nstars*] [*prog*]

DESCRIPTION

The *prof* command interprets a profile file produced by the prof library. The *prof* then reads the symbol table in the object file *prog* (**a.out** by default) and correlates it with the profile file *pdata* (**prof.out** by default). The *prof* command generates eight columns of data. These are: *label, label address, bucket address, secs, hits, %prof time, %actual time, histogram.* Addresses are all printed in parcel format with 0 == a, 2 == b, 4 == c, and 6 == d. Only buckets with at least one hit are printed. The histogram associates *nstars* (25 by default) stars with 100% in the *%prof time* column. The following options are accepted by *prof*:

–s Attempts to print subtotals for each label

–t Does not print the actual buckets (should be used in conjunction with **–s**).

–z Prints all symbols, even if no bucket is associated with them.

To get a program ready for profiling you must implement the *ld* options as indicated; first load the program with the **–l** *prof* option on the load line before the **–lc** option (see *ld*(1)). (When using *cc*(1), just add "-lprof" to the end of the *cc* line). When the program is invoked, profiling is automatically turned on. The data is written out when the program terminates successfully. (This means if the job aborts, no data is written out). There are three controllable parameters when profiling. These are: **PROF_WPB** (words per bucket), **PROF_SADDR** (starting parcel address), and **PROF_EADDR** (ending parcel address). If any of these exist as an environment variable (see *sh*(1)), then their value is converted (see *strtod(3)),* and used in place of the defaults.

The defaults are:

PROF_WPB == 0100
PROF_SADDR == 0
PROF_EADDR == end of text

BUGS

Currently the clock rate is 100HZ and cannot be changed.
Currently profiling only works on CPU-A

SEE ALSO

ld(1), sh(1)

NAME

prs – Prints an SCCS file

SYNOPSIS

prs [**–d**[*dataspec*]] [**–r**[*SID*]] [**–e**] [**–l**] [**–c**[*date-time*]] [**–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 name of **–** is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed; non-SCCS files and unreadable files are silently ignored.

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

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

–d[*dataspec*]	Used to specify the output data specification. The *dataspec* is a string consisting of SCCS file *data keywords* (see *DATA KEYWORDS*) interspersed with optional user supplied text.
–r[*SID*]	Used to specify the *S*CCS *ID*entification (SID) string of a delta for which information is desired. If no SID is specified, the SID of the most recently created delta is assumed. The format for the date is: mm/dd/yy [hh:mm:ss].
–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]]]]]
–c[*date-time*]	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 in the form: **"–c77/2/2 9:22:25"**.
–a	Requests printing of information for both removed deltas, that is, delta type = *R* (see *rmdel*(1)) and existing deltas, that is, delta type = *D*. If the **–a** keyletter is not specified, information for existing deltas only is provided.

DATA KEYWORDS

Data keywords specify which parts of an SCCS file are to be retrieved and output. All parts of an SCCS file (see *sccsfile*(4F)) 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.
A tab is specified by **\t** and carriage return/new-line is specified by **\n**. The default data keywords are:

":Dt:\t:DL:\nMRs:\n:MR:COMMENTS:\n:C:"

TABLE 1. SCCS Files Data Keywords

Keyword	*Data Item*	*File Section*	*Value*	*Format*
:Dt:	Delta information	Delta Table	See below*	S
:DL:	Delta line statistics	"	:Li:/:Ld:/:Lu:	S
:Li:	Lines inserted by Delta	"	nnnnn	S
:Ld:	Lines deleted by Delta	"	nnnnn	S
:Lu:	Lines unchanged by Delta	"	nnnnn	S
:DT:	Delta type	"	*D* or *R*	S
:I:	SCCS ID string (SID)	"	:R:.:L:.:B:.:S:	S
:R:	Release number	"	nnnn	S
:L:	Level number	"	nnnn	S
:B:	Branch number	"	nnnn	S
:S:	Sequence number	"	nnnn	S
:D:	Date Delta created	"	:Dy:/:Dm:/:Dd:	S
:Dy:	Year Delta created	"	nn	S
:Dm:	Month Delta created	"	nn	S
:Dd:	Day Delta created	"	nn	S
:T:	Time Delta created	"	:Th:::Tm:::Ts:	S
:Th:	Hour Delta created	"	nn	S
:Tm:	Minutes Delta created	"	nn	S
:Ts:	Seconds Delta created	"	nn	S
:P:	Programmer who created Delta	"	logname	S
:DS:	Delta sequence number	"	nnnn	S
:DP:	Predecessor Delta seq-no.	"	nnnn	S
:DI:	Seq-no. of deltas incl., excl., ignored	"	:Dn:/:Dx:/:Dg:	S
:Dn:	Deltas included (seq #)	"	:DS: :DS:...	S
:Dx:	Deltas excluded (seq #)	"	:DS: :DS:...	S
:Dg:	Deltas ignored (seq #)	"	:DS: :DS:...	S
:MR:	MR numbers for delta	"	text	M
:C:	Comments for delta	"	text	M
:UN:	User names	User Names	text	M
:FL:	Flag list	Flags	text	M
:Y:	Module type flag	"	text	S
:MF:	MR validation flag	"	*yes* or *no*	S
:MP:	MR validation pgm name	"	text	S
:KF:	Keyword error/warning flag	"	*yes* or *no*	S
:KV:	Keyword validation string	"	text	S
:BF:	Branch flag	"	*yes* or *no*	S
:J:	Joint edit flag	"	*yes* or *no*	S
:LK:	Locked releases	"	:R:...	S
:Q:	User defined keyword	"	text	S
:M:	Module name	"	text	S
:FB:	Floor boundary	"	:R:	S
:CB:	Ceiling boundary	"	:R:	S
:Ds:	Default SID	"	:I:	S
:ND:	Null delta flag	"	*yes* or *no*	S
:FD:	File descriptive text	Comments	text	M
:BD:	Body	Body	text	M
:GB:	Gotten body	"	text	M
:W:	A form of *what*(1) string	N/A	:Z::M:\t:I:	S
:A:	A form of *what*(1) string	N/A	:Z::Y: :M: :I::Z:	S
:Z:	*what*(1) string delimiter	N/A	@(#)	S
:F:	SCCS file name	N/A	text	S
:PN:	SCCS file path name	N/A	text	S

* :Dt: = :DT: :I: :D: :T: :P: :DS: :DP:

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:*

```
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?????

MESSAGES

Use *help*(1) for explanations.

SEE ALSO

admin(1), delta(1), get(1), help(1)
sccsfile(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

ps – Reports process status

SYNOPSIS

ps [**–edafl**] [**–c** *corefile*] [**–n** *namelist*] [**–t** *termlist*] [**–p** *proclist*] [**–u** *uidlist*] [**–g** *grplist*]

DESCRIPTION

The *ps* command prints certain information about active processes. Without options, information is printed only about processes associated with the terminal you are at. The output consists of a short listing containing only the process ID, terminal identifier, cumulative time, and the command name. Otherwise, the information that is displayed is controlled by the options.

Using lists as arguments, options 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	Prints information about all processes
–d	Prints information about all processes, except process group leaders
–a	Prints information about all processes, except process group leaders and processes not associated with a terminal
–f	Generates a *full* listing. (See below for meaning of columns in a full listing.)
–l	Generates a *long* listing. See below.
–c *corefile*	Uses the file *corefile* in place of **/dev/mem**
–n *namelist*	Takes *namelist* as the name of an alternative system *namelist* file in place of **/unicos.**
–t *termlist*	Restricts the listing to data about the processes associated with the terminals given in *termlist*. You may specify terminal identifiers in one of two forms: the device's file name (such as, tty04) or if the device's file name starts with tty, just the digit identifier (such as, 04).
–p *proclist*	Restricts the listing to data about processes whose process ID numbers are given in *proclist*.
–u *uidlist*	Restricts the listing to data about processes whose user ID numbers or login names are given in *uidlist*. In the listing, the numerical user ID is printed unless the **–f** option is used, in which case the login name is 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 (such as, for physical I/O)
10	Being swapped
20	Being traced by another process
40	Another tracing flag
100	Suspended
200	SIGCPULIMIT has been set

		1000 Connected to a CPU
		2000 Semaphore flag was set on disconnect
		4000 Process is being disconnected
S	(l)	The state of the process:
		S Sleeping
		W Waiting
		R Running
		I Intermediate
		Z Terminated
		T Stopped
UID	(f,l)	The user ID number of the process owner; the login name is printed under the –f option.
PID	(all)	The process ID of the process; it is possible to kill a process if you know this number.
PPID	(f,l)	The process ID of the parent process.
CM	(f,l)	CPU mask; used to limit a process to one or more specified CPUs.
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.
SIZE	(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 runnable.
TTY	(all)	The controlling terminal for the process.
TIME	(all)	The cumulative execution time for the process.
COMMAND	(all)	The command name.

A process that has exited and has a parent, but has not yet been waited for by the parent, is marked **<defunct>**.

FILES

/unicos	System namelist
/dev/mem	Memory
/etc/passwd	Supplies UID information
/etc/ps_data	Internal data structure
/dev	Searched to find terminal (''tty'') names

MESSAGES

Exit status is 0 if *ps* finds processes to report and 1 otherwise.

BUGS

Some data printed for defunct processes are irrelevant.

SEE ALSO

kill(1), nice(1), tty(1)

NAME

ps – Report status information about a process

SYNOPSIS

ps [**–edafl**] [**–t** *tlist*] [**–p** *proclist*] [**–g** *grplist*] [**–r** *rtime*] [**–R** *rtime*]

DESCRIPTION

The *ps* command prints information, such as process ID, terminal identifier, execution time, and the command name, about active processes. If you do not specify any options, information is printed about processes associated with the current terminal. Otherwise, the information that is displayed is controlled by the following options:

–e	Prints information about all processes
–d	Prints information about all processes, except process group leaders
–a	Prints information about all processes, except process group leaders and processes not associated with a terminal
–f	Generates a *full* listing. (Normally, a short listing containing only process ID, terminal ("tty") identifier, cumulative execution time, and the command name is printed.) See below for meaning of columns in a full listing.
–l	Generates a *long* listing. See below.
–t *tlist*	Restricts the listing to data about the processes associated with the terminals given in *tlist*, where *tlist* can be in one of two forms: a list of terminal identifiers separated from one another by a comma, or a list of terminal identifiers enclosed in double quotes and separated from one another by a comma and/or one or more spaces.
–p *proclist*	Restricts the listing to data about processes whose process ID numbers are given in *pro-clist*
–g *grplist*	Restricts the listing to data about processes whose process groups are given in *grplist*
–r *rtime*	Refreshes the current screen at *rtime* second intervals. (See the **–R** option for commands while in refresh mode.)
–R *rtime*	Refreshes and scrolls at *rtime* second intervals. The following commands are available while in refresh mode:

+	Go to next screen
-	Go to previous screen
>	Increment refresh interval by 1 second
<	Decrement refresh interval by 1 second
r	Refresh current screen
R	Refresh with scroll
Q (or ^c)	Quit display

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. If you generate a long listing, the following columns are printed:

F	(l)	Flags (octal and additive) associated with the process:
		01 In core
		02 System process
		04 Locked in core (such as, for physical I/O)
		10 Being swapped
		20 Being traced by another process
		40 Another tracing flag
		100 Connected to CPU
		200 Suspended for single threading
		2000 Suspended for deadlock
		4000 Suspended by user
		10000 CPU limit exceeded
S	(l)	The state of the process:
		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; you can kill a process if you know this number.
PPID	(f,l)	The process ID of the parent process.
CPU	(f,l)	CPU number
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 clicks of the core image of the process.
WCHAN	(l)	The event for which the process is waiting or sleeping; if 0, the process is runnable.
TTY	(all)	The controlling terminal for the process.
TIME	(all)	The cumulative execution time for the process.
COMMAND	(all)	The command name.

FILES

/dev Searched to find terminal (“tty”) names.

BUGS

Things can change while *ps* is running; the picture it gives is only a close approximation to reality. Some data printed for defunct processes are irrelevant.

SEE ALSO

kill(1), nice(1)

NAME

pwd – Prints working directory name

SYNOPSIS

pwd

DESCRIPTION

The *pwd* command prints the path name of the current (working) directory.

MESSAGES

Cannot open ..
or
Read error in ..

Indicates possible file system trouble and a user should try the full path name (cd */name/name/*) followed by a *pwd* command. If the problem persists, then contact a Cray Research, Inc., site analyst.

SEE ALSO

cd(1)

NAME

qdel – Deletes or signals NQS requests

SYNOPSIS

qdel [**–k**] [*–signo*] [**–u** *username*] *request-id* ...

DESCRIPTION

Qdel deletes or sends a signal to queued NQS requests specified on the command line. The **–k** flag, if included, sends the default signal of SIGKILL(-9) to running requests specified on the command line. This exits and deletes the receiving request. The *–signo* flag designates an alternate signal to SIGKILL. In the absence of the **–k** and *–signo* flags, *qdel* does not delete a running NQS request.

To delete or signal an NQS request, you must be the owner of the request, unless you have superuser privileges or are an NQS system administrator or system manager. If you are a user with these special privileges, use the **–u** flag to specify the owner of the request you wish to signal or delete.

The request-id of any NQS request is displayed when the request is first submitted (unless the silent mode of operation for the given NQS command was specified). You can also obtain the request-id of any request with the *qstat*(1) command.

LIMITATIONS

When you signal an NQS request, the signal is sent to all processes in the NQS request that are in the same process group. Whenever an NQS request is spawned, a new process group is established for all processes in the request. However, should one or more processes of the request successfully execute a *setpgrp*(2) system call, these processes do not receive signals sent by the *qdel*(1) command. This can lead to rogue request processes that must be killed by other means, such as the *kill*(1) command. NQS takes advantage of UNIX implementations that support the ability to lock a process into a single process group.

SEE ALSO

nqsintro(1), qdev(1), qlimit(1), qpr(1), qstat(1), qsub(1), qmgr(1M)
kill(2), setpgrp(2), and signal(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

qdev – Displays the status of NQS devices

SYNOPSIS

qdev [*device-name*] [*device-name@host-name* ...]

DESCRIPTION

Qdev displays the status of devices known to the Network Queuing System (NQS).

If no devices are specified, *qdev* displays the current state of each NQS device on the local host. Otherwise, the response is limited to the devices specified. Devices may be specified either as *device-name* or *device-name@host-name*. In the absence of a *host-name* specifier, *qdev* specifies the local host.

Qdev displays a device header with several headings for each selected device. The headings and their associated values are as follows:

Device:	Device name, in the form *device-name@host-name*
Fullname:	Full path name of the special file associated with the device
Server:	Command line used to execute the device server with *execve*(2)
Forms:	Forms configured with the device
Status:	General device state

The following paragraphs describe the two properties that represent the general device state.

The first property specifies whether the device will continue accepting queued requests. The following values are possible:

Enabled	Queued requests are accepted.
Disabled	Queued requests are not accepted and the device is idle.
Enabled/Closed	Queued requests are not accepted, but the device is not yet idle.

The second principal property of a device specifies whether the device is busy. The following values are possible:

Active	Device is busy.
Inactive	Device is idle and not known to be out of service.
Failed	Device is idle and is out of service (applies to both hardware and software failures).

If a device is busy, information about the active device follows the device header. The *qdev* command displays the request-name, request-id, and the name of the user who submitted the request.

SEE ALSO

nqsintro(1), qdel(1), qlimit(1), qpr(1), qstat(1), qsub(1), qmgr(1M)

NAME

qlimit – Shows supported batch limits and shell strategy for the named host or hosts

SYNOPSIS

qlimit [*host-name* ...]

DESCRIPTION

Qlimit displays the batch request resource limit types that NQS can directly enforce and the batch request shell strategy defined by the system administrator. Implementation of *qlimit* for systems other than the local host is deferred.

If you do not specify a *host-name*, *qlimit* displays information relevant only to the local host. Otherwise, *qlimit* displays the supported batch request limits and batch request shell strategy for each of the hosts you include on the command line.

NQS Resourse Limits

NQS supports many batch request resource limit types that can be applied to a batch request. However, not all UNIX implementations support the rather extensive set of NQS limit types.

The set of limits applied to a batch request is restricted to the set of limits that can be directly supported by the underlying UNIX implementation. If a batch request specifies a limit that cannot be enforced, the limit is ignored.

When you attempt to queue a batch request, each specified limit-value is compared against the limit-value configured for the destination batch queue. If the batch queue limit-value is greater than or equal to the corresponding batch request limit-value, the request can be successfully queued. If a request has a limit-value of infinity, the corresponding queue limit-value must also be infinity.

These resource limit checks are performed regardless of the arrival mechanism, either by direct use of the *qsub*(1) command, or by the indirect placement of a batch request into a batch queue through a pipe queue. An NQS batch request cannot be queued if any of the resource limit checks fail.

If a request does not specify a value for a resource limit type that is supported on the execution machine, the limit-value configured for the destination queue is the default.

When a request is successfully queued, the set of limits under which the request will execute is frozen.

NQS Shell Strategies

Qlimit also displays the shell strategy configured for the implied local host or named hosts. In the absence of a shell specification for a batch request, NQS chooses the shell used to execute that batch request. NQS supports three different algorithms, or strategies, to solve this problem. The NQS system administrator configures these shell strategies for each system, taking into account user needs and performance criteria.

The three possible shell strategies and their meanings follow:

Fixed	Specifies that the shell chosen by the system administrator is used to execute all batch requests
Free	The default NQS shell strategy. Specifies that the user's login shell (as defined in the password file) is executed; the login shell then chooses and spawns the appropriate shell to run the batch request. This shell strategy runs the request exactly as if the shell commands were entered interactively.
Login	Specifies that only the user's login shell is executed to interpret the script

The fixed and login strategies are appropriate for host systems short on available free processes. In these two strategies, a single shell is executed, and that same shell executes all commands in the batch request shell script.

When a system administrator configures a fixed shell strategy for a particular NQS system, *qlimit* displays the designated shell.

SEE ALSO

nqsintro(1), qdel(1), qdev(1), qpr(1), qstat(1), qsub(1), qmgr(1M)

NAME

qpr – Submits a hard-copy print request to NQS

SYNOPSIS

qpr [**–a** *date-time*] [**–f** *form-name*] [**–mb**] [**–me**]
[**–mu** *user-name*] [**–n** *number-of-copies*] [**–p** *priority*]
[**–q** *queue-name*] [**–r** *request-name*] [–z] [*files*]

DESCRIPTION

Qpr places files in an Network Queuing System (NQS) queue to be printed by a device such as a line printer or a laser printer. If no files are specified, *qpr* reads from standard input.

In the absence of the **–z** option, *qpr* prints a request-id on the standard output upon successful queuing of a request. To determine the status of your request, compare this request-id with *qdev*(1) and *qstat*(1). You can also use this request-id as an argument to *qdel*(1) to delete a request. A request-id takes the form *seqno.hostname,* where *seqno* refers to the sequence number NQS assigns to the request, and *hostname* refers to the name of originating local machine.

The following *qpr* options may appear in any order and may be mixed within file names.

–a *date-time*
: Submits at the specified date and time. In the absence of this option, *qpr* submits the request immediately.

  See *qsub(1)* for information on how to specify the *date-time* variable.

–f *form-name*
: Limits the set of acceptable devices to those devices loaded with the forms specified by *form-name*. In the absence of this option, *qpr* submits the request only to a device that is loaded with the default forms. If no default forms are defined, the request is submitted to the appropriate output device regardless of the forms configured for the device. In any case, only those devices associated with the chosen queue are considered.

–mb
: Sends mail to the user on the originating machine when the request begins execution. If the **–mu** option is also present, mail is sent to the user specified by the **–mu** option instead of to the invoking user.

–me
: Sends mail to the invoker on the originating machine when the request has ended execution. If the **–mu** option is also present, mail is sent to the user specified by the **–mu** option instead of to the invoking user.

–mu *user-name*
: Specifies that any mail concerning the request is delivered to the user *user-name*. *User-name* may be formatted either as *user* (containing no '@' character), or as *user@machine*. In the absence of this option, any mail concerning the request is sent to the invoker on the originating machine.

–n *number-of-copies*
: Specifies the number of copies printed; the default is 1.

–p *priority*
: Assigns an intraqueue priority to this request. The specified *priority* must be an integer in the range 0 through 127. A value of 127 defines the highest intraqueue request priority, while a value of 0 defines the lowest. This priority does not determine the execution priority of the request. Rather, it determines the relative ordering of requests within a queue.

  New requests on a queue are placed ahead of all existing requests of lower priority value. The existing request takes precedence if two requests are of equal priority.

NQS assigns a midrange default value if you do not specify an intraqueue priority to your request.

–q *queue-name*
: Specifies the queue to which the device request is submitted. If you do not designate a queue, NQS searches your **QSUB_QUEUE** environment variable and sends the request to the designated queue. If NQS does not locate **QSUB_QUEUE**, the request is sent to the default batch request queue designated by the system administrator; otherwise, the request cannot be queued.

–r *request-name*
: Assigns a name to this request. In the absence of an explict **–r** *request-name* specification, the default is the name of the first print file (leading path name removed) specified on the command line. If no print files were specified, **stdin** is the default request-name.

  If the request-name begins with a digit, the system prepends the character R to the request-name. All request-names are truncated to a maximum length of 15 characters.

  Be sure not to confuse *request-name* with *request-id*.

–z
: Submits the request silently. If the request is submitted successfully, nothing is written to **stdout** or **stderr**.

QUEUE ACCESS

NQS supports queue access restrictions. For each queue of queue type other than *network*, access may be either *unrestricted* or *restricted*. If access is *unrestricted*, any request may enter the queue. If access is *restricted*, a request can only enter the queue if the requester or the requester's login group has been given access to that queue (see *qmgr*(1M)). Requests submitted by root are an exception; they are always queued, even if root has not explicitly been given access.

Use *qstat*(1) to determine who has access to a particular queue.

SEE ALSO

mail(1), nqsintro(1), qdel(1), qdev(1), qlimit(1), qstat(1), qsub(1), qmgr(1M)

NAME

qstat – Displays the status of NQS queues

SYNOPSIS

qstat [–a] [–l] [–m] [–u *user-name* **] [–x]**
[*queue-name* ...] [*queue-name@host-name* ...]

DESCRIPTION

Qstat displays the status of Network Queueing System (NQS) queues.

If no queues are specified, the current state of each NQS queue on the local host is displayed. Otherwise, information is displayed for the specified queues only. Queues may be specified either as *queue-name* or *queue-name@host-name*. In the absence of a *host-name* specifier, the local host is assumed.

For each selected queue, *qstat* displays a *queue header* (information about the queue itself), followed by information about requests in the queue. Ordinarily, *qstat* shows only those requests belonging to the invoker. The following options are available:

-a	Shows all requests
-l	Requests are shown in a long format
-m	Requests are shown in a medium-length format
-u *user-name*	Shows only those requests belonging to *user-name*
-x	The queue header is shown in an extended format

The *queue header* always includes the queue-name, queue type, queue status (see following), an indication of whether or not the queue is *pipeonly* (accepts requests from pipe queues only), and the number of requests in the queue. An extended queue header goes on to display the priority and run limit of a queue, as well as the access restrictions, cumulative use statistics, server and destinations (if a pipe queue), queue to device mappings (if a device queue), and resource limits (if a batch queue).

By default, *qstat* displays the following information about a request: the *request-name*, the *request-id*, the owner, the relative request priority, and the current request state (see following). For running requests, the UNICOS job identifier group is also shown, as soon as this information becomes available to the local NQS daemon.

Qstat -m shows the following additional information: If the request was submitted with the constraint that it not run before a certain time and date, the constraining time and date is also displayed.

Qstat -l shows the time at which the request was created, an indication of whether or not mail will be sent, where mail will be sent, and the username on the originating machine. If a batch queue is being examined, resource limits, planned disposition of **stderr** and **stdout**, any advice concerning the command interpreter, and the umask value are shown. If a device queue is being examined, the requested forms are shown.

QUEUE STATE

The command displays a queue header with several headings for each of the selected queues. The first heading is followed by the name of the queue formatted as *queue-name@host-name*.

The second heading, which appears as **type=**, is followed by the queue type. The queue type is either BATCH (indicating a batch request queue), DEVICE (indicating a device queue), or PIPE (indicating a queue that pipes NQS requests to other NQS queues or machines).

The third heading prefaces a display of the general queue state. The general queue state is defined by two properties, which are described in the following paragraphs.

The first property determines whether requests can be submitted to the queue. The possible values follow:

ENABLED	Requests are accepted (enabled and the local NQS daemon is running).
DISABLED	Requests are not accepted.
CLOSED	Requests are not accepted (enabled with no local NQS daemon).

The second property indicates the following:

- Whether queued requests not currently running will be permitted to run after the completion of currently running requests
- Whether any requests are currently running in the queue

The following states are possible:

STOPPED	Queued requests not currently running are blocked from running, and no requests are currently in the queue.
STOPPING	Queued requests not currently running are blocked from running, but at least one job is running and will be allowed to complete execution.
RUNNING	Queued requests are permitted to run and one or more requests are currently running.
INACTIVE	Queued requests are permitted to run, but no requests are currently running.
SHUTDOWN	The NQS daemon for the local host on which the queue resides is not running.

Below the queue header, all requests residing in that queue are displayed in either short or long form, depending on the presence of the -l option. In both cases, however, the following information is displayed concerning each queued request:

- Queue position
- Execution state
- Relative priority compared to other requests in that queue
- Job name, job identification, and job owner

If the request is running, the process group of the request is displayed when this information is available to the local NQS daemon. If the request is being held until a specified time or date, the time or date is also displayed.

REQUEST STATE

The state of a request may be *arriving*, *holding*, *waiting*, *queued*, *staging*, *routing*, *running*, *departing*, or *exiting*. A request is *arriving* if it is being enqueued from a remote host. *Holding* indicates that the request is presently prevented from entering any other state (including the *running* state), because a *hold* has been placed on the request. A request is said to be *waiting* if it was submitted with the constraint that it not run before a certain date and time, and that date and time have not yet arrived. *Queued* requests are eligible to proceed (by *routing* or *running*). When a request reaches the head of a pipe queue and receives service there, it is *routing*. A request is *departing* from the time the pipe queue turns to other work until the request has arrived intact at its destination. *Staging* denotes a *batch* request that has not yet begun execution, but for which input files are being brought on to the execution machine. A *running* request has reached its final destination queue, and is actually executing. Finally, *exiting* describes a batch request that has completed execution, and it will exit from the system after the required output files have been returned (to possibly remote machines).

A batch request originating on a workstation and destined for the batch queue of a Cray computer system would go through the following states in a local pipe queue: *queued*, *routing*, and *departing*. It would then disappear from the pipe queue. From the point of view of a queue on the Cray computer system, the request would first be *arriving*, then *queued*, *staging* (if required by the batch request), *running*, and finally *exiting*. Upon completion of the *exiting* phase of execution, the batch request would disappear from the batch queue.

The ordering of requests within a queue does not always determine the order in which the requests are run; the NQS request scheduler can make exceptions to the request ordering for the sake of efficiency. Generally, however, requests appearing near the beginning of the queue have higher priority than requests appearing later, and early requests are usually run before requests appearing later in the queue.

SEE ALSO

nqsintro(1), qdel(1), qdev(1), qlimit(1), qpr(1), qsub(1), qmgr(1M)

NAME

qsub – Submits an NQS batch request

SYNOPSIS

qsub [*options*] [*script-file*]

DESCRIPTION

Qsub submits a batch request to the Network Queuing System (NQS). For an introduction to the use of NQS, see *nqsintro*(1).

If you do not specify the script file on the command line, the set of commands to be executed as a batch request is taken directly from the standard input file **stdin.** In all cases, however, the script file is immediately spooled so that later changes to the script file do not affect previously queued batch requests.

If your batch request is successfully submitted and the **–z** option was not specified, NQS displays the corresponding request-id. The request-id is a combination of the sequence number NQS assigns to your request and the name of the originating machine. For example, a request-id of 71.C2 specifies that NQS has assigned you a request sequence number of 71 and that you are working on a machine designated as C2. This identifier uniquely identifies your batch request throughout the network. The name of the queue to which the request is submitted is also displayed.

By default, NQS returns the output produced by your request (**stdout**) and any error messages (**stderr**) to the machine and directory from which you submitted the request. You can redirect both the **stderr** and **stdout** files with the options described later in this discussion.

You can include options within the script file, provided that the options appear before any of the shell commands executed as part of the batch request. The simplest method of including options within the script file is to precede the option or options with the #, @, and $ characters, as follows:

```
# @$-a 11pm tues.
```

This example specifies that the batch request should be submitted no earlier than 11 p.m. on Tuesday. The @, $, and - characters must not be separated by white space.

You can include a comment line within the script file by preceding it with a # character, as follows:

```
# This comment is not processed.
```

You can also include comments on the same line as an option by preceding the comment with a # sign:

```
# @$-a 11pm tues. # Sends job at 11 p.m. on Tuesday
```

Here is an extended example of the use of embedded options within the script file:

```
#
#  Batch request shell script example:
#
#  @$–q batch1 # Queue request to queue: batch1 by default.
#  @$–z # Submits the request silently
#  @$          # No more embedded options.
#
make all
```

If the same option appears both in the batch file and on the command line, the command line option takes precedence.

A summary of the options accepted by *qsub* follow:

FLAG	DESCRIPTION
–a	Runs request after stated time
–e	Directs standard error output to the stated destination
–eo	Directs standard error output to the standard output destination
–ke	Keeps standard error output on the execution machine
–ko	Keeps standard output output on the execution machine
–lf	Establishes per-process file-size limits
–lm	Establishes per-process memory size limits
–lM	Establishes per-request memory size limits
–ln	Establishes per-process nice execution value limits
–lt	Establishes per-process CPU time limits
–lT	Establishes per-request CPU time limits
–mb	Sends mail when the request begins execution
–me	Sends mail when the request ends execution
–mu	Sends mail for the request to the stated user
–nr	Specifies that the batch request is not restartable
–o	Directs standard output to the stated destination
–p	Specifies intra-queue request priority
–q	Queues requests in the stated queue
–r	Assigns stated request name to the request
–re	Remotely accesses the standard error output file
–ro	Remotely accesses the standard output output file
–s	Specifies shell to interpret the batch shell script
–x	Exports all environment variables with request
–z	Submits the request silently

Complete descriptions of the options follow:

–a *date-time*

Holds the batch request until the specified date and time. When you enter a *date-time* specification separated by white space on the command line, enclose the specification within double quotes, as follows: **–a** *"July 4, 2026 12:31-EDT"*. Double quotes cannot be used when the *date-time* specification appears within the script file.

You can specify a wide range of values for the *date-time* variable. If you do not specify a date, the default value is the current day, month, or year. As a result, if you do not include a date and the specified time is earlier than the current time, NQS processes the request immediately. You can also specify the date as a weekday (for example, Tuesday) or with the words today or tomorrow. Abbreviate weekdays and months by any three-or-more character prefix (for example, tues or feb). A period can optionally follow abbreviated months or days.

Specify the time of day using either the 24-hour clock or meridian (a.m. or p.m.) specifications. The 24-hour clock, which is the default, is specified in the form *hh:mm:ss*. For example, 12am on Friday is equivalent to 00:00:00 on Friday; 12n on Friday is equivalent to 12:00:00 on Friday; and 12pm on Friday is equivalent to 24:00:00 on Friday. The words midnight and noon are also accepted as time of day specifications; midnight is equivalent to 24:00:00 on the specified day.

You can also include a time zone designation in the *date-time* specification; the default is the local time zone. For example, April 1, 1987 13:01-EST specifies Eastern Standard Time. NQS accounts for daylight savings time, when appropriate.

Date and time specifications are not case-sensitive.

Some valid *date-time* examples follow:

```
01-Jan-1986 12am, PDT
Tuesday, 23:00:00
11pm tues.
tomorrow MST 23
```

–e *[machine:][[/]path/]stderr-filename*
Directs the **stderr** file produced by the batch request to the specified machine, path, and standard-error filename. The standard error file produced by the batch job is referred to as the **stderr** file.

If you do not specify *machine*, and the path/filename does not begin with a /, the current working directory is prepended to create a fully qualified path name, provided that the **–ke** option is absent. In all other cases, any partial path/filename is interpreted relative to your home directory on the **stderr** destination machine.

You cannot specify this option when the **–eo** option is also specified.

If the **stderr** filename is not specified with the **–eo** and **–e** options, all **stderr** output is sent to a file named as follows: the first 7 characters of the *request-name*, followed by the characters **.e**, followed by the request sequence number portion of the *request-id* discussed below. In the absence of the **–ke** option, the **stderr** file is placed in the current working directory of the originating machine. Otherwise, the **stderr** file is placed in your home directory on the executing machine.

–eo Directs all output that would normally be sent to the **stderr** file to the **stdout** file for the batch request. This option is not valid when the **–e** *[machine:][[/]path/] stderr-filename* option is present.

–ke Leaves the **stderr** file on the execution machine; normally, the **stderr** file is returned to the originating machine.

This option is invalid if the **–eo** option is specified or if an explicit *machine* destination is given for the **stderr** parameter of the **–e** option.

–ko Leaves the **stdout** file on the originating machine; normally, NQS returns the **stdout** file to the machine that originated the request.

Do not specify this option if an explicit *machine* destination is given for the **stdout** parameter of the **–o** option.

–lf *per-process file-size limit*
Set a *per-process* maximum for files associated with the batch request.

If a maximum limit specification is comprised of two or more tokens separated by whitespace, enclose the specification within double quotes or escape characters.

–lm *per-process memory size limit*
Specifies the maximum memory per process in a batch request.

See the LIMITS subsection for more information.

–lM *per-request memory space limit*
Specifies the maximum memory size allowed for a batch request. If the total memory used by a request exceeds the specified limit, processes in the request are not allowed to allocate more memory.

See the LIMITS subsection for more information.

–ln *per-process nice-value limit*
Specifies the priority by which the batch request is processed on the executing machine.

The nice-value, which exists on most UNIX implementations, determines the execution-time priority of a process relative to all other processes in the system.

Use the nice-value when you are executing a CPU-intensive batch request on a machine used by a significant number of interactive users. By setting a low execution-time priority, you can make a long-running batch request defer to higher-priority interactive processes during the day.

On most systems, increasingly negative nice-values increase the relative execution priority of a process; increasingly positive nice-values decrease the relative priority. For example, a specification of **–ln** -20 assigns a higher execution priority than **–ln** 20.

Because varying UNIX implementations support a different range of nice values, NQS lets you specify values that are outside the limits for the executing machine. In such cases, NQS binds the specified nice-value limit to a value within the necessary range.

Any nice-value specified by the use of this option must be acceptable to the batch queue in which the request is ultimately placed (see the LIMITS subsection for more information).

–lt *per-process CPU time limit*
Specifies the CPU time limit for all processes that constitute a batch request. If the CPU time limit is exceeded, the offending process is terminated.

Not all UNIX implementations support per-process CPU time limits. If the executing machine does not enforce this limit, the limit is ignored.

See the LIMITS subsection for more information on the implementation of batch request limits and for a description of the syntax of a *per-process CPU time limit*.

–lT *per-request CPU time limit*
Specifies the maximum cumulative CPU time limit allowed for a batch request. If the CPU time limit is exceeded, the request is terminated by the executing system.

See the LIMITS subsection for more information on the implementation of batch request limits and for a description of the syntax of a *per-request CPU time limit*.

–mb Sends mail to the user on the originating machine when the request begins execution. If the **–mu** option is also present, NQS sends mail to the user specified by **–mu** instead of to the invoking user.

–me Sends mail to the user on the originating machine when the request has ended execution. If the **–mu** option is also present, NQS sends mail to the user specified by **–mu** instead of to the invoking user.

–mu *user-name*
Specifies that any mail concerning the request is delivered to the user *user-name*. *User-name* may be formatted either as *user* (containing no '@' character) or *user@machine*. In the absence of this option, NQS sends any mail concerning the request to the invoker on the originating machine.

–nr Specifies that the request is nonrestartable; the request will not be restarted during system boot if the request was running at the time of an NQS shutdown or system crash.

In the absence of **–nr**, NQS assumes that all requests are restartable. However, you should ensure that the request will execute correctly if restarted.

Requests that are not running at the time of a host crash or shutdown are always preserved for later requeuing, with or without this option.

When an operator shuts down NQS, a **SIGTERM** signal is sent to the processes which comprise all running NQS requests on the local host; all queued NQS requests are also barred from beginning execution. After a predetermined time period, the processes comprising all remaining running NQS requests are killed by **SIGKILL**.

To ensure that an NQS request is properly restarted after an NQS shutdown, you must not specify the **–nr** option and the spawned batch request shell must ignore **SIGTERM** signals (which is done by default). Also, the spawned batch request shell must not exit before the final **SIGKILL** arrives. The commands and programs spawned by the batch request must be immune to **SIGTERM** signals, saving state as appropriate before being killed by the final **SIGKILL** signal.

See the LIMITATIONS subsection for more discussion concerning the restarting of NQS batch requests.

–o *[machine:][[/]path/]stdout-filename*

Directs standard output to the designated machine, path name, and **stdout** filename.

If you do not specify *machine*, the default is the machine that originated the batch request (unless the **–ko** option is also specified).

If you do not specify *machine* and do not supply a fully qualified path name, NQS assumes your current working directory (unless the **–ke** option is specified). In all other cases, any partial path/filename is interpreted relative to your home directory on the **stdout** destination machine.

By default, all standard output for the batch request is sent to a file named as follows: the first 7 characters of the request-name, followed by the characters **.o**, followed by the request sequence number portion of the *request-id*. In the absence of the **–ko** option, NQS returns this output file to the submitting machine in the directory from which you submitted the batch request. Otherwise, the file is placed in your home directory on the execution machine.

–p *priority*

Assigns an intraqueue priority to the request. The specified *priority* must be an integer in the range 0 through 127, with 127 representing the highest priority. This priority does not determine the execution priority of the request, but rather determines ordering of requests within a queue.

New requests on a queue are placed ahead of all existing requests of lower priority value. The existing request takes precedence if two requests are of equal priority.

NQS assigns a midrange default value if you do not specify an intraqueue priority.

–q *queue-name*

Specifies the queue to which the batch request is submitted. If you do not designate a *queue-name*, NQS searches your environment variable set for **QSUB_QUEUE** and sends the request to the designated queue. If the **QSUB_QUEUE** environment variable is not found, the request is sent to the default batch request queue designated by the system administrator.

–r *request-name*

Assigns the specified *request-name* to the request. If you do not specify *request-name*, the default is the name of the *script file* (leading path name removed) given on the command line.

If you did not specify a script file, the default *request-name* is **stdin**. The request-name cannot begin with a digit.

–re Specifies that the **stderr** output produced by the request is written to the final destination file as output is generated, regardless of the networking cost. This option is deferred for remote machines.

By default, all standard error output generated by a request is spooled to a temporary file in a protected directory which NQS maintains on the executing machine. When the batch request completes execution, this file is spooled to its final destination, possibly on a remote machine. This default spooling reduces the network traffic costs; the **–re** option overrides this default.

–ro Specifies that the standard output produced by the request is written to the destination file as output is generated, regardless of the networking cost. This option is deferred for remote machines.

By default, all standard output generated by a batch request is spooled into a temporary file residing in a protected directory on the executing machine. When the batch request completes execution, this file is spooled to its final destination. This default spooling reduces the network traffic costs; the **–ro** option overrides this default.

–s *shell-name*
Specifies the absolute path name of the shell used to interpret the batch request shell script. This option overrides any shell strategy configured on the execution machine.

In the absence of this option, the NQS system at the executing machine uses one of three shell strategies for batch request execution. See *qlimit*(1) for further information on these shell strategies.

–x Exports all environment variables.

When you submit a batch request, NQS saves the current values of the environment variables **HOME**, **SHELL**, **PATH**, **LOGNAME**, **MAIL**, and **TZ** for later re-creation as the respective environment variables **QSUB_HOME**, **QSUB_SHELL**, **QSUB_PATH**, **QSUB_LOGNAME**, **QSUB_MAIL**, and **QSUB_TZ**. Unless you specify the **–x** option, no other environment variables are be exported from the originating host for the batch request. If you specify the **–x** option, all remaining environment variables whose names do not conflict with the automatically exported variables are also exported. These additional environment variables are re-created under the same name when the batch request is spawned.

–z Submits the batch request silently. If the request is submitted successfully, no indicating messages are displayed. Error messages, however, are still displayed.

Sequence of Events

The following events occur in the specified order when you submit an NQS batch request:

NQS creates the process that will become the process group for all processes comprising the batch request.

NQS sets the real and effective user-id of the process to your user-id.

NQS sets the real and effective group-id of the process to the group-id defined in your local password file.

NQS sets the user file creation mask to the value existing on the originating machine when you submitted the batch request.

If you used the **–s** option, NQS selects the specified shell to execute the batch request shell script. Otherwise, a shell is chosen based on the shell strategy as configured for the local NQS system.

NQS sets the environment variables **HOME, SHELL, PATH, LOGNAME,** and **MAIL** from your password file entry as though you have logged on directly to the executing machine.

NQS adds the environment string **ENVIRONMENT=BATCH** to the environment so that shell scripts (and your **.profile** or **.cshrc** and **.login** files) can test for batch request execution when appropriate.

NQS adds the environment variables **QSUB_WORKDIR, QSUB_HOST, QSUB_REQNAME,** and **QSUB_REQID** to the environment. These environment variables equate to the following: the obvious respective strings of the working directory at the time that the request was submitted; the name of the originating host; the name of the request; and the request *request-id*.

NQS adds all remaining environment variables saved for re-creation when the batch request is spawned. When a batch request is initially submitted, the current values of the environment variables **HOME, SHELL, PATH, LOGNAME, MAIL,** and **TZ** are saved for this purpose. When they are recreated, however, NQS adds these variables to the environment under the following names: **QSUB_HOME, QSUB_SHELL, QSUB_PATH, QSUB_LOGNAME, QSUB_MAIL,** and **QSUB_TZ.** NQS also adds at this time all environment variables exported from the originating host by the **–x** option. This renaming eliminates conflict with the local versions of these environment variables.

NQS sets the current working directory to the home directory on the executing machine. The chosen shell then executes the batch request shell script with the environment constructed as outlined in the preceeding steps.

In all cases, NQS execs the chosen shell as though it were the login shell. If the Bourne shell executes the script, the **.profile** file is read. If the C-shell executes, the **.cshrc** and **.login** scripts are read.

If you did not specify a specific shell for the batch request, NQS chooses a shell script based on the shell strategy configured by the system administrator. See *qlimit*(1) for a discussion of shell strategies.

In the absence of a specified shell, a free shell strategy instructs NQS to execute your login shell (as configured in the password file). The login shell examines the shell script file and forks another shell of the appropriate type to interpret the shell script, behaving exactly as an interactive invocation of the script.

Otherwise, no additional shell is spawned, and the chosen fixed or login shell sequentially executes the commands contained in the shell script file until completion of the batch request.

Queue Access

NQS supports queue access restrictions. For each queue of a type different than network, access may be either restricted or unrestricted. If access is unrestricted, any request may enter the queue. If access is restricted, a request can enter the queue only if the requester or the requester's login group has access to that queue. Use *qstat*(1) to determine who has access to a particular queue.

Limits

NQS supports configurable limits that let you set resource limits within which your request executes. These limits also let system administrators set queue-specific resource limits to which all batch requests must adhere.

Finite limits are either time-related or nontime-related. Use the following syntax to specify the limit value for finite CPU time-limits:

[[*hours* :] *minutes* :] *seconds* [*.milliseconds*]

White space can appear anywhere between the principal tokens, with the exception that no white space can appear around the decimal point.

Example time *limit-values* are as follows:

1234 : 58 : 21.29	1234 hours, 58 minutes, and 21.290 seconds
12345	12,345 seconds
121.1	121.100 seconds
59:01	59 minutes and 1 second

For all other finite limits (with the exclusion of the nice limit-value), use the following syntax:

.fraction [units]

or

integer [.fraction] [units]

where the integer and fraction tokens represent strings of up to 8 decimal digits, denoting the obvious values. In both cases, the *units* of allocation may also be specified as one of the case insensitive strings:

b	Bytes
w	Words
Kb	Kilobytes (2^10 bytes)
Kw	Kilowords (2^10 words)
Mb	Megabytes (2^20 bytes)
Mw	Megawords (2^20 words)
Gb	Gigabytes (2^30 bytes)
Gw	Gigawords (2^30 words)

In the absence of any units specification, the units of bytes are assumed.

For all limit types, with the exception of the nice limit-value, you can designate an unlimited value by specifying a limit-value of "unlimited," or any initial substring thereof.

If a batch request specifies a limit that cannot be enforced by the underlying UNIX implementation, the limit is ignored, and the batch request operates as though no limit exists. *Qmgr*(1M) describes in detail how NQS limits are interpreted by various architectures.

Limitations

When an NQS batch request is spawned, a new job is established. Subsequently, all processes of the request exist in the same job. If you use *qdel*(1) to send a signal to an NQS batch request, the signal is sent to all processes of the request in the created job.

All processes of an NQS request should catch any **SIGTERM** signals. By default, the receipt of a **SIGTERM** signal causes the receiving process to die. NQS sends a **SIGTERM** signal to all processes in the established process group for a batch request as a notification that the request should be prepared to be killed. This signal is sent by a *qmgr*(1) *abort queue* command or because of a general host shutdown.

The spawned shell ignores **SIGTERM** signals. If the current immediate child of the shell does not ignore or catch **SIGTERM** signals, it is killed by the receipt of such, and the shell goes on to execute the next command from the script (if there is one). In any case, the shell is not killed by the **SIGTERM** signal, although the executing command is killed.

After receiving a **SIGTERM** signal delivered from NQS, a process of a batch request typically has 20 seconds before receiving a **SIGKILL** signal.

NQS considers all batch requests terminated because of an operator NQS shutdown request restartable, unless the batch request specified the **-nr** option. (Implementation of the **-nr** option is deferred.) When NQS is rebooted following a shutdown, eligible batch requests are requeued, provided that the batch request shell process is still present at the time of the **SIGKILL** signal broadcast. It is up to you, however, to ensure that the request is restartable.

There is no good method to echo commands executed by unmodified versions of the Bourne and C shells. The C shell can be spawned so as to echo the commands it executes. It is, however, difficult to tell an echoed command from genuine output produced by the batch request, because no identifying character is displayed before the echoed command.

Thus, an efficient method of writing shell scripts for a batch request is to place appropriate lines in the shell script, as follows:

```
echo "explanatory-message"
```

SEE ALSO

mail(1), nqsintro(1), qdel(1), qdev(1), qlimit(1), qpr(1), qstat(1)
qmgr(1M) in the UNICOS Administrator Commands Reference Manual, publication SR-2022

NAME

rcp – Copies remote files

SYNOPSIS

rcp *file1 file2*
rcp [**–r**] *file* ... *directory*

DESCRIPTION

The *rcp* command copies files between machines. *File* arguments may refer to remote files, local files, or directories; arguments may consist of either absolute or relative path names. Remote files are specified in the form *rhost*:*file*, where *rhost* is a remote hostname or alias (described in *hosts(4F)*). The local file name *file* may not contain a colon (:) unless it is preceded anywhere in the name by a slash (/).

If the **–r** option is specified and any of the source files are directories, *rcp* copies each subtree rooted at that name; in this case the destination must be a directory.

If *path* is not a full path name, it is interpreted relative to your login directory on *rhost*. A path name on a remote host may be quoted (using \, ", or ´) so that metacharacters are interpreted remotely.

Rcp does not prompt for passwords; your current local user name must exist on *rhost* and allow remote command execution via *remsh(1)*.

Rcp handles third party copies, where neither source nor target files are on the current machine. Hostnames may also take the form *rhost.rname* to use *rname* rather than the current user name on the remote host.

BUGS

Rcp does not detect all cases where the target of a copy might be a file when only a directory should be legal.
Rcp is confused by any output generated by commands in a *.profile* file on the remote host.

SEE ALSO

ftp(1), remsh(1), rlogin(1)

NAME

rdist – Remote file distribution program

SYNOPSIS

rdist [**–DnqbRvwyhimr**] [**–f** *distfile*] [**–d** *var=value*] [*name* ...]

rdist [**–DnqbRvwyhimr**] **-c** *name* ... *host*[*.login*] [*:dest*]

DESCRIPTION

The *rdist* command maintains identical copies of files over multiple hosts. It preserves the owner, group, mode, and mtime of files if possible. It can update executing programs. The *rdist* command reads the commands from *distfile* to update files and/or directories. If *distfile* is '–', the standard input is used. If you do not specify **–f**, *distfile* is used for input. If you do not specify any *names* on the command line, *rdist* updates all of the files and directories listed in *distfile*; otherwise, *rdist* only updates the listed files. The following options are available:

–c Forces *rdist* to interpret the remaining arguments as a small *distfile*. The equivalent distfile is as follows.

```
( name ... ) -> host[.login]
        install   dest ;
```

–d Defines *var* to have *value*. Use the **–d** option to define or override variable definitions in the *distfile*. *Value* can be the empty string, one name, or a list of names surrounded by parentheses and separated by tabs and/or spaces.

–n Prints the commands without executing them. This option is useful for debugging *distfile*.

–q Invokes quiet mode. Files that are being modified are normally printed on standard output. The **–q** option suppresses this.

–R Removes extraneous files. If a directory is being updated, any files that exist on the remote host that do not exist in the master directory are removed. This is useful for maintaining truely identical copies of directories.

–v Verifies that the files are up to date on all the hosts. Any files that are out of date will be displayed but no files will be changed nor any mail sent.

–m Invokes force directory mode. This will force an update of all remote directories into having the same permissions as the master directory.

–r Follows symbolic links on the remote host. Normally a symbolic link will be overwritten if the master isn't a symbolic link.

–w Invokes whole mode. The whole file name is appended to the destination directory name. Normally, only the last component of a name is used when renaming files. This will preserve the directory structure of the files being copied instead of flattening the directory structure.

–y Invokes younger mode. Files are normally updated if their *mtime* and *size* (see *stat*(2)) disagree. The **–y** option causes *rdist* to only update files that are younger than the master copy. This can be used to prevent newer copies on other hosts from being replaced. A warning message is printed for files which are newer than the master copy.

–h Invokes follow symbolic links. Force *rdist* to install copies of the file pointed to instead of installing this symbolic link.

–i Invokes ignore links. This causes *rdist* to ignore all hard links. Normally they are installed.

-D Invokes debug switch. Rdist will output debugging information. This switch is only useful if you need to debug rdist.

–b Invokes binary comparison. Perform a binary comparison and update files if they differ rather than comparing dates and sizes.

Distfile contains a sequence of entries that specify the files to be copied, the destination hosts, and what operations to perform to do the updating. Each entry has one of the following formats.

```
<variable name> '=' <name list>
<source list> '–>' <destination list> <command list>
<source list> '::' <time_stamp file> <command list>
```

The first format defines variables. The second format distributes files to other hosts. The third format makes lists of files that have been changed since some given date. The *source list* specifies a list of files and/or directories on the local host that are to be used as the master copy for distribution. The *destination list* is the list of hosts these files are to be copied to. Each file in the source list is added to a list of changes if the file is out of date on the host being updated (second format) or the file is newer than the time stamp file (third format).

Newlines, tabs, and blanks are only used as separators and are otherwise ignored. Comments begin with '#' and end with a newline.

The source and destination lists have the following format:

```
        <name>
or
        '(' <zero or more names separated by white-space> ')'
```

The shell meta-characters '[', ']', '{', '}', '*', and '?' are recognized and expanded (on the local host only) in the same way as *csh*(1). The '~' character is also expanded in the same way as *csh* but is expanded separately on the local and destination hosts. When the –w option is used with a file name that begins with '~', everything except the home directory is appended to the destination name.

The *command list* consists of zero or more commands of the following format.

```
'install'  <options>  opt_dest_name ';'
'notify'  <name list> ';'
'except'  <name list> ';'
'special' <name list> string ';'
```

install copies out of date files and/or directories. Each source file is copied to each host in the destination list. Directories are recursively copied in the same way. *Opt_dest_name* is an optional parameter to rename files. If no *install* command appears in the command list or the destination name is not specified, the source file name is used. Directories in the path name will be created if they do not exist on the remote host. The options are '–R', '–v', '–w', '–y', and '–b' and have the same semantics as options on the command line except they only apply to the files in the source list. The login name used on the destination host is the same as the local host unless the destination name is of the format ''host.login".

Notify mails the list of files updated (and any errors that may have occured) to the listed names. If no '@' appears in the name, the destination host is appended to the name (e.g., name1@host, name2@host, ...).

Except updates all the files in the source list **except** for the files listed in *name list*. This is mostly used to copy everything in a directory except certain files.

Special specifies shell commands that are to be executed on the remote host after the file in *name list* is updated or installed. *String* starts and ends with '"' and can cross multiple lines in *distfile*. Multiple commands to the shell should be separated by ';'. The *special* command can be used to create links, rebuild private databases, and so on. after a program has been updated.

EXAMPLE

```
HOSTS = ( matisse arpa.root )

FILES = ( /bin /lib /usr/bin /usr/games
        /usr/include/{*.h,{stand,sys,vax*,pascal,machine}/*.h}
        /usr/lib /usr/man/man? /usr/ucb /usr/local/rdist )

EXLIB = ( Mail.rc aliases aliases.dir aliases.pag crontab dshrc
        sendmail.cf sendmail.fc sendmail.hf sendmail.st uucp vfont )

${FILES} -> ${HOSTS}
        install -r ;
        except /usr/lib/${EXLIB} ;
        except /usr/games/lib ;
        except /usr/ucb/f ;
        special /usr/ucb/finger "rm /usr/ucb/f; ln /usr/ucb/finger /usr/ucb/f" ;

IMAGEN = (ips dviimp catdvi)
/usr/local/${IMAGEN} -> arpa
        install /usr/local/lib ;
        notify ralph ;

${FILES} :: stamp.cory
        notify root@cory ;
```

FILES

distfile	Input command file
/tmp/rdist*	Temporary file for update lists

BUGS

Source files must reside on the local host where *rdist* is executed.

The names used to update specific files from *distfile* must match the expanded name (that is, *rdist* "${FILES}" will not work).

SEE ALSO

csh(1)
stat(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

rdrop – Reloads a recoverable drop file

SYNOPSIS

rdrop [**–l**] *pid*

DESCRIPTION

The *rdrop* command reloads a recoverable drop file from either the **system/drop** device or the current working directory. The **–l** option specifies the location from which the image is to be reloaded. If you do not specify **-l**, the image is always reloaded from the system/**drop** directory. The *pid* argument is the process id embedded in the drop file name. The format is: **drop.** *XXXXXXX*, where *XXXXXXX* is the process ID. The *pid* argument is required.

SEE ALSO

wdrop(1)
wdrop(2), rdrop(2) in the UNICOS Systems Calls Manual, publication SR-2012

NAME

regcmp – Compiles regular expressions

SYNOPSIS

regcmp [–] *files*

DESCRIPTION

Regcmp, in most cases, precludes the need for calling *regcmp*(3C) from C programs. This saves on both execution time and program size. The command *regcmp* compiles the regular expressions in *file* and places the output in *file*.i. If the – option is used, the output will be placed in *file*.c. The format of entries in *file* is a name (C variable) followed by one or more blanks followed by a regular expression enclosed in double quotes. The output of *regcmp* is C source code. Compiled regular expressions are represented as **extern char** vectors. *File*.i files may thus be *included* into C programs, or *file*.c files may be compiled and later loaded. In the C program which uses the *regcmp* output, *regex*(*abc*,*line*) will apply the regular expression named *abc* to *line*.

EXAMPLES

```
name    "([A–Za–z][A–Za–z0–9_]*)$0"
telno   "\({0,1}([2–9][01][1–9])$0\){0,1} *"
        "([2–9][0–9]{2})$1[ –]{0,1}"
        "([0–9]{4})$2"
```

In the C program that uses the *regcmp* output,

```
regex(telno, line, area, exch, rest)
```

will apply the regular expression named *telno* to *line*.

SEE ALSO

regcmp(3C)

NAME

remsh – Invokes a remote shell

SYNOPSIS

remsh *host* [**–l** *username*] [**–n**] *command*

DESCRIPTION

Remsh connects to the specified *host* and executes the specified *command. Remsh* copies its standard input to the remote command, the standard output of the remote command to its standard output, and the standard error of the remote command to its standard error. Interrupt, quit and terminate signals are propagated to the remote command; *remsh* normally terminates when the remote command does.

The remote *username* used is the same as your local *username*, unless you specify a different remote name with the **–l** option. This remote name must be equivalent (in the sense of *rlogin(1)*) to the originating account; no provision is made for specifying a password with a command.

If you omit *command,* instead of executing a single command, you will be logged in on the remote host using *rlogin*(1).

Shell metacharacters that are not quoted are interpreted on local machine, while metacharacters that are are quoted are interpreted on the remote machine. Thus, the command

```
remsh otherhost cat remotefile >> localfile
```

appends the remote file *remotefile* to the localfile *localfile,* while

```
remsh otherhost cat remotefile ">>" otherremotefile
```

appends *remotefile* to *otherremotefile.*

Host names are given in *hosts(4F).* Each host has one standard name (the first name given in the file), which is rather long and unambiguous, and, optionally, one or more nicknames. hosts(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

BUGS

If no input is desired, you must redirect the input of *remsh* to **/dev/null** using the **–n** option.

SEE ALSO

rlogin(1) - deferred implementation

NAME

rlogin – Invokes the remote login

SYNOPSIS

rlogin *rhost* [**–e***c*] [**–l** *username*] [**–8**]

DESCRIPTION

Rlogin connects your terminal on the current local host system *lhost* to the remote host system *rhost*. Your remote terminal type is the same as your local terminal type (as given in your environment TERM variable). All echoing takes place at the remote site, so that the *rlogin* is transparent (except for delays). Flow control via control-S and control-Q and flushing of input and output on interrupts are handled properly. A line of the form ~. disconnects from the remote host, where ~ is the escape character. A line of the form ~z suspends the *rlogin* process and creates a local shell.

The **–8** option allows the transmission of 8 bit data. A different escape character may be specified by the **–e** option. There is no space separating this option flag and the argument character.

BUGS

More terminal characteristics should be propagated.

SEE ALSO

remsh(1)

NAME

rm, rmdir – Removes files or directories

SYNOPSIS

rm [**–fri**] *file* ...
rmdir *dir* ...

DESCRIPTION

The *rm* command removes the entries for one or more files from a directory. If the removed entry is the last link to the file, the file is destroyed. Removal of a file requires write permission in its directory, but neither read nor write permission on the file itself.

If a file has no write permission and the standard input is a terminal, its permissions are printed and a line is read from the standard input. If that line begins with **y** the file is deleted, otherwise the file remains. No questions are asked if the standard input is not a terminal.

If a designated file is a directory, an error comment is printed.

Options are:

–f	Suppress questions on removal of file with no write permission
–r	Recursively delete entire contents of specified directory and directory itself
–i	Ask whether to delete each file; when used with the **-r** option, ask whether to examine each directory.

Rmdir removes entries for the named directories. The named directories must be empty.

MESSAGES

Error messages are generally self-explanatory.

NOTES

The file .. cannot be removed, which avoids consequences of inadvertently doing something like:

```
rm –r .*
```

SEE ALSO

unlink(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

rmdel – Removes a delta from an SCCS file

SYNOPSIS

rmdel –r*SID files*

DESCRIPTION

The *rmdel* command 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 specified delta must *not* be that of a version being edited for the purpose of making a delta (that is, if a *p-file* (see *get*(1)) exists for the named SCCS file, the specified delta 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 (SCCS) User's Guide. Simply stated, they are either (1) if you make a delta you can remove it; or (2) if you own the file and directory you can remove a delta.

FILES

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

MESSAGES

Use *help*(1) for explanations.

SEE ALSO

delta(1), get(1), help(1), prs(1)
sccsfile(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014
the UNICOS Source Code Control System (SCCS) User's Guide, publication SG-2017

NAME

ruptime – Shows host status of local machines

SYNOPSIS

ruptime [**–a**] [**–l**] [**–t**] [**–u**]

DESCRIPTION

For each machine on the local network, *ruptime* prints its name, the length of time it has been up, and the average number of processes in the run queue over the last fifteen minutes. These are formed from packets broadcast by each host on the network once a minute.

Machines for which no status report has been received for five minutes are shown as being down.

Users who have been idle for an hour or more are not counted unless the **–a** flag is given.

Normally, the listing is sorted by host name. The **–l** , **–t** , and **–u** flags specify sorting by load average, uptime, and number of users, respectively.

FILES

/usr/spool/rwho/whod.*	data files

SEE ALSO

rwho(1) - deferred implementation

NAME

rwho – Indicates who is logged in on local machines

SYNOPSIS

rwho [**–a**]

DESCRIPTION

The *rwho* command produces output similar to *who(1)*, but displays information for all machines on the local network. If no report has been received from a machine for five minutes, *rwho* assumes the machine is down and does not report the status for the users last known to be logged into that machine.

If a user has not sent any characters to the system for a minute or more, *rwho* reports this as idle time. If a user has not sent any characters to the system for an hour or more, this user will be omitted from the output of *rwho* unless the **–a** flag is given.

FILES

/usr/spool/rwho/whod.*	information about other machines

BUGS

Rwho is unwieldy when the number of machines on the local net is large.

SEE ALSO

ruptime(1) - deferred implementation, who(1), rwhod(1M) - deferred implementation

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 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 (that is, 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.

MESSAGES

Use *help*(1) for explanations.

SEE ALSO

delta(1), get(1), unget(1)
the UNICOS Source Code Control System (SCCS) User's Guide, publication SG-2017

NAME

sag – System activity graph

SYNOPSIS

sag [**–s** *time*] [**–e** *time*] [**–i** *sec*] [**–f** *file*] [**–T** *term*] [**–x** *spec*] [**–y** *spec*]

DESCRIPTION

The *sag* command graphically displays the system activity data stored in a binary data file by a previous *sar*(1) run. You can plot any of the *sar* data items singly or in combination, as cross plots or versus time; simple arithmetic data combinations can be specified. The *sag* command invokes *sar* and finds the desired data by string-matching the data column header (run *sar* to see what is available). These following options are passed to *sar*:

–s *time* Selects data later than *time* in the form *hh*[:*mm*]. The default is 08:00.

–e *time* Selects data up to *time*. The default is 18:00.

–i *sec* Selects data at intervals as close as possible to *sec* seconds.

–f *file* Uses *file* as the data source for *sar*. The default is the current daily data file **/usr/adm/sa/sa***dd*.

Other options:

–T *term* Produces output suitable for terminal *term*. If *term* is **vpr**, output is processed by **vpr –p** and queued to a Versatec printer. The default for *term* is **$TERM**.

–x *spec* x axis specification with *spec* in the form shown under the –y option.

–y *spec* y axis specification with *spec* in the form:

"*name* [*op name*] . . . [*lo hi*]"

Name is either a string that matches a column header in the *sar* report, with an optional device name in square brackets (for example, r+w/s[dsk–1]) or an integer value. *Op* is + – * or / surrounded by blanks. You can specify up to five names. *Sag* does not recognize parentheses. Contrary to custom, + and - have precedence over * and . Evaluation is left to right. Thus A / A + B * 100 is evaluated (A/(A+B))*100, and A + B / C + D is (A+B)/(C+D). *Lo* and *hi* are optional numeric scale limits. If unspecified, they are deduced from the data.

A single *spec* is permitted for the x axis. If unspecified, *time* is used. Up to 5 *spec*s separated by ; may be given for –y. Enclose the –x and –y arguments in " " if blanks or \<CR> are included. The –y default is:

```
–y "%usr 0 100; %usr + %sys 0 100; %usr + %sys + %wio 0 100"
```

EXAMPLES

To see today's CPU utilization:

```
sag
```

To see activity over 15 minutes of all disk drives:

```
TS=date +%H:%M
sar –o tempfile 60 15
TE=date +%H:%M
sag –f tempfile –s $TS –e $TE –y "r+w/s[dsk]"
```

FILES

/usr/adm/sa/sa*dd*	Daily data file for day *dd*.

BUGS

Sag produces a command file consisting of graphic commands and several data files. These must be used on a machine that supports plots or graphs, such as a VAX. The scaling produced by *sag* is not correct for CRAY X-MP and CRAY-1 computer systems.

SEE ALSO

sar(1)

NAME

sar – Extracts operating system activity information according to a specified time interval

SYNOPSIS

sar [**–upbdycwaqvA**] [**–o** *file*] *t* [*n*]
sar [**–upbdycwaqvA**] [**–s** *time*] [**–e** *time*] [**–i** *sec*] [**–f** *file*]

DESCRIPTION

The *sar* command, in the first instance, samples cumulative activity counters in the operating system *n* times every *t* seconds. If **–o***file* is specified, *sar* saves the samples in *file* in binary format.

In the second instance, if you do not specify a sampling interval, *sar* extracts data from a previously recorded *file*, either the one specified by the **–f** option or, by default, the standard system activity daily data file, **/usr/adm/sa/sa***dd*, for the current day *dd*. The starting and ending times of the report can be bounded through the **–s** and **–e** *time* arguments using the 24-hour clock form *hh*[:*mm*[:*ss*]]. The **–i** option selects records at *sec* second intervals. Otherwise, all intervals found in the data file are reported.

In either case, subsets of data to be printed are specified by the following options:

–u Reports CPU utilization (the default):
%usr, %sys, %wio, %idle – portion of time running in user mode, running in system mode, idle with some process waiting for block I/O, and otherwise idle, respectively.

–p Reports CPU utilization by processor:
%usr, %sys, %wio, %idle – portion of time running in user mode, running in system mode, idle with some process waiting for block I/O, and otherwise idle, respectively.

–b Reports buffer activity:
bread/s, bwrit/s – transfers per second of data between system buffers and disk or other block devices;
lread/s, lwrit/s – accesses of system buffers;
%rcache, %wcache – cache hit ratios, for example, 1 – bread/lread;
pread/s, pwrit/s – transfers via raw (physical) device mechanism.

–d Reports activity for each block device, for example, disk or tape drive:
%busy, avque – portion of time device was busy servicing a transfer request, average number of requests outstanding during that time;
r+w/s, blks/s – number of data transfers from or to device, number of bytes transferred in 512-byte units;
avwait, avserv – average time in ms. that transfer requests wait idly on queue, and average time to be serviced (which for disks includes seek, rotational latency, and data transfer times).

–y Reports TTY device activity:
rawch/s, canch/s, outch/s – input character rate, input character rate processed by canon, and output character rate;
rcvin/s, xmtin/s, mdmin/s – receive, transmit, and modem interrupt rates.

–c Reports system calls:
scall/s – system calls of all types;
sread/s, swrit/s, fork/s, exec/s – specific system calls;
rchar/s, wchar/s – characters transferred by read and write system calls.

–w Reports system swapping and switching activity:
swpin/s, swpot/s, bswin/s, bswot/s – number of transfers and number of 512-byte units transferred

for swapins (including initial loading of some programs) and swapouts;
pswch/s – process switches.

–a Reports use of file access system routines:
iget/s, namei/s, dirblk/s.

–q Reports average queue length while occupied, and % of time occupied:
runq-sz, %runocc – run queue of processes in memory and runnable;
swpq-sz, %swpocc – swap queue of processes swapped out but ready to run.

–v Reports status of text, process, inode, and file tables:
text-sz, proc-sz, inod-sz, file-sz – entries/size for each table, evaluated once at sampling point;
text-ov, proc-ov, inod-ov, file-ov – overflows occurring between sampling points.

–A Reports all data. Equivalent to **–updqbwcayv**.

EXAMPLES

To see today's CPU activity so far:

```
sar
```

To watch CPU activity evolve for 10 minutes and save data:

```
sar –o temp 60 10
```

To later review disk and tape activity from that period:

```
sar –d –f temp
```

FILES

/usr/adm/sa/sa*dd* Daily data file, where *dd* are digits representing the day of the month.

SEE ALSO

sag(1), sar(1)

NAME

sc – Invokes the front end for the scm source code control program

SYNOPSIS

sc *keyword pl* [*parameters*]

DESCRIPTION

The *sc* shell script is a front end to *scm*(1) that helps *scm* control source code similar to the UPDATE utility available with the Cray operating systems COS and UNICOS. *Sc* invokes *scm* in the appropriate directories and with the appropriate options.

A *program library* (PL) under *sc* control is simply a collection of *scm* source control files (**.m** suffix) arranged in a directory structure parallel to the "vanilla" source files.

The environment variable $PREFIX must be set to the name of the directory containing the PLs before calling *sc*. The options are as follows:

keyword Specifies the operation to be performed. *keyword* can be one of the following:

build Takes raw source files in $PREFIX/*pl*/.SCM. No parameters are needed.

list *parameter* Lists various administrative files used by *sc*. Available parameters are: *pl, files, mods, index,* and *log.*

addmod *modname*
Installs an *scm* modset in the PL. The parameter is the name of the modset, which must be in a file of the same name in the current directory.

delmod *modname*
Deletes an *scm* modset from the PL. The parameter is the name of the modset.

addfile *filename* Adds a new module to the PL. The parameter is the name of the file to be added. (the *scm* ***n** directive should be used instead of addfile.)

delfile *filename* Removes the named module from the PL

get *filename* Places an up-to-date copy of the given module in the current directory, as well as the associated source code control (**.m** suffix) file.

getx *filename* This is the same as get, except the *scm* is invoked with the **-z** option, which tells *scm* to use the basename of the module's name. This is useful when modules are stored in subdirectories within the PL> Without the **-z** option, all of the intervening subdirectories must be present in the current directory before the given module can be operated on. With the **-z** option, the subdirectory names are discarded, and the module is placed in the current directory directly.

yank *modname* Undoes the effects of the given modset without actually removing it from the PL.

unyank *modname*
Reverses the effects of a *yank.*

sync	Used internally by most other operations to ensure the text version of the module is consistent with the source control version and all applied modsets.

pl Indicates the PL (relative to $PREFIX) on which the operation is to be performed. Typical choices for *pl* are **cmd**, **include**, and **uts**.

parameters Parameters vary depending on the operaton being performed. Usually the parameter is the name of the module (for example, **mv.c**) on which to operate.

EXAMPLES

To put a file called *myprogram.c* into *scm* format for the first time, assuming also that no *sc* directory already exists, type the following after the scum% prompt:

```
setenv PREFIX 'pwd'
mkdir mypl
mv myprogram.c mypl
sc build mypl
```

To obtain a copy of *myprogram.c* for editing, edit that file, then place it back in the *sc* database (again, type the following commands after the scum% prompt):

```
mkdir work
cd work
sc get mypl myprogram.c
vi myprogram.c
your editing session
mxm –i myprogram.c –m modname
sc addmod mypl modname
```

To obtain a list of the names of all files in the PL, type the following after the scum% prompt:

```
sc list mypl files
```

FILES

$PREFIX/pl	Directory for source files
$PREFIX/pl.SCM	Directory for *scm* control files
$PREFIX/pl.SCMfilelist	List of all files in the PL
$PREFIX/pl.SCM/.modslist	List of all modsets in the PL.
$PREFIX/pl.SCM/.mods	Directory for modsets
$PREFIX/pl.SCM/.index	List of modsets per file

SEE ALSO

scm(1), mxm(1)

NAME

sccsdiff – Compares two versions of an SCCS file

SYNOPSIS

sccsdiff –r*SID1* **–r***SID2* [**–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.

–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

MESSAGES

file: No differences
  The two versions are the same.

Use *help*(1) for explanations.

SEE ALSO

bdiff(1), get(1), help(1), pr(1)
the UNICOS Source Code Control System (SCCS) User's Guide, publication SG-2017

NAME

scm – Invokes the source control program

SYNOPSIS

scm [**–cn**] [**–i** *modnames*] [**–I** *indirect*] [**–b** *basedir*] [**–p** *paths*] [**–q** *qnames*]

DESCRIPTION

The *scm* command invokes a source control program similar to the UPDATE utility available with the Cray operating systems COS and UNICOS.

Definitions. A *source file* is an input file containing C or other code. A *compile file* is an output file, otherwise identical to a source file. A *source control module* can be an output or input file. It is an ASCII text file that corresponds to a source file. Each line in the source control module has an identifier associated with it.

Scm performs two basic functions:

- Creates one or more ``base'' source control modules. Also called a *creation run*. A unique permanent identifier is assigned to each line of source. For example, source file *foo.c* would be converted into source control module *foo.c.m*. This function is requested with the **–n** option.
- Updates a source control module using a set of directives. Also called an *update run*. The directives are usually in files called *mods*. Each mod implements one change, or adds one new feature, to one or more source control modules. The directives can add and delete lines of source. Any new lines of source will be assigned unique permanent identifiers. For example, if a mod were applied to source control module *foo.c.m* a new version of *foo.c.m* would be created. If desired, a compilable version of *foo.c* would also be created. This function is requested by omitting the **–n** option.

If there are no errors, *scm* creates one or more files in the current working directory (see also *FILES*). *Scm* operates in either *full* mode or *quick* mode. A full run creates output for all source files named by the input directives. A quick run creates output for only specific source files. A full run is the default; a quick run is requested with the **–q** option.

If neither of the input options **–i** or **–I** are specified, directives will be read from *stdin*.

The options and their meanings are:

–c Compile file output. If this is a full update run, all files named by the input directives will be created. If this is a quick update run (**–q** option), only specific files will be created.

–n Create new source control module(s). If this option is specified, this will be a creation run, which means that each source file named by the input directives will be converted into a source control module (with an extension of *.m*). The only directives allowed with the **–n** option are ***f** and ***c**.

–i *modnames*
Specify one or more input mod filenames. If more than one filename is specified, they must be separated by commas with no intervening white space.

–I *indirect*
Read a list of input mod filenames from the file *indirect*. This option is useful if applying many mod files in a single run.

–p *paths*
These two options are used together to specify the directory or directories *scm* will search to find files named by the ***f** directive. If only one directory is to be searched, it may be specified with the **–b** option alone. The default for **–b** is the current working directory. If more than one directory is to be searched, suffixes may be specified with the **–p** option. The pathname suffixes *paths* must be separated by commas with no intervening white space. The directory pathnames to search are derived by appending the **–p** option pathnames to the **–b** option pathname. The default for **–p** is "/".

–q *qnames*
Quick update run. One or more source file names must be specified. If more than one source file name is specified, they must be separated by commas with no intervening white space.

DIRECTIVES

***m** *modname*
Specify mod name. There must be only one ***m** directive in a mod file. It must be the first directive in a mod file. The mod name is used to generate the unique identifier for each source line added by a mod.

***f** *filename*
Specify source file name. In a creation run, one ***f** directive is used for each source file that is to be converted to a source control module. No other directives (other than ***c**) are required or allowed. In an update run, the ***f** directive is used to name the source file that subsequent directives will modify. For an update run, *scm* will search for the corresponding source control module (*filename*.m). More than one ***f** directive may appear in a mod file; up to twenty-six source files may be referenced by a single mod.

***n** *filename*
Specify source file name. In an update run, one ***n** directive is used for each source file that is to be converted to a source control module. The ***n** directives can be intermixed with ***f** directives, but cannot rely on each other. The twenty-six source file limitation applies to the total of ***n** directives and ***f** directives.

***i** *ident* Insert the lines following this directive, up to the next directive, into the current source file after the line with identifier *ident.*

***b** *ident*
Insert the lines following this directive, up to the next directive, into the current source file before the line with identifier *ident.*

***d** *ident1* [*,ident2*]
Delete one or more lines from the current source file, starting with identifier *ident1,* and optionally continuing up to identifier *ident2.* Optionally replace the deleted lines with the lines following this directive, up to the next directive.

***c** [*text*]
Comment line. Ignored.

IDENTIFIERS

Unique identifiers are assigned by *scm* to each line of source in a source control module. These identifiers are of two types.

During a creation run, the ident is based on the filename. Therefore, the lines of file *foo.c* would be numbered *foo.1, foo.2, foo.3,* and so on.

During an update run, the ident is based on the modname. Now, a mod can affect up to twenty-six source control modules; the modname receives a suffix "a" for the first file modified, "b" for the second file modified, and so on. Therefore, the lines added by mod *foomod* would be numbered *foomoda.1, foomoda.2, foomoda.3,* and so on, for the first file; *foomodb.1, foomodb.2, foomodb.3,* and so on for the second file.

EXAMPLE

Creation run:

```
scm –i modules –b /usr/src/os/ –p os,io –n
```

Creates two base source control modules in the current working directory, *acct.c.m* and *bio.c.m.* The original source files are */usr/src/os/os/acct.c* and */usr/src/os/io/bio.c.* The source control modules can then be moved to an appropriate directory. For the following two examples we will assume that *acct.c.m* is moved into */usr/src/os/os,* and that *bio.c.m* is moved into */usr/src/os/io.*

Full update run:

```
scm –i mod1,mod2 –b /usr/src/os/ –p os,io –c
```

Applies two mods to the source control modules, and creates four files in the current working directory: new source control modules *acct.c.m* and *bio.c.m;* and compile files *acct.c* and *bio.c.* This kind of update run could be used to obtain the source for a large number of modules (i.e. the entire kernel) with a specific set of mods applied.

Quick update run:

```
scm –i mod1,mod2 –b /usr/src/os/ –p os,io –c –q bio.c
```

Applies the same two mods to the source control modules, but creates only two files in the current working directory: new source control module *bio.c.m,* and compile file *bio.c.* This kind of update run could be used to obtain just one source file for editing. The edited source file could then be used to create a new mod (see *mxm(1)* for details of this procedure).

Contents of creation run input file modules:

```
*f acct.c
*f bio.c
```

Contents of mod file mod1:

```
*m mod1
*c comments describing mod1
*f acct.c
*i acct.123
/* new source line */
*d acct.200,acct.220
*f bio.c
*i bio.10
/* new source line */
```

Contents of mod file mod2:

```
*m mod2
*c comments describing mod2
*f acct.c
*d mod1a.1
```

```
/* fix to source line added in mod1 */
```

FILES

Input files are specified with the **–i** and **–I** options, and with the ***f** directive. A file named by the ***f** directive is searched for in the directory or directories derived from the **–b** and **–p** options.

No output files are created if *scm* has detected any errors. Output files are always placed in the current working directory. *Scm* never silently replaces a file; an error is reported if an output file name will conflict with an existing file name. Two kinds of output file can be created. *Scm* always creates a new source control module. If the **–c** option is selected, a "compile file" (updated source file) is created.

BUGS

Scm does not sort directives. Thus if directives that add new lines to a source file are not in ascending order, the resulting identifiers given to the new lines will not be in ascending order.

Currently, to delete a module, you must submit a mod to delete all the lines in the file, and then remove the zero length *.m* module.

SEE ALSO

mxm(1)

NAME

scpqsub – Allows you to submit jobs to NQS from USCP spawned job

SYNOPSIS

scpqsub [**–d** *mf*] [**–t** *tid*] *filename*

DESCRIPTION

The *scpqsub* command lets you submit jobs to NQS and specify the front end to which you want the output returned if

- You submitted the job to NQS through USCP
- You are logged in interactively to UNICOS through USCP

Jobs submitted using *scpqsub* do not require a user ID record or a COS job or account card. A COS job or account card causes the job to abort.

The following options are available:

–d *mf*	Specifies a two character alternate station ID *mf*. The output is directed to the specified station. The default is the station of origin.
–t *tid*	Specifies an alternate *tid* (up to 8 characters) to be used by the front-end station. The default is the original *tid*.

MESSAGES

Returns a zero if successful and a nonzero if unsuccessful.

SEE ALSO

nqsintro(1), qsub(1)
COS Version 1 Reference Manual, publication SR-0011
Front-end Protocol Internal Reference Manual, publication SM-0042

NAME

scpreroute – Allows you to define station processing of job output

SYNOPSIS

scpreroute [**–d** *mf*] [**–t** *tid*] [**–c** *dc*] [**–f** *fm*] [**–s** *sf*] [**–i** *uid*] [**–T** *text*]

DESCRIPTION

The *scpreroute* command lets you supply information to the station with regard to the processing required for a job's output files.

The following options are available:

–d *mf*	Two character station ID. The default is the station of origin.
–t *tid*	Terminal identifier (up to 8 characters).
–c *dc*	Two character disposition code. The default is **PR** (disposed to printer).
–f *fm*	Two character format selection. The default is **UD** (Unicos Data).
–s *sf*	Special forms selection (up to 8 characters).
–i *uid*	User identification (up to 8 characters).
–T *text*	Up to 240 characters of text information for the station. This information must be contained within quotes.

NOTE

This command is only available to jobs submitted to NQS using USCP. It is equivalent to the COS deferred dispose, but applies only to **stdout** and **stderr**.

MESSAGES

Returns a zero if successful or a nonzero if unsuccessful.

SEE ALSO

dispose(1) COS Version 1 Reference Manual, publication SR-2011
Front-end Protocol Internal Reference Manual, publication SM-0042

NAME

script – Makes typescript of terminal session

SYNOPSIS

script [**–a**] [*file*]

DESCRIPTION

The *script* command makes a typescript of everything printed on your terminal. The typescript is saved in a file, and can be sent to the line printer later with *lpr*. If you specify a file name with the *file* argument, the typescript is saved there. If you do not specify a file name, the typescript is saved in the **typescript** file. The **–a** option causes the script to append to the **typescript** file instead of creating a new file.

The script ends when the forked shell exits.

This program is useful when you are using a CRT but you desire a hard-copy record of the dialog (for example, a student handing in a program that was developed on a CRT when hard copy terminals are in short supply).

NAME

script – Makes a typescript of terminal session

SYNOPSIS

script [**–n**] [**–s**] [**–a**] [**–q**] [**–S** *shell*] [*file*]

DESCRIPTION

The *script* command makes a typescript of everything printed on your terminal. The typescript is saved in a file and can be sent to the line printer later with *lpr*. If you specify a file name, the typescript is saved there. If not, the typescript is saved in the file **typescript.**

To exit script, type CONTROL-d. This sends an end-of-file to all processes you have started up, and causes script to exit. For this reason, CONTROL-d behaves as though you had typed an infinite number of CONTROL-ds.

This program is useful when you are using a CRT but you desire a hard-copy record of the dialog.

The options control what shell is used.

–n	Invokes the new shell
–s	Invokes the standard shell
–a	Causes script to append to the typescript file instead of creating a new file
–q	Invokes "quiet mode", where the "script started" and "script done" messages are turned off
–S *shell*	Allows you to specify any shell you want. The default depends on the system: **/bin/csh** is used where possible, otherwise **/bin/sh** is used. If the requested shell is not available, *script* uses any shell it can find.
file	Specifies the file where the typescript is saved (the default is *typescript)*

BUGS

Since UNICOS cannot write an end-of-file down a pipe without closing the pipe, you cannot simulate a single CONTROL-d without ending script.

The new shell has its standard input coming from a pipe rather than a tty, so *stty* does not work, and neither will *ttyname*.

When the user interrupts a printing process, *script* attempts to flush the output backed up in the pipe for better response. Usually the next prompt also gets flushed.

The *pty*(4D) feature for the CRAY X-MP could be used to make script work right.

SEE ALSO

pty (4D)

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 *output*.

SEE ALSO

diff(1), ed(1)

NAME

sed – Invokes the stream editor

SYNOPSIS

sed [**–n**] [**–e** *script*] [**–f** *sfile*] [*files*]

DESCRIPTION

Sed copies the named *files* (standard input by 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, that is, 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 **!** described 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*
Negation. 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.

(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

ed(1), grep(1)

NAME

segldr – Invokes the segment loader (SEGLDR)

SYNOPSIS

segldr [**–o** *objfile*] [**–V**] *file* ...

DESCRIPTION

The *segldr* command links relocatable binaries together to produce an executable binary. Directives to *segldr* are read from all the files named that do not end with **.o**. All files ending in **.o** are assumed to be relocatable input files (that is, **bin=test.o**). The listing and error messages are written to standard output. An error message summary is written to standard error. The executable file will be execute-enabled if no **WARNING** or **FATAL** errors occurred during the load.

The **–o** *objfile* option indicates the file where the executable output is to be written. If the **–o** option is not used, the executable is written to the file named by the **abs** directive. If neither the **–o** option nor the **abs** is specified, the executable output is written to **a.out**. The **–V** option indicates that *segldr* should output its version line to standard error.

FILES

stdout	*segldr* listing and eror messages
stderr	*segldr* version line and error summary
SYMBOLS	Symbol tables and debug table
a.out	Executable binary

MESSAGES

The full range of *segldr* error messages and proper response to each can be found in the Segment Loader (SEGLDR) Reference Manual, publication SR-0066.

SEE ALSO

relo(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014
All *segldr* directives are described in the Segment Loader (SEGLDR)Reference Manual, publication SR-0066. Particular attention should be given to the **bin** and **map** directives.

NAME

sh, rsh – Shell, the standard/restricted command programming language

SYNOPSIS

sh [**–acefhiknrstuvx**] [*args*]
rsh [**–acefhiknrstuvx**] [*args*]

DESCRIPTION

The *sh* invokes the command programming language that executes commands read from a terminal or a file. The *rsh* command 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 (that is, 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 a conditional command. Unless otherwise stated, the value returned by a command is that of the last simple-command executed in the command.

for *name* [**in** *word* ...] **do** *list* **done**
: Each time a **for** command is executed, *name* is set to the next *word* taken from the **in** *word* list. If **in** *word* ... is omitted, then the **for** command executes the **do** *list* once for each positional parameter that is set (see *Parameter Substitution* below). Execution ends when there are no more words in the list.

case *word* **in** [*pattern* [**|** *pattern*] ... **)** *list* **;;**] ... **esac**
: A **case** command executes the *list* associated with the first *pattern* that matches *word*. The form of the patterns is the same as that used for file-name generation (see *File Name Generation*) 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 pipeline or 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 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, the message "parameter null or not set" is printed.

${*parameter*:+*word*}

If *parameter* is set and is non-null, 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, 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.

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.

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 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, 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* (that is, made to stand for itself) by preceding it with a \. The pair **\\new-line** is ignored. All characters enclosed between a pair of single quote marks (''), 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, the secondary prompt (that is, the value of **PS2**) is issued.

Input/Output

Before a command is executed, its input and output may be redirected using a special notation interpreted by the shell. The following may appear anywhere in a simple-command or may precede or follow a *command* and are *not* passed on to the invoked command; 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, it is created; otherwise, it is truncated to zero length.
>>word	Use file *word* as standard output. If the file exists, 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, 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 **<<**, 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**.
<&–	The standard input is closed. Similarly for the standard output using **>&–**.

If any of the above is preceded by a digit, 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.

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 (that is, *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*. However, the proper syntax to direct output to a pipe (which is normally associated with file descriptor 1) is:

```
cmd1 2>&1 | cmd2
```

If a command is followed by & 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* 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. 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 / 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 (that is, a machine-language binary file, *exec*(2)), it is assumed to be a file containing shell commands. A sub-shell 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 *exec* operations 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

Input/output redirection is now 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*.

break [*n*]
Exit from the enclosing **for** or **while** loop, if any. If *n* is specified break *n* levels.

continue [*n*]
Resume the next iteration of the enclosing **for** or **while** loop. If *n* is specified 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 cdpath is <**null**> (specifying the current directory). Note that the current directory is specified by a null cdpath name, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the cdpath list. If *arg* begins with a / the search path is not used. Otherwise, each directory in the cdpath 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*, where *n* is between 0 and 255. If *n* is less than 0, the shell will complain of a bad exit number. If *n* is greater than 255, the exit status is the remainder of *n* divided by 255 (*n* mod 255). If *n* is omitted 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, 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 *name*s are marked *readonly* and the values of these *name*s may not be changed by subsequent assignment. If no arguments are given, a list of all *readonly* names is printed. The *readonly* attribute is not passed in the environment but is effective in "(list)" subshells.

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 for export as they get modified or created.
–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 or list of commands.
–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**, **$3**, ... If no arguments are given, 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 *expr*
: [*expr*]
: Evaluate conditional expressions. See *test*(1) for usage and description.

times
: Print the accumulated user and system times for processes run from the shell.

trap [*arg*] [*n*] ...
: The command *arg* is to be read and executed when the shell receives signal(s) *n*. (Note that *arg* is scanned once when the trap is set and once when the trap is taken.) Trap commands are executed in order of signal number. Any attempt to set a trap on a signal that was ignored on entry to the current shell is ineffective. An attempt to trap on signal 11 (memory fault) produces an error. If *arg* is absent, all traps *n* are reset to their original values. If *arg* is the null string, this signal is ignored by the shell and by the commands it invokes. If *n* is 0, 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*]
: imposes a size limit of *n*

–f	imposes a size limit of *n* blocks 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 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 commands are read from *string*.

–s If the **–s** flag is present or if no arguments remain 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, 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 (Restricted Shell)

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 (that is, **/usr/rbin**) that can be safely invoked by *rsh*. Some systems also provide a restricted editor *red*.

FILES

/etc/profile
$HOME/.profile
/tmp/sh*
/dev/null
a.out(4F), profile(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

MESSAGES

Errors detected by the shell, such as syntax errors, cause the shell to return a non-zero exit status. If the shell is being used non-interactively execution of the shell file is abandoned. Otherwise, the shell returns the exit status of the last command executed (see also the **exit** command above).

NOTES

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.

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.

SEE ALSO

cd(1), csh(1), echo(1), env(1), login(1), pwd(1), test(1), umask(1)
dup(2), exec(2), fork(2), pipe(2), signal(2), ulimit(2), umask(2), wait(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

sim – Invokes the interactive Cray Simulator

SYNOPSIS

sim [**–dstv**] [**–m***n*] [*file*] [*args*]

DESCRIPTION

The *sim* command invokes an interactive Cray simulator. It can simulate user programs for CRAY-2 computer systems and can simulate UNICOS for the CRAY-2. The simulator builds an argument vector when the program is loaded.

The following options are available:

–d	Starts the simulator with display of tracing off
–m*n*	Sets the debug message level to *n*
–s	Starts the simulator in system mode. This is required to simulate UNICOS (CRAY-2 only). In system mode, the simulator uses files in the current directory for the simulated disk devices. These files are named **simdev.***nn*, where *nn* is the phyical device number (decimal) as known to the foreground processor. The simulator assumes phyical device numbers below 20 to be DD-29 disks. Physical device numbers of 20 or more are assumed to be D-49 disks.
–t	Starts the simulator with instruction timing on
–v	Turns off Virtual Memory. Normally, the simulator keeps the entire absolute file in memory but uses Virtual Memory pages for any BSS space and any space that the simulated program requests. This option tells the simulator to keep the entire simulated image in memory.
file	The name of an absolute binary file created by the loader
args	Arguments for the simulated program

Once you invoke the simulator, you can give it a number of commands. Commands to *sim* are single characters followed by other optional information. They fall into several categories:

- Execution control (for example, **x22**)
- Display control (for example, **.a=100**)
- Setting memory and registers (for example, **s0=123**)
- File manipulation (for example, **f=file**)

The following is a summary of the available commands.

^C CONTROL-C interrupts simulation and prompts for more commands.

^D CONTROL-D repeats the last command. This is especially useful for stepping through a program several lines at a time.

![*command*]
Executes shell commands. If you do not specify an argument, the simulator starts a shell and suspends execution until the shell reads an end of file. If you supply *command*, the simulator spawns a shell to execute the command.

*****[*comment*]
Allows commenting. This is especially useful when using an alternate command file.

? Displays available commands and their syntaxes

+[*n*] Scrolls the last display forward *n* octal words. The default is the display size.

−[*n*] Scrolls the last display backward *n* octal words. The default is the display size.

.[*a*[=[*add*][*type*][,*size*][*format*][/*label*]]]
: Controls displays. There are twenty six displays available. They are named a-z and can be individually set up. They are saved with checkpoints for convenience. A period specified alone prints the current display setup. A period followed by a single letter (.*a*) prints that display. *Add* can be a global symbol, an octal address, or an A or S register indicator (for example, "!a0" uses contents of register A0). *Type* can be: *c* for Common Memory, *l* for Local Memory, or *v* for vector register (for example, "120v" is register V1, element 020). *Size* is the octal number of words to print for the display. *Format* is the desired display format. The format can be: *b* for bit format, *d* for decimal format, *f* for floating point format, *h* for hexadecimal format, *i* for instruction format, *p* for parcel format, or *w* for word format, The optional *label* parameter allows the display to be labeled. This has no effect other than to be displayed when listing displays.

/*symbol*[=*value*]
: Displays the location of *symbol* or optionally change the contents of that location. This value can be a string of octal digits optionally followed by a parcel indicator (*a, b, c, d*). It can also be a quoted string (for example, 'ABC'), which is right adjusted, zero filled.

<*file* Opens *file* and reads directives from it. Continues reading from standard input when an end of file is reached. The directive file can also specify another directive file, but there is no return to the original directive file.

>[>][:]*file*
: Opens trace file. This creates the named file and begins duplicating all output onto this file. If you specify the second '>', all output is appended if the file already exists. If you specify the ':', instruction traces are only written to the trace file and not to the terminal.

add[*p*]=*val*
: Stores a value *val* into common memory address *add*. This value can be a string of octal digits optionally followed by a parcel indicator (*a, b, c, d*). It can also be a quoted string (for example, 'ABC'), which is entered right adjusted, zero filled. A parcel indicator *p* can be appended to the address. This indicates that only one parcel is to be changed.

a*n*=*val* Sets A register *n* to an octal value *val*.

b[*n*[=*padd*[/*label*][(*cond*)][;*cmds*]]]
: Controls breakpoints. If you specify **b** alone, all current breakpoints are listed. If you do not provide a parcel address, the breakpoint is cleared. The parameter *n* is a single decimal digit. *Padd* can be an octal parcel address, a global symbol name, or an A or S register indicator (example: "!a0"). *Cond* is a condition expression. Conditional breakpoint expressions are of the form (*operand operator operand*), where *operand* may be: a common memory location (example: (100) or (symbol)), a local memory location (example: [200] or [symbol]), an A or S register (example: a4 or s7), or a constant. Memory addresses and constants are assumed to be octal. *Operator* can be one of: =, !=, <, <=, >, >=. *Cmds* is an optional string of commands, separated by semi-colons, to be executed when the breakpoint is reached. These may include setting other breakpoints (this is where using a register as a breakpoint address comes in handy). This provides for a maximum of ten active breakpoints. Breakpoints remain active until cleared. A breakpoint must be cleared before it is re-used. The optional *label* argument allows the breakpoint to be labeled. This has no effect other than to be displayed when listing breakpoints and when the breakpoint is hit.

c=*file* Creates a checkpoint of the current simulator state in *file*. The file is overwritten if it already exists. All breakpoints and display set-ups are recorded in the checkpoint file.

d{+,–,r}
: Controls display of instruction tracing. The + turns on all tracing, – turns off all tracing, and **r** turns on tracing of only return jumps.

f=*file* [*args*]
: Loads the absolute binary *file*. This file is the output from the loader. *Args* are the arguments for the simulated program.

h Displays a history of how I got here. Sometimes known as a traceback.

i<[*file*] Provides the ability to redirect the simulated program's input from a file other than the terminal. If you do no specify *file*, the input is read from the terminal.

i{*b,d,i,j,p,t*}
: Prints information about simulation. The *b* option displays the number of clock periods spent waiting for instruction buffer loads (timing must be on). The *d* option displays device interrupt status. This is only meaningful when simulating in system mode. The *i* option displays the number of times each instruction was executed. The *j* option displays information about conditional jump execution. The *p* option displays current virtual memory page imformation. The *t* option displays current simulator time used.

l*add*=*val*
: This command is only valid on the CRAY-2 only. It stores a value *val* into local memory address *add*. This value may be a string of octal digits optionally followed by a parcel indicator (*a, b, c, d*). It may also be a quoted string (for example, 'ABC'), which is entered right adjusted, zero filled.

m=*n* Sets the debug message level to *n*. The higher the message level, the more debug messages that are printed.

o>[>[*file*]]
: Sends the standard output of the simulated program to another file. If you specify the second '>', the data is appended. If you do not specify *file*, the output is directed back to the terminal.

p=*padd* Sets the **P** register to a parcel address. *Padd* may be a global symbol or an octal parcel address.

pn=*n* Changes the currently active processor number to processor *n*. The simulator provides the capability to simulate multi-tasking programs. The TFORK system call activates another simulated processor and makes a copy of the current register and local memory contents. The simulator automatically switches between processors whenever the semaphore is cleared or when a set semaphore is tested. This command allows switching on demand. The register and local memory displays always display the contents of the currently active processor.

q Quit. Terminate the simulator.

r=*file* Restarts the simulation from a checkpoint in *file*.

s*n*=*val* Sets S register *n* to an octal value *val*.

t{+,–} Controls instruction timings. The + turns on instruction timings. It also zeroes the timer if timing was already on. The – turns off instruction timings.

v*nnn*=*val*
: Sets **V** register *nnn* to a value *val*. This value may be a string of octal digits optionally followed by a parcel indicator (*a, b, c, d*). It may also be a quoted string (for example, 'ABC'), which is right adjusted, zero filled.

vl=*val* Sets the vector length register to a decimal value *val*.

vm=*val* Sets the vector mask register to an octal value *val*.

w[*n*[=[*add1*][**l**][,*add2*][{**r**,**w**}][/*label*][(*cond*)]]]

Controls watchpoints. Watchpoints are memory addresses that are watched for a reference. If the specified address or range of addresses is referenced, the simulator stops and prints a message. If you specify **w** with no arguments, all current watchpoints are listed. If you do not specify an address, the watchpoint is cleared. The parameter *n* is a single decimal digit. *Add1* is the start address of the area to be watched. When the address is followed by l, this indicates that local memory is to be watched. *Add2* is the end address of the area to be watched. If omitted, it is the same as the start address. The optional trailing letter gives the ability to watch for only reads or only writes. If you specify **r**, only memory reads are watched. If you specify **w**, only memory writes are watched. The default is to watch both reads and writes. The optional *label* parameter allows the watchpoint to be labeled. This has no effect other than to be displayed when listing watchpoints and when the watchpoint is hit. *Cond* is a condition expression. Condition expressions are the same as for breakpoints (explained above). This mechanism provides for a maximum of ten active watchpoints. Watchpoints remain active until cleared. A watchpoint must be cleared before it is re-used.

x[{*n*,*****,**j**,**r**,**e**}][{**+**,**–**}]

Executes instructions. You can specify a count *n*. The default is to execute one instruction. An asterisk indicates infinity. The **j** option means to execute to the next jump instruction (this includes return jumps and EXITs). The **r** option means to execute to the next return jump instruction. The **e** option means to execute to the next EXIT instruction. The optional trailing + or – is available to override the current instruction trace status. If you use the – when instruction tracing is currently on, the simulator will execute silently to the next stopping point, but tracing will still remain on by default.

FILES

/usr/bin/sim	Cray simulator

SEE ALSO

csim(1)

NAME

size – Prints section sizes of executable files

SYNOPSIS

size [**–o**] [**–x**] *files*

DESCRIPTION

The *size* command produces size information for each section in an absolute binary executable file. The size of the text, data and bss (uninitialized data) sections are printed along with the total size of the object file.

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.

SEE ALSO

as(1), cc(1), ld(1)
a.out(4) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

sleep – Suspends 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) in the UNICOS System Calls Reference Manual, publication SR-2012
sleep(3C) in the CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual, publication SR-2013

NAME

sno – Invokes the SNOBOL interpreter

SYNOPSIS

sno [*files*]

DESCRIPTION

Sno is a SNOBOL compiler and interpreter (with slight differences). *Sno* obtains input from the concatenation of the named *file*s and the standard input. All input through a statement containing the label **end** is considered program and is compiled. The rest is available to **syspit**.

Sno differs from SNOBOL in the following ways:

There are no unanchored searches. To get the same effect:

a ** b	unanchored search for *b*.
a *x* b = x c	unanchored assignment

There is no back referencing.

x = "abc"	
a *x* x	is an unanchored search for **abc**.

Function declaration is done at compile time by the use of the (non-unique) label **define**. Execution of a function call begins at the statement following the **define**. Functions cannot be defined at run time, and the use of the name **define** is preempted. There is no provision for automatic variables other than parameters. Examples:

```
define f( )
define f(a, b, c)
```

All labels except **define** (even **end**) must have a non-empty statement.

Labels, functions and variables must all have distinct names. In particular, the non-empty statement on **end** cannot merely name a label.

If **start** is a label in the program, program execution will start there. If not, execution begins with the first executable statement; **define** is not an executable statement.

There are no builtin functions.

Parentheses for arithmetic are not needed. Normal precedence applies. Because of this, the arithmetic operators / and * must be set off by spaces.

The right side of assignments must be non-empty.

Either ′ or " may be used for literal quotes.

The pseudo-variable **sysppt** is not available.

SEE ALSO

awk(1)

NAME

sort – Sorts and/or merges files

SYNOPSIS

sort [–cmu] [–o*output*] [–y[*kmem*]] [–z*recsz*] [–dfiMnr] [–bt*x*] [+*pos1* [–*pos2*]] [*files*]

DESCRIPTION

The *sort* command 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.

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.

–ooutput
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 blanks between **–o** and *output*.

–y[kmem]
The amount of 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, –y0 is guaranteed to start with minimum memory. By convention, –y (without an argument) starts with maximum memory.

–zrecsz
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 lower case letters into upper case.

–i Ignore characters outside the ASCII range 040-0176 in non-numeric comparisons.

–M Compare as months. The first three non-blank characters of the field are folded to upper case and compared so that "JAN" < "FEB" < ... < "DEC". Invalid fields 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.

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 (for example, *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

While the above explanation of *pos* arguments is technically correct, it is rather difficult to understand. The confusion seems to stem from the difference between *+pos* and *-pos* in the handling of their *m* part. Consider the following line of input to *sort*:

```
        abc:def:ghi
```

and assume that the : character is being used as a delimiter. Imagine that *+pos* and *-pos* are pointers to characters on the input line so that the sort key begins with the *+pos* pointer and ends with the *-pos* pointer. The rules are that *+m* is the first character of the *m+1*-st field, and *-m* is the last character of the *m*-th field. For example, the arguments +1 and *-2* place pointers at the "d" and "f" characters, respectively. Additional position control is provided by the *n* part, which moves the pointer forward by the specified number of characters. For example, the arguments +1.1 and *-2.2* would position the start and end pointers to the "e" and "g" characters, respectively.

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*(4F)) 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
```

Remember that lines whose keys are identical are ordered by sorting on the entire line.

FILES

/usr/tmp/stm???

MESSAGES

Comments and exits with non-zero status for various trouble conditions (such as 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.

SEE ALSO

comm(1), join(1), uniq(1)

NAME

split – Splits a file into pieces

SYNOPSIS

split [*–n*] [*file* [*name*]]

DESCRIPTION

The *split* command 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 in its stead, then the standard input file is used.

NAME

strings – Finds the printable strings in a object or other binary file

SYNOPSIS

strings [**–**] [**–o**] [*–number*] *file* ...

DESCRIPTION

The *strings* command looks for ASCII strings in the specified binary *file*. A string is any sequence of 4 (the default) or more printing characters ending with a newline or a null. Unless you specify the **–** option, *strings* only looks in the initialized data space of object files. If you specify the **–o** option, each string is preceded by its offset in the file (in octal). If you specify the *–number* option, *number* is used as the minimum string length rather than 4 (the default).

The *strings* command is useful for identifying random object files.

BUGS

The algorithm for identifying strings is extremely primitive

SEE ALSO

od(1)

NAME

stty – Set the options for a terminal

SYNOPSIS

stty [**–a**] [**–g**] [*options*]

DESCRIPTION

The *stty* command sets certain terminal I/O options for the current standard input device. Without arguments, it reports the settings of the following options: speed, erase, brkint, and echo. If you specify the **–a** option, *stty* reports all of the option settings. If you specify the **–g** option, *stty* 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 can be found in *termio*(4D). 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. Select your termininal I/O setup is selected from the following terminal options:

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*(4D))
0	Hang up phone line immediately
50 75 110 134 150 200 300 600 1200 1800 2400 4800 9600 exta extb	Set terminal baud rate to the number given, if possible. (Not all speeds are supported by all hardware interfaces.)
hupcl (–hupcl)	Hang up (do not hang up) DATA-PHONE® 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
clocal (–clocal)	Assume a line without (with) modem control
loblk (–loblk)	Block (do not block) output from a non-current layer

Input Modes

ignbrk (–ignbrk)	Ignore (do not ignore) break on input
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*(4D))
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*(4D))
nl0 nl1	Select style of delay for line-feeds (see *termio*(4D))
tab0 tab1 tab2 tab3	Select style of delay for horizontal tabs (see *termio*(4D))
bs0 bs1	Select style of delay for backspaces (see *termio*(4D))
ff0 ff1	Select style of delay for form-feeds (see *termio*(4D))
vt0 vt1	Select style of delay for vertical tabs (see *termio*(4D))

Local Modes

isig (–isig)	Enable (disable) the checking of characters against the special control characters INTR, QUIT, and SWTCH
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 erases 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, QUIT, or SWTCH

Control Assignments

control-character c	Set *control-character* to *c*, where *control-character* is **erase, kill, intr, quit, swtch, eof, min,** or **time** (**min** and **time** are used with **–icanon**; see *termio*(4)). If *c* is preceded by an (escaped from the shell) caret (^), then the value used is the corresponding CONTROL character (for example, "**^d**" is a **CONTROL-d**); "**^?**" is interpreted as DEL and "**^–**" is interpreted as undefined.
line *i*	Set line discipline to *i* (0 < *i*< 127). Currently only line discipline implemented is 0. Nothing else works.

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, SWTCH, 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 **DEL** and **^U**
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**, or **tek**

SEE ALSO

ioctl(2) in the UNICOS System Calls Reference Manual, publication SR-2012
termio(4D) the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

su – Lets you become super user or another user

SYNOPSIS

su [–] [*name* [*arg* ...]]

DESCRIPTION

The *su* command allows one to become another user without logging off. The default user *name* is **root** (that is, 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 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*(4F)), or **/bin/sh** if none is specified (see *sh*(1)). To restore normal user ID privileges, type an **EOF** (*cntrl-d*) to the new shell.

Any additional arguments given on the command line are passed to the program invoked as the shell. For example, when *sh*(1) is used, an argument of the form **–c** *string* executes *string* via the shell and an a argument 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, with the possible exception of **$PATH**, which is 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 the program is invoked with an *arg0* of *–program* by both *login*(1) and *su*(1).

All attempts to become another user using *su* are logged in the log file **/usr/adm/sulog**.

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

/etc/passwd	System's password file
/etc/profile	System's profile
$HOME/.profile	User's profile
/usr/adm/sulog	Log file

SEE ALSO

env(1), login(1), sh(1)
passwd(4F), profile(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

sum – Prints checksum and block count of a file

SYNOPSIS

sum [**–r**] *file*

DESCRIPTION

The *sum* command calculates and prints a 16-bit checksum for the named *file* and prints the number of 512-byte blocks in the file. It 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.

MESSAGES

''Read error'' is indistinguishable from end-of-file on most devices; check the block count.

SEE ALSO

wc(1)

NAME

sync – Flushes file system cache to disk

SYNOPSIS

sync

DESCRIPTION

The *sync* command executes the *sync* system primitive. If the system is to be stopped, *sync* must be called to ensure file system integrity. It will flush all previously unwritten system buffers out to disk, thus assuring that all file modifications up to that point will be saved. See *sync*(2) for details.

SEE ALSO

sync(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

tail – Displays the last part of a file

SYNOPSIS

tail [± [*number*][**lbc** [**f**]]] [*file*]

DESCRIPTION

The *tail* command 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. For example, the command:

```
tail –f fred
```

will print 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. 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.

BUGS

Tails relative to the end of the file are stored up in a buffer, and thus are limited in length.
Various kinds of anomalous behavior may happen with character special files.

NAME

tar – Tape file archiver

SYNOPSIS

tar [*key*] [*files*]

DESCRIPTION

The *tar* command saves and restores files on magnetic tape. 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. 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 written on the end of the tape. The **c** function implies this function.

x The named *files* are extracted from the tape. If a named file matches a directory whose contents had been written onto the tape, this directory is (recursively) extracted. The owner, modification time, and mode are restored (if possible). If no *files* argument is given, the entire content of the tape is extracted. Note that if several files with the same name are on the tape, the last one overwrites all earlier ones.

t The names of the specified files are listed each time that they occur on the tape. If no *files* argument is given, all the names on the tape are listed.

u The named *files* are added to the tape if they are not already there, or have been modified since last written on that tape.

c Create a new tape; writing begins at the beginning of the tape, instead of after the last file. This command implies the **r** function.

The following characters may be used in addition to the letter that selects the desired function:

0,...,9 This modifier selects the drive on which the tape is mounted. The default is **1**.

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/mt?**. If the name of the file is –, *tar* writes to the standard output or reads from the standard input, whichever is appropriate. Thus, *tar* can be used as the head or tail of a pipeline. *Tar* can also be used to move hierarchies with the command:

```
cd fromdir; tar cf – . | (cd todir; tar xf –)
```

b causes *tar* to use the next argument as the blocking factor for tape records. The default is 1, the maximum is 20. This option should only be used with raw magnetic tape archives (see **f** above). The block size is determined automatically when reading tapes (key letters **x** and **t**).

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 times. The modification time of the file will be the time of extraction.

o causes extracted files to take on the user and group identifier of the user running the program rather than those on the tape; this option is always on.

EXAMPLES

The following command backs up a user's entire directory to IOS expander tape on a CRAY X-MP or CRAY-1 computer system:

```
cd; tar -cv .
```

The following command backs up a user's entire directory to online magnetic tape on a CRAY X-MP or CRAY-1 computer system:

```
cd; tar; -cvf - . | bmxio -o -b 4096
```

In the second example, a blocksize of 4096 was chosen so the tape can be read back with either online or expander tape.

FILES

/dev/mt?
/tmp/tar*

MESSAGES

Complains about bad key characters and tape read/write errors.
Complains if enough memory is not available to hold the link tables.

BUGS

There is no way to ask for the *n*-th occurrence of a file.
Tape errors are handled ungracefully.
The **u** option can be slow.
The **b** option should not be used with archives that are going to be updated. The current magnetic tape driver cannot backspace raw magnetic tape. If the archive is on a disk file, the **b** option should not be used at all, because updating an archive stored on disk can destroy it.
The current limit on file-name length is 100 characters.

SEE ALSO

cpio(1), bmxio(1)

NAME

target – Verify target CPU characteristics

SYNOPSIS

target [*cpuname*]

DESCRIPTION

The *target* command determines and prints the CPU characteristic for the machine type specified by *cpuname*. The *cpuname* argument may be any of the following (upper or lower case):

cpuname	*cpuname*
host (or *host)	cray-1m
target (or *target); the default	cray-xmp
cray-1	cray-x1 (CRAY X-MP models 11, 12, 14, and 18)
cray-1a	cray-x2 (CRAY X-MP models 22, 24, and 28)
cray-1b	cray x4 (CRAY X-MP models 48 and 416)
cray-1s	cray-2

If the machine type is **target**, the current environment is checked for the **TARGET** environment variable. If it is found, *target* parses the environment variable and displays the machine characteristics for the target machine. If the machine type is **host**, the host machine characteristics are displayed. For any other specified machine type, the machine type's default characteristics are given.

The *target* command does not make changes to the current environment variable TARGET (for more information on the **TARGET** environment variable, see the *sh*(1) command. The user must initialize and/or make changes to this environment variable at the shell level when necessary. The *target* command verifies that the environment variable is syntactically correct.

To initialize the **TARGET** environment variable, enter

```
export TARGET
```

The format to set up or change the **TARGET** environment variable is as follows:

TARGET=[*cpuname*]{,[*charac*]}

The target machine represented by **TARGET** takes on the default machine characteristics specified by *cpuname*, modified accordingly by any specified *charac*'s. If you do not specify *cpuname*, the machine characteristics of the host machine are used and modified. The *cpuname* and *charac* arguments on the **TARGET** variable can be the following:

cpuname	Same options as listed above.
charac	These are possible features that may be specified for the given *cpuname* computer.

The CRAY-2 computer systems do not have special options.

The CRAY X-MP and CRAY-1 computer systems let you specify logical and numerical traits as listed in the following tables:

Logical Traits	
ema noema	Extended memory addressing No extended memory addressing
cigs nocigs	Compressed index and gather/scatter No compressed index or gather/scatter
vpop novpop	Vector pop count No vector pop count
pc nopc	Programmable clock No programmable clock
readvl noreadvl	Read vector length Do not read vector length
vrecur novrecur	Vector recursion No vector recursion
avl noavl	Additional vector logical No additional vector logical
hpm nohpm	Hardware performance monitor No hardware performance monitor
statrg nostatrg	Status register No status register
bdm nobdm	Bidirectional memory No bidirectional memory
cori nocori	Control operand range interrupts No control operand range interrupts

Numeric Traits	
banks=*n*	Number of memory banks
numcpus=	Number of CPU's
ibufsize=*n*	Instruction buffer size
memsize=*n*{*k m*}	Memory size in words *k* implies (*n* * 1024) words *m* implies (*n* * 1,048,576) words
memspeed=*n*	Memory speed in clock periods
clocktim=*n*	Clock period in picoseconds
numclstr=*n*	Number of clusters
bankbusy=*n*	Number of clock periods that the memory bank reserved

SEE ALSO

sh(1)
exec(2) in the UNICOS System Calls Reference Manual, publication SR-2012
profile(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014
System Library Reference Manual, publication SM-0114 (Machine Characteristics Routines)

NAME

tee – Duplicates output

SYNOPSIS

tee [**–i**] [**–a**] [*file*] ...

DESCRIPTION

Tee is a filter that transcribes the standard input to the standard output and makes a copy in *file*. The **–i** option ignores interrupts; the **–a** option causes the output to be appended to *file* rather than overwriting it.

NAME

telnet – User interface to the TELNET protocol

SYNOPSIS

telnet [*host* [*port*]]

DESCRIPTION

Telnet is used to communicate with another host using the TELNET protocol. If *telnet* is invoked without arguments, it enters command mode, indicated by its prompt (**telnet>**). In this mode, it accepts and executes the commands listed below. If it is invoked with arguments, it performs an *open* command (see below) with those arguments. **NOTE:** Outbound telnet is not currently supported on CRAY X-MP and CRAY-1 Computer Systems.

Once a connection has been opened, *telnet* enters input mode. In this mode, text typed is sent to the remote host. To issue *telnet* commands when in input mode, precede them with the *telnet* escape character, which is displayed at the beginning of the *telnet* session. When in command mode, the normal terminal editing conventions are available.

The following *telnet* commands are available. Only enough of each command to uniquely identify it need be typed.

? [*command*]
: Get help. With no arguments, *telnet* prints a help summary. If *command* is specified, *telnet* will print the help information available about the command only.

open *host* [*port*]
: Open a connection to the named host. If the no port number is specified, *telnet* will attempt to contact a TELNET server at the default port. The host specification may be either a host name (see *hosts(4F)*) or an Internet address specified in dot notation (see *inet* (3N)).

close Close a TELNET session and return to command mode.

quit Close any open TELNET session and exit *telnet*.

z Create a local shell.

escape [*escape-char*]
: Set the *telnet* escape character. Control characters may be specified as ^ followed by a single letter; for example, control-X is **^x**.

status Show the current status of *telnet*. This includes the peer one is connected to, as well as the state of debugging.

options Toggle viewing of TELNET options processing. When options viewing is enabled, all TELNET option negotiations will be displayed. Options sent by *telnet* are displayed as SENT, while options received from the TELNET server are displayed as RCVD.

crmod Toggle carriage return mode. When this mode is enabled any carriage return characters received from the remote host will be mapped into a carriage return and a line feed. This mode does not affect those characters typed by the user, only those received. This mode is not very useful, but is required for some hosts that like to ask the user to do local echoing.

negotiate [*command option*]
: Negotiate TELNET options over an open connection.

: The *negotiate* command can be used to negotiate options over an open connection. The following options are currently supported and available for negotiating:

binary	Transmit in binary mode
sga	Suppress "go ahead"
tm	Timing mark
echo	Enable remote echoing
status	Show current status of options
exopl	Print extended options list

Currently no options are defined on the extended options list, but perhaps in the future (possibly distant future), some options may be defined on the extended options list.

Once a negotiate command is typed, *telnet* prompts you for an option. The list of options available can be displayed by typing a question mark (?). The option must be preceded by a DO, DONT, WILL or WONT command. Only enough of each command option to uniquely identify it need by typed.

The options negotiations follow the loop-preventing rules mentioned in the RFC 854 specifications. For a detailed description of the options, see the TELNET specifications (RFC 854-861).

NAME

test – Performs a conditional evaluation

SYNOPSIS

test *expr*
[*expr*]

DESCRIPTION

The *test* command 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 fifo (named pipe) special file
–u *file*	True if *file* exists and its set-user-ID bit is set
–g *file*	True if *file* exists and its set-group-ID bit is set
–k *file*	True if *file* exists and its sticky bit is set. This is always false; there is no sticky bit in UNICOS.
–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.

WARNING

In the second form of the command (that is, the one that uses [ ], rather than the word *test*), the square brackets must be delimited by blanks.

SEE ALSO

find(1), sh(1)

NAME

tftp – Invokes the trivial file transfer program

SYNOPSIS

tftp [*host*] [*port*]

DESCRIPTION

The *tftp* command is the user's interface to the ARPANET standard Trivial File Transfer Protocol (TFTP). The program allows you to transfer files to and from a remote network site using the User Datagram Protocol (UDP). It defaults to the standard UDP port 69, which TFTP protocol uses.

The client *host* with which *tftp* is to communicate may be specified on the command line. If this is done, *tftp* will immediately attempt to establish a connection to a TFTP server on that host; either way, *tftp* will enter its command interpreter and await instructions from the user. When *tftp* is awaiting commands from the user, the prompt **tftp>** is displayed. The following commands are recognized by *tftp*:

connect *host-name* [*port*]
: Establish a connection to the supplied remote host.

get *remote-file* [*local-file*]
: Retrieve *remote-file* and store it on the local machine. If the name *local-file* is not specified, it is the same as on the remote machine. *Remote-file* may have the format *host*:*remote-file*, at which time a connection is established to the named remote host and *remote-file* is retrieved.

mode [*mode-name*]
: Set the file transfer *mode* to *mode-name*. The two possible modes are "ascii" and "binary"; "ascii" is the default. If *mode-name* is not specified, the current mode is printed.

put *local-file* [*remote-file*]
: Store *local-file* on the remote machine. If *remote-file* is left unspecified, *local-file* is used in naming the remote file. If *remote-file* has the form *host*:*remote-file*, a connection is established to the named host and *local-file* is sent to the remote host and named *remote-file*.

quit Terminate the TFTP session with the remote server and exit *tftp*.

status Show the current status of *tftp*.

trace Toggle packet tracing. By default, tracing is off.

verbose Toggle *verbose* mode. If **verbose** is on, when a file transfer completes, statistics regarding the efficiency of the transfer are reported. By default, **verbose** is off.

? [*command*]
: Print an informative message about the meaning of *command*. If no argument is given, *tftp* prints a list of the known commands.

rexmt *value*
: Set per-packet retransmission timeout to *value* seconds. Default *value* is five seconds.

timeout *value*
: Set total retransmission timeout to *value* seconds. Default *value* is fifteen seconds.

NAME

time – Times 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 and in the corresponding number of cpu clock cycles.

The times are printed on standard error.

NOTES

csh(1) has a built-in time with slightly different characteristics. See csh(1).

SEE ALSO

times(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

timex – Times a command; reports process data and system activity

SYNOPSIS

timex [**–pos**] *command*

DESCRIPTION

When you invoke the *timex* command, the given *command* is executed; the elapsed time, user time, and system time spent in execution are reported in seconds. Optionally, process accounting data for the *command* and all its children can be listed or summarized, and total system activity during the execution interval can be reported.

The output of *timex* is written on standard error.

The *timex* accepts the following options:

–p Lists process accounting records for *command* and all its children. Suboptions **f, h, k, m, r,** and **t** modify the data items reported, as defined in *acctcom*(1). The number of blocks read or written and the number of characters transferred are always reported.

–o Reports the total number of blocks read or written and total characters transferred by *command* and all its children.

–s Reports total system activity (not just that due to *command*) that occurred during the execution interval of *command*. All the data items listed in *sar*(1) are reported.

WARNING

Process records associated with *command* are selected from the accounting file **/usr/adm/pacct** by inference, since process genealogy is not available. Background processes having the same user-id, termid, and execution is spuriously included.

EXAMPLES

A simple example:

```
timex –ops sleep 60
```

A terminal session of arbitrary complexity can be measured by timing a subshell:

```
timex –opskmt sh

        session commands

EOT
```

SEE ALSO

acctcom(1), sar(1)

NAME

to2, from2, tovax, fromvax – Copies files between VAX and CRAY-2

SYNOPSIS

to2 *file* ... *target*
tovax *file* ... *target*
from2 *file* ... *target*
fromvax *file* ... *target*

DESCRIPTION

When running on the VAX, use *to2* to copy files to the CRAY-2, and use *from2* to copy files from the CRAY-2. When running on the CRAY-2, *tovax* and *fromvax* work analogously.

With *to2* and *tovax*, the *file* on the local machine is copied to *target* on the remote machine. With *from2* and *fromvax*, the *file* on the remote machine is copied to *target* on the local machine. If *target* is a directory, then one or more files are copied to that directory. If *file* or *target* does not begin with a '/', the current working directory name is prepended.

Protection is based on user name: if the user would be allowed to read and write the file when running directly on the remote machine, the copy is allowed. If the *target* file already exists, copying a file into *target* does not change its owner, group, nor mode. If the *target* file does not exist, its owner and group are set to that of the user, and its mode is set to 0644 (or the local default).

To2, from2, tovax, and *fromvax* are all shell files that execute a program called *hyft_send* (HYPERchannel file transfer send program). *Hyft_send* waits until the logical channel is free, then if called by *to2* or *tovax,* sends files over the channel to the *receive* daemon. Otherwise, if called by *from2* or *fromvax, hyft_send* sends a request for files to the *receive* daemon which invokes the proper *"to"* program to transfer the files back.

Before file transfers can take place, the super user must start up the *receive* daemon, *hyft_recv*.

MESSAGES

Messages from *to2* and *tovax* are sent to the standard error output. Diagnostics from *from2* and *fromvax* are sent to the controlling tty. Diagnostic messages should be self explanatory.

BUGS

Because *from2* uses the shell (**/bin/sh**) to expand wildcard characters, grave accents (`) are interpreted and cause unpredictable results.

From2 and *fromvax* always return successful exit status.

NOTES

This version of *hyft_send* operates synchronously. When the prompt returns, the file transfer is complete.

The unsupported **-d** option prints (mostly unintelligible) extra diagnostic information which is sometimes useful for determining if the link is up.

SEE ALSO

hyft(1M)

NAME

touch – Updates access and modification times of a file

SYNOPSIS

touch [**–amc**] [*mmddhhmm* [*yy*]] *files*

DESCRIPTION

The *touch* command causes the access and modification times of each argument to be updated. The filename is created if it does not exist. If no time is specified (see *date*(1)), the current time is used. If no options are specified, *touch* updates the access times and modification times. Valid options follow.

–a	Causes *touch* to update only the access times (default is **–am**).
–m	Causes *touch* to update only the modification times (default is **–am**).
–c	Silently prevents *touch* from creating the file if it did not previously exist.

MESSAGES

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) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

tput – Makes *terminfo* data available to shell

SYNOPSIS

tput [**–T***type*] *capname*

DESCRIPTION

The *tput* command uses the *terminfo*(4F) database to make terminal-dependent capabilities and information available to the shell. *Tput* outputs a string if the attribute (capability name or *capname)* is type string or an integer if the attribute is type integer. If the attribute is type boolean, *tput* sets the exit code (0 for TRUE, 1 for FALSE) and does no output.

The options and arguments to *tput* are as follows:

-T*type*	Indicates the type of terminal. Normally this flag is unnecessary; the default is taken from the environment variable **TERM.**
capname	Indicates the attribute from the *terminfo* database. See *terminfo*(4).

EXAMPLES

The following example echoes a clear-screen sequence for the current terminal:

```
tput clear
```

The following example prints the number of columns for the current terminal:

```
tput cols
```

The following example prints the number of columns for the 450 terminal:

```
tput -T450 cols
```

The following example sets a shell variable "bold" to stand-out mode sequence for the current terminal:

```
bold=‘tput smso‘
```

You can follow this with a prompt, as follows:

```
echo "${bold}Please type in your name: \c"
```

The following example sets the exit code to indicate if the current terminal is a hardcopy terminal:

```
tput hc
```

FILES

/etc/term/?/*	Terminal descriptor files
/usr/include/term.h	Definition files
/usr/include/curses.h	

MESSAGES

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 (for example, tput -T450 lines), **-1** is printed.

SEE ALSO

stty(1)
terminfo(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

tr – Translates characters

SYNOPSIS

tr [**–cds**] [*string1* [*string2*]]

DESCRIPTION

The *tr* command copies the standard input to the standard output with the 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 Compresses 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. I *n* is zero or if *n* is omitted, it defaults to "huge"; *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.

EXAMPLES

The following example 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 alphabetic characters. The strings are quoted to protect the special characters from interpretation by the shell; 012 is the ASCII code for a new-line character.

```
tr –cs "[A–Z][a–z]" "[\012*]" <file1 >file2
```

BUGS

Tr does not recognize ASCII NUL in *string1* or *string2*; it always deletes NUL from input.

SEE ALSO

ed(1), sh(1)

NAME

true, false – Provides truth values about processor type

SYNOPSIS

true

false

DESCRIPTION

The *true* command does nothing, successfully; it returns a 0 as an exit code. *False* does nothing, unsuccessfully; it returns a non-zero as an exit code. They are typically used in input to *sh*(1) such as:

```
while true
do
        command
done
```

MESSAGES

True has exit status zero, *false* nonzero.

SEE ALSO

sh(1)

NAME

tset, reset – Terminal dependent initialization

SYNOPSIS

tset [**–e***c*] [**–k***c*] [**–nIQ**] [**–m**] [*ident*][*test baudrate*]:*type*] ... [*type*] [**–**]

reset ...

DESCRIPTION

The *tset* command sets up your terminal when you first log in to a UNICOS system. It performs 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 performs necessary initializations and mode settings. Type names for terminals can be found in the *terminfo*(4) database. If a port is not wired permanently to a specific terminal (not hardwired), it is given an appropriate generic identifier such as *dialup*.

If you do not specify any arguments, *tset* simply reads the terminal type out of the environment variable **TERM** and reinitializes the terminal.

When used in a startup procedure (**.profile** for *sh*(1) users or **.login** for *csh*(1) users) it is desirable to provide information about the type of terminal you usually use on ports that are not hardwired. These ports are identified as *dialup*, *plugboard,* or *arpanet,* and so on. To specify what terminal type you usually use on these ports, the **–m** (map) option is followed by the appropriate port type identifier (*ident*), an optional baud rate specification (*baudrate*), and the terminal type (*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, assume that I'm on that kind of terminal".) If you specify more than one mapping, the first applicable mapping prevails. A missing port type identifier matches all identifiers. Any of the alternate generic names given in *terminfo* can be used for the identifier.

A *baudrate* is specified as with *stty*(1), and is compared with the speed of the diagnostic output (which should be the control terminal). The baud rate *test* may be any combination of: >, @, <, and !; @ means "at" and ! inverts the sense of the test. To avoid problems with metacharacters, it is best to place the entire argument to **–m** within "´" characters; *csh*(1) users must also put a "\" before any "!" used here.

Thus

```
tset –m ´dialup>300:adm3a´ –m dialup:dw2 –m ´plugboard:?adm3a´
```

causes the terminal type to be set to an *adm3a* if the port in use is a dialup at a speed greater than 300 baud; to a *dw2* if the port is (otherwise) a dialup (that is, at 300 baud or less). If the *type* finally determined by *tset* begins with a question mark, you are asked if you really want that type. A null response means you do want that type; otherwise, you can enter another type which is then used instead. Thus, in the above case, you will be queried on a plugboard port as to whether you are actually using an *adm3a*.

If no mapping applies and you specify a final *type* option (not preceded by a **–m**) on the command line, that type is used; otherwise the identifier found in the environment is taken to be the terminal type. This 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. You can do this by using the **–** option. For example, using the Bourne shell, *sh*(1):

```
export TERM; TERM=`tset – options...`
```

or using the C shell, *csh*(1):

```
setenv TERM `tset – options...`
```

With *csh*, it is convenient to make an alias in your **.cshrc:**

```
alias tset 'setenv TERM `tset – \!*`'
```

Either of these aliases allow the command

```
tset 2621
```

to be invoked at any time from your login csh. **Note to Bourne Shell users:** It is **not** possible to get this aliasing effect with a shell procedure because shell procedures cannot set the environment of their parent.

These commands cause *tset* to place the name of your terminal in the variable TERM in the environment; see *environ*(4).

Once the terminal type is known, *tset* engages in terminal driver mode setting. This normally involves sending an initialization sequence to the terminal, setting the single character erase (and optionally the line-kill (full line erase)) 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 (CONTROL-H).

The following options are available:

–e*c* Sets the erase character to the named character *c* on all terminals. The default is the backspace character on the terminal, usually CONTROL-H. The character *c* can either be typed directly, or entered using the hat notation used here.

–k*c* Sets the line Kill character to the named character c on all terminals; *c* defaults to CONTROL-U. The kill character is left alone if **–k** is not specified. The hat notation can also be used for this option.

–n On systems with the Berkeley 4BSD tty driver, **–n** specifies that the new tty driver modes should be initialized for this terminal. For a CRT, the CRTERASE and CRTKILL modes are set only if the baud rate is 1200 or greater. See *tty*(4) for more details.

–I Suppresses transmitting terminal initialization strings

–Q Suppresses printing the "Erase set to" and "Kill set to" messages

– Specifies the name of the terminal finally decided upon to be output on the standard output. This is intended to be captured by the shell and placed in the environment variable **TERM.**

If **tset** is invoked as **reset**, it sets cooked and echo modes, turns off cbreak and raw modes, turns on newline translation, and restores special characters to a sensible state before any terminal dependent processing is done. Any special character found to be NULL or "–1" is reset to its default value.

This is most useful after a program dies leaving a terminal in an unusual state. You may have to type "<LF>reset<LF>" to get it to work since <CR> may not work in this state. Often none of this will echo.

EXAMPLES

These examples all assume the Bourne shell and use the – option. If you use *csh*, use one of the variations described above. Note that a typical use of *tset* in a **.profile** or **.login** also use the **–e** and **–k** options, and often the **–n** or **–Q** options as well. (**NOTE:** some of the examples given here appear to take up more than one line, for text processing reasons. When you type in real *tset* commands, you must enter them entirely on one line.)

At the moment, you are on a 2621. This is suitable for typing by hand but not for a **.profile**, unless you are *always* on a 2621.

```
export TERM; TERM=`tset – 2621`
```

You have an h1,9 at home that you dial up on, but your office terminal is hardwired and known in **/etc/ttytype**.

```
export TERM; TERM=`tset – –m dialup:h19`
```

You have a switch that connects everything to everything, making it nearly impossible to indicate on which port you are coming. You use a vt100 in your office at 9600 baud, and dial up to switch ports at 1200 baud from home on a 2621. Sometimes you use someone elses 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 2621. Note the placement of the question mark, and the quotes to protect the greater than and question mark from interpretation by the shell.

```
export TERM; TERM=`tset – –m 'switch>1200:?vt100' –m 'switch<=1200:2621'
```

All of the above entries will fall back on the terminal type specified in the environment 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 adm3a. It always asks you what kind of terminal you are on, defaulting to adm3a.

```
export TERM; TERM=`tset – ?adm3a`
```

If the environment is not properly initialized and you want to key entirely on the baud rate, the following can be used:

```
export TERM; TERM=`tset – –m '>1200:vt100' 2621`
```

Here is a fancy example to illustrate the power of *tset* and to hopelessly confuse anyone who has made it this far. You dial up at 1200 baud or less on a concept100, sometimes over switch ports and sometimes over regular dialups. You use various terminals at speeds higher than 1200 over switch ports, most often the terminal in your office, which is a vt100. However, sometimes you log in from the university you used to go to, over the ARPANET; in this case you are on an ALTO emulating a dm2500. You also often log in on various hardwired ports, such as the console, all of which are properly entered in the environment. You want your erase character set to control H, your kill character set to control U, and don't want *tset* to print the "Erase set to Backspace, Kill set to Control U" message.

```
export TERM; TERM=`tset –e –k^U –Q – –m 'switch<=1200:concept100' –m 'switch:?vt100'
–m dialup:concept100 –m arpanet:dm2500`
```

FILES

/usr/lib/terminfo terminal capability database

SEE ALSO

csh(1), sh(1), stty(1)
terminfo(4), environ(4) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

tsort – Performs a 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 *file* is not specified, the standard input is used.

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

Tsort uses a quadratic algorithm; not worth fixing for the typical use of ordering a library archive file.

NAME

tty – Gets the name of the terminal

SYNOPSIS

tty [**–l**] [**–s**]

DESCRIPTION

The *tty* command prints the path name of the user's terminal. Valid options are:

–l Prints the synchronous line number to which the user's terminal is connected, if it is on an active synchronous line.

–s Inhibits printing of the terminal's path name, allowing one to test just the exit code.

EXIT CODES

2 – Invalid options were specified
0 – Standard input is a terminal
1 – Otherwise

MESSAGES

Not on an active synchronous line
    The standard input is not a synchronous terminal and **–l** is specified.

Not a tty
    The standard input is not a terminal and **–s** is not specified.

NAME

ul – Underlines text

SYNOPSIS

ul [**–i**] [**–t** *terminal*] [*name* ...]

DESCRIPTION

The *ul* command reads named files (or standard input if no named files are given) and translates underscores to the sequence that indicates underlining for the terminal in use. The –t option overrides the terminal type specified in the environment (by the environment variable TERM).

The file **/usr/lib/terminfo** contains the appropriate sequences for underlining. If the terminal cannot underline but is capable of a standout mode, then that mode is used. If the terminal can overstrike, or can handle underlining automatically, *ul* degenerates to *cat*(1). If the terminal cannot underline, underlining is ignored.

The –i option causes *ul* to indicate underlining (with dashes '–') on a separate line than the affected text. This is useful when you want to look at the underlining present in an *nroff* output stream on a crt-terminal.

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.

SEE ALSO

man(1), nroff(1), colcrt(1)

NAME

umask – Sets file-creation mode mask

SYNOPSIS

umask [*ooo*]

DESCRIPTION

The user file-creation mode mask is set to *ooo*. The three octal digits (*ooo*) 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 (so that files normally created with mode **777** become mode **755** and files created with mode **666** become mode **644**).

If *ooo* is omitted, the current value of the mask is printed.

Umask is recognized and executed by the shell.

SEE ALSO

chmod(1), sh(1)
chmod(2), creat(2), umask(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

uname – Prints name of current system

SYNOPSIS

uname [-snrvma]

DESCRIPTION

The *uname* command prints the current system name on the standard output file. It is mainly useful to determine what 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 node name (the node name may be a name by which system is known to a communications network).
-r	Print the operating system release number.
-v	Print the operating system version number.
-m	Print the machine hardware name.
-a	Print all the above information in the format: *system node operating system version hardware*

SEE ALSO

uname(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

unget – Undoes 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.

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 *gets* 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.
–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.

MESSAGES

Use *help*(1) for explanations.

SEE ALSO

delta(1), get(1), sact(1)
the UNICOS Source Code Control System (SCCS) User's Guide, publication SG-2017

NAME

uniq – Reports repeated lines in a file

SYNOPSIS

uniq [–udc [+ *n* **] [–** *n* **]] [** *input* **[** *output* **]]**

DESCRIPTION

The *uniq* command reads the input file comparing adjacent lines. By default, *uniq* removes the second and succeeding copies of repeated lines the remainder is written on the output file. *Input* and *output* should always be different files. Note that repeated lines must be adjacent in order to be found; see *sort*(1). If the **–u** option is used, only the lines that are not repeated in the original file are output. The **–d** option specifies that one copy of only the repeated lines is to be written. The normal mode output is the union of the **–u** and **–d** mode output.

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)

NAME

units – Unit conversion program

SYNOPSIS

units

DESCRIPTION

The *units* command 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, division by the usual sign:

```
You have:  15 lbs force/in2
You want:  atm
        * 1.020689e+00
        / 9.797299e-01
```

The *units* command only does multiplicative scale changes; for example, 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, (such as, **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
```

You must hit CONTROL-d to exit the program.

FILES

/usr/lib/unittab

MESSAGES

Conformability	You specified mutually incompatible scales
Cannot recognize *string*	Unknown keyword

NOTES

To exit program, you must key in a CONTOL-d character.

NAME

update – Invokes the UPDATE utility

SYNOPSIS

update [**–p** *pdf*] [**–i** *idf*] [**–c** *cdf*] [**–a** *x1 x2 ... xN*] [**–u** *udf*] [**–d** *ifp*] [**–n** *ndf*] [**–s** *sdf*] [**–x** *chr*] [–y *chr*] [**–w** *dwv*] [**–m** *n*] [**–f** **–q** *qf1 qf2 ... qfn*] [**–o** *options*]

DESCRIPTION

The *update* command is a source code maintenance utility that is also available under the Cray operating system COS.

The options for *update* are as follows:

–p *pdf* PL (program library) file name. This file is created by *update* and is intended to be used only by *update*. The keyword alone is invalid. Omit the **–p** parameter for creation runs.

–i *idf* Input file name. Use of multiple input file names is deferred. The **–i** parameter must be specified for creation runs. The keyword alone is invalid.

–c *cdf* Compile file name. For multiple compile files, the directory name is appended with a period and a single letter as defined by default (**f**,**s**, **p**, and **c**) or specified by the **–a** parameter. The file name and the following period and suffix should not exceed 14 characters (thus the input file name must not exceed 12 characters). The files must not exist as they will be created by *update*. The keyword alone is invalid. The default suffixes are assigned in the following order:

f	Fortran
s	Assembly
p	Pascal
c	C

–a *x1 x2 ... xN*

Specifies the appended characters that are to be assigned to multiple compile files. Each character must be unique and of the 62 alphanumeric characters. If *x1* is a single character, the file name will be appended with a period followed by the character. The keyword alone is invalid.

–u *udf* User common deck directory. Deferred implementation

–d *ifp* Defines names to be used with an IF directive (up to 8 characters may be used). Defined names need not be unique from deck names, common deck names, or modification identifiers. They are known only in the *update* run being processed. Up to 100 names may be defined with the **–d** parameter. The keyword alone is invalid.

–n *ndf* New PL file name. This file is created by, and used only by, *update*. For a normal *update* run, **–n** *ndir* must be specified and the **–f** and **–q** parameters omitted. The *update* mode determines the contents of the new PL. The specified file must be empty as *update* will create a PL in it. The keyword alone is invalid.

–s *sdf* Source file name. The *update* mode determines the contents of this file. The file can be the input for a subsequent creation run, but the file name must not exist on input as it will be created by *update*. The keyword alone is invalid.

–x *chr* Master character; the first character of directives read from the input file or written to the source file. Invalid master characters are comma, period, colon, equal sign, and space. If the character is a metacharacter, it must be quoted or escaped. For example, an * must be represented in one of the following ways: *, "*", or '*'. The keyword alone is invalid.

If omitted in a creation run, the master character for directives is *. If omitted in a modification run, the master character is that used in the creation run for the PL.

–y *chr* Comment character that indicates a comment. If the character is a metacharacter, it must be quoted or escaped. For example, an * must be represented in one of the following ways: *, "*", or '*'. The keyword alone is invalid.

If omitted in a creation run, the comment character is /. If omitted in a modification run, the comment character is that used in the creation run for the PL.

–w *dwv* Data width value; the number of characters of data written to each line in the compile and source file. The *dwv* range is 1 through 256. The keyword alone is invalid.

If this keyword is omitted in a creation run, columns 1 through 72 contain data. Otherwise, columns 1 through *dwv* contain data.

If **–w '*'** is specified in a creation run, variable length records are written to the compile and source files. (The asterisk is a metacharacter and must be quoted or escaped.)

In a modification run, if the PL was created with **–w '*';**, only **–w '*'** or no **–w** parameter are accepted, and *update* continues to process variable length text lines for the PL.

In a modification run with **–w** unspecified, columns 1 through *lastdw* contain data; *lastdw* is the **–w** value specified when the PL was written. If **–w dwv** is specified, columns 1 through *dwv* contain data.

The number of characters per line written to the new PL is the maximum of either *dwv*, *pldw*, or 80; *pldw* is the number of characters per line in the existing regular PL. For variable-length record PLs, the same number of characters per line appears in the new PL as was in the old PL.

For regular PLs, sequence information is provided as follows:

- When the data width value is omitted or specified as *dwv*, *dwv+1* through *dwv+8* contain an identifier, right-justified with leading spaces; *dwv+9* contains a period; *dwv+10* through *dwv+15* contain a sequence number, left-justified with trailing spaces.
- When the data width value is specified as *ldwv*, the entire sequencing field of the compile output is left-justified.
- For variable-length record PLs, no sequence information appears in the compile file. Source file sequence information, if requested, appears a few spaces to the right of the end of the text line.

–m *n* Message level; the highest level of severity for *update* listing messages to be suppressed.

The following levels are available:

Level	Severity	Description
1	COMMENT	Currently unused
2	NOTE	Information not related to errors
3	CAUTION	Possible error
4	WARNING	Probable error
5	ERROR	Fatal error

For example, **–m 2** allows CAUTION, WARNING, and ERROR messages to be printed to **stdout**. The default used when **–m** is omitted is 3. The keyword alone is invalid.

–f –q (omitted)
Full, quick, or normal *update* run. This determines the compile file contents, the source file, and the new PL contents.

–f Full *update* mode; all active lines are processed. The PL Identifier Table determines the sequence. No COMPILE directive is necessary. The keyword must be used alone.

–q *qf1 qf2 ... qfn*
or

-q'*d1,d2,...,dj.dk,...,dn*'
Quick *update* mode. Decks that are specified with the **–q** parameter and decks specified by a COMPILE directive are written to the compile file and/or the source file, and to the new PL. These decks must be uppercase. Externally defined common decks cannot be named with the **–q** parameter or the COMPILE directive. The PL Identifier Table determines the sequence unless the **k** option is used. Corrections to decks that are not specified with the **–q** parameter or by a COMPILE directive are not included.

In the first method shown, up to 100 decks can be specified. After all input has been entered, unknown deck names are errors. The keyword alone is invalid.

In the second method shown, single decks are separated by commas, and ranges of decks are separated by periods. After all input has been entered, unknown deck names are errors. The maximum size of the string used with the second method is 96 characters. The two methods cannot be combined.

omitted Normal *update* mode. Decks specified by COMPILE directives, modified decks, and decks calling modified common decks are written to compile and/or source files. Externally defined common decks cannot be named with the COMPILE directive, and they are not written to the source file. For a normal mode run, the **–n** parameter must be specified; a new PL will be created.

–o *options* (keyword only)
The following output options are available in the command line. Each option must be separated by a space.

cd Write the generation directives for the compile file to **stdout**.

dc Declared modifications option. This ensures that modifications apply to the correct deck or common deck. Declaration of PL modifications might be required.

ed Write the edited line summary to **stdout**.

id Write the identifier summary to **stdout**.

if	Write a conditional text summary to **stdout**.
in	List the input to **stdout**.
k	Order all decks that are written to the compile file, as directed by the **–q** parameter values on the control statement and the COMPILE directives. This option is ignored in full and normal modes. If a modification set affects two or more decks and the **k** option is in effect, the sequence numbers of inserted lines can be inconsistent with sequencing that has occurred without the **k** option.
na	Do not abort if directive errors or modification errors occur. All requested files are generated.
nr	Do not "rewind" the source or compile files at the end of *update* execution.
ns	Suppress line sequence information in the compile files. SEQ and NOSEQ directives are ignored when this option is used. **Ns** has no affect on the source file output and is ignored for variable-length record PLs.
sq	Update the source output that is provided with sequencing information. **Sq** has no affect on the compile file output.
um	Write unprocessed modifications to **stdout**.

EXAMPLES

```
update –n newpl –i input –c cplfile
```

This example shows how a PL is created. The omitted **–p** parameter indicates that there is no existing PL. The new PL is written to the file **newpl** and the input is read from **input**. The compile output is written to **cplfile**; all decks are selected.

```
update –p plname –i mods –q DECK3 DECK2 DECK4 –o k na –c cplfile
```

This example shows a modification of a PL. The parameters indicate the following:

The PL file is **plname** in the current directory.

- The input is read from the file **mods** in the current directory.
- Quick mode with the **k** output option. If a single COMPILE directive is used (*COMPILE DECK1.DECK4), DECK1 through DECK4 are written to the compile file (**–c cplfile**) in the following order:
  DECK3 DECK2 DECK4 DECK1
- *Update* does not abort if directive or modification errors occur.

SEE ALSO

The UPDATE Reference Manual, publication SR-0013

NAME

uucp, uulog, uuname – UNIX system to UNIX system copy

SYNOPSIS

uucp [**–dfcrjC**] [**–m***file*] [**–n***user*] [**–e***sys*] source-files destination-file

uulog [**–s**[*sys*]] [**–u**[*user*]]

uuname [**–l**] [**–v**]

DESCRIPTION

The *uucp* command 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 that *uucp* knows about. The *system-name* can 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 using the specified route, and only to a destination in **PUBDIR** (see below). Ensure that intermediate nodes in the route are willing to forward information.

The shell metacharacters **?**, ***** and **[...]** appearing in *path-name* are expanded on the appropriate system.

Path names may be one of the following:

- A full path name
- 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
- 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**
- Anything else is prefixed by the current directory

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 *uucp* command preserves execute permissions across the transmission and gives 0666 read and write permissions (see *chmod*(2)).

The following options are interpreted by *uucp*:

–d	Makes all necessary directories for the file copy (default)
–f	Does not make intermediate directories for the file copy
–c	Uses the source file when copying out rather than copying the file to the spool directory (default)
–C	Copies the source file to the spool directory
–m*file*	Reports status of the transfer in *file*. If *file* is omitted, sends mail to the requester when the copy is completed.
–r	Queues job but does not start the file transfer process. By default a file transfer process is started each time *uucp* is evoked.
–j	Controls writing of the *uucp* job number to standard output (see below).

–n*user* Notifies *user* on the remote system that a file was sent

–e*sys* Sends 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**.)

The *uucp* command 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 is not listed (default). If *uucp* is then invoked with the **–j** option, the job number is listed. If the environment variable **JOBNO** is set to **ON** and is exported, a job number is written to standard output each time uucp is invoked. In this case, the **–j** option supresses output of the job number.

The *uulog* command 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*] Prints information about work involving system *sys*. If *sys* is not specified, then logging information for all systems is printed.

–u[*user*]
Prints information about work done for the specified *user*. If *user* is not specified then logging information for all users is printed.

The *uuname* command lists the uucp names of known systems. The **–l** option returns the local system name. The **–v** option prints additional information about each system. A description is 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/spool/uucp	Spool directory
/usr/spool/uucppublic	Public directory for receiving and sending (PUBDIR)
/usr/lib/uucp/*	Other data and program files

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 **~nuucp** or just **~**).

NOTES

In order to send files that begin with a dot (.profile) the files must by qualified with a dot. For example: **.profile, .prof*, .profil?** are correct; whereas *prof*, ?profile are incorrect.

The *uucp* command does not generate a job number for a strictly local transaction.

BUGS

All files received by *uucp* are owned by *uucp*.
The **–m** option only works sending files or receiving a single file. Receiving multiple files specified by special shell characters **? * [...]** does not activate the **–m** option.
The **–m** option does not work if all transactions are local or if **uucp** is executed remotely via the **–e** option.
The **–n** option functions only when the source and destination are not on the same machine.
Only the first six characters of a *system-name* are significant. Any excess characters are ignored.

SEE ALSO

mail(1), uux(1).
chmod(2) in the UNIX System Programmer Reference Manual.

NAME

uustat – Uucp status inquiry and job control

SYNOPSIS

uustat [**–jkr***jobn*] [**–coy***hour*] [**–u***user*] [**–s***sys*] [**–mM***mch*] [**–Oq**]

DESCRIPTION

The *uustat* command displays the status of, or cancels, previously specified *uucp* commands. It will also provide the general status on *uucp* connections to other systems. The following options are recognized:

–j*jobn*	Reports 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 is printed and the request fails.
–k*jobn*	Kills the *uucp* request whose job number is *jobn*. The killed *uucp* request must belong to the person issuing the *uustat* command unless one is the super user.
–r*jobn*	Rejuvenates *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 that are older than *hour* hours. This administrative option can only be initiated by the user *uucp* or the super user.
–o*hour*	Reports the status of all *uucp* requests which are older than *hour* hours
–y*hour*	Reports the status of all *uucp* requests which are younger than *hour* hours
–u*user*	Reports the status of all *uucp* requests issued by *user*
–s*sys*	Reports the status of all *uucp* requests which communicate with remote system *sys*
–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.
–O	Reports the *uucp* status using the octal status codes listed below. If you do not specify this option, the verbose description is printed with each *uucp* request.
–q	Lists 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 options are omitted, *uustat* outputs the status of all *uucp* requests issued by the current user. Only one of the options **–j**, **–m**, **–k**, **–c**, **–r**, can be used with the rest of the other options.

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** can 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/spool/uucp	Spool directory
/usr/lib/uucp/L_stat	System status file
/usr/lib/uucp/R_stat	Request status file

SEE ALSO

uucp(1)

NAME

uuto, uupick – Public UNICOS-to-UNICOS system file copy

SYNOPSIS

uuto [**–p**] [**–m**] *source-files destination*
uupick [**–s** *system*]

DESCRIPTION

The *uuto* command sends *source-files* to *destination*. It 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!fluser

where *system* is taken from a list of system names that *uucp* knows about (see *uuname*). User is the login name of someone on the specified system.

Two options are available:

–p	Copies the source file into the spool directory before transmission
–m	Sends mail to the sender when the copy is complete

The files (or sub-trees if directories are specified) are sent to **PUBDIR** on *system*, where **PUBDIR** is a public directory defined in the *uucp* source. Specifically the files are sent to

PUBDIR/receive/*user*/*mysystem*/files.

The destined recipient is notified by *mail*(1) of the arrival of files.

The *uupick* command accepts or rejects the files transmitted to the user. 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*] ?

The *uupick* command 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).
a [*dir*]	Same as **m** except moving all the files sent from *system*.
p	Print the content of the file.
q	Stop.
EOT (control-d)	Same as **q**.
!*command*	Escape to the shell to do *command*.
*	Print a command summary.

If *uupick* is invoked with the –s *system* option, it only searches 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, the files must by qualified with a dot. For example: .profile, .prof*, .profil? are correct; whereas *prof*, ?profile are incorrect.

SEE ALSO

mail(1), uucp(1), uustat(1), uux(1)
uuclean(1M) in the UNICOS Administrator Commands Reference Manual, publication SR-2022

NAME

uux – UNIX-to-UNIX/UNICOS-to-UNICOS system command execution

SYNOPSIS

uux [**–nj**] [**–m***file*] [**–**] *command-string*

DESCRIPTION

The *uux* command gathers zero or more files from various systems, executes a command on a specified system, and then sends standard output to a file on a specified system. For security reasons, many installations limit the list of commands executable on behalf of an incoming request from *uux*. Many sites permit little more than the receipt of mail (see mail (1)) using *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 the following:

- A full path name
- 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
- 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.

The *uux* command attempts to get all files to the execution system. For output files, the file name must be escaped using parentheses. For example, the command

```
uux a!uucp b!/usr/file \(c!/usr/file\)
```

sends a *uucp* command to system ''a'' to get **/usr/file** from system ''b'' and send it to system ''c''.

The *uux* command notifies you if the requested command on the remote system was disallowed. The response comes by remote mail from the remote machine. 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 you do not specify any machines, then any machine can execute **cmd**. If you specify machines, only the listed machines can execute **cmd**. If the desired command is not listed in **L.sys**, it cannot be executed.

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*:

–n	Sends no notification to user
–j	Controls writing of the *uucp* job number to standard output.
–m*file*	Reports status of the transfer in *file*. If *file* is omitted, sends mail to the requester upon copy completion.

– The standard input to *uux* is made the standard input to the *command-string*.

The *uux* command 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 controls the listing of the *uux* job number on standard output. If the environment variable **JOBNO** is undefined or set to **OFF**, the job number is not listed (default). If **uucp** is then invoked with the **–j** option, the job number is listed. If the environment variable **JOBNO** is set to **ON** and is exported, a job number is written to standard output each time *uux* is invoked. In this case, the **–j** option suppresss output of the job number.

FILES

/usr/spool/uucp	Spool directory
/usr/spool/uucppublic	Public directory (PUBDIR)
/usr/lib/uucp/*	Other data and programs

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.
Only the first six characters of the **system-name** are significant. Any excess characters are ignored.
The shell metacharacter * will probably not do what you want it to do. The shell tokens << and >> are not implemented.

SEE ALSO

mail(1), uucp(1)
uuclean(1M) in the UNICOS Administrator Commands Reference Manual, publication SR-2022

NAME

val – Validates 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.
–r*SID*	The argument value *SID* (*S*CCS *ID*entification String) is an SCCS delta number. A check is made to determine if the *SID* is ambiguous (such as, r1 is ambiguous because it physically does not exist but implies 1.1, 1.2, etc., which may exist) or invalid (such as, r1.0 or r1.1.0 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, that is, 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.

MESSAGES

Use *help*(1) for explanations.

LIMITATIONS

Val can process up to 50 files on a single command line. Any number above 50 will produce a core dump.

SEE ALSO

admin(1), delta(1), get(1), prs(1)
the UNICOS Source Code Control System (SCCS) User's Guide, publication SG-2017

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 error) 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** statements 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 (such as, : 012 > 12 is false). All other operators take strings as arguments (such as, : 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.

MESSAGES

Use *help*(1) for explanations.

Vc returns an exit code of 0 on normal termination and returns 1 if any error occurs.

SEE ALSO

ed(1), help(1)

NAME

vi – Screen-oriented (visual) display editor based on ex(1)

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* ...

DESCRIPTION

The *vi* command (visual) invokes 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. (If you are not familiar with *ex*, please refer to *ex*(1).) Intelligent and high speed terminals are very pleasent to use with *vi*.

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.

The following command line 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.
name	Indicates files to be edited.

The *view* command is the same as *vi* except that the **readonly** flag is set.

The *vedit* command 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 *vi*.

VI MODES

At any one time, you are in one of the following modes within *vi*.

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 can then be entered. Input mode is normally terminated with ESC or abnormally with interrupt.
Last line	Reading input for : / ? or !; terminate with CR to execute, interrupt to cancel.

COMMAND SUMMARY

Following is a summary of commands *vi* accepts.

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	... 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

Numbers 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*
: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	noth line after *pat*
?pat?–n	noth 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 bottom
z.CR	... at center
*/pat/***z–CR**	*pat* line at bottom
z*n***.CR**	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	backtab over *autoindent*
↑^D	kill *autoindent*, save for next
0^D	... but at margin next also
^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, for example, **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

CAUTIONS AND BUGS

Software tabs using CONTROL-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.

There should be an interactive *help* facility and a tutorial suited for beginners.

SEE ALSO

ex (1)

NAME

wait – Awaits completion of process

SYNOPSIS

wait

DESCRIPTION

The *wait* command waits until all processes started with & have completed and reports 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.

Note that not all the processes of a 3-stage or more pipeline are children of the shell, and thus *wait* cannot wait for these processes to start and complete.

SEE ALSO

sh(1)
wait(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

wc – Counts words, lines, and characters in a file

SYNOPSIS

wc [**–lwc**] [*names*]

DESCRIPTION

The *wc* command 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-line characters.

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, respectively. All are printed by default.

When *names* are specified on the command line, they will be printed along with the counts.

NAME

wdrop – Writes recoverable drop file

SYNOPSIS

wdrop [**–l**] [*–m*] [*pid*]

DESCRIPTION

The *wdrop* command writes a recoverable image to either the **system/drop** device or the user's local (current working) directory. The format of the file name is **drop.** *XXXXXXX* where *XXXXXXX* is the process ID. The drop filename format is built in and cannot be changed at the user level.

If you specify the **–l** option, *wdrop* writes the drop file to the current working directory. If you specify the –m option, the requesting process image will be written. If you do not specify a process id (*pid*), *wdrop* writes the requesting process image. Subsequent writes of the same process to the same directory overwrites the previous process image in that directory.

The valid range of *pid* is from 0 to MAXPID. Super user privileges are required for a *pid* value of 1(INT).

If you do not specify any arguments with *wdrop*, it writes the current process image to the **system/drop** directory.

SEE ALSO

rdrop(1)
wdrop(2), rdrop(2) in the UNICOS System Calls Reference Manual, publication SR-2012

NAME

what – Identifies SCCS files

SYNOPSIS

what [–s] *files*

DESCRIPTION

What searches the given files for all occurrences of the pattern that *get*(1) substitutes for *%Z%* (this is @(#) and prints out what follows until the first ", >, new-line, \ or null character. For example, if the C program in file **f.c** contains

```
char ident[] = "@(#)identification information";
```

and **f.c** is compiled to yield **f.o** and **a.out**, then the command

```
what f.c f.o a.out
```

will print

```
f.c:     identification information
f.o:     identification information
a.out:   identification information
```

What is intended to be used in conjunction with the command *get*(1), which automatically inserts identifying information, but it can also be used where the information is inserted manually. Only one option exists:

-s Quit after finding the first occurrence of pattern in each file.

MESSAGES

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.

SEE ALSO

get(1), help(1)
the UNICOS Source Code Control System (SCCS) User's Guide, publication SG-2017

NAME

whereis – Locates source, binary, and or manual for program

SYNOPSIS

whereis [**–sbm**] [**–u**] [**–SBM** *dir* ... **–f**] *name* ...

DESCRIPTION

The *whereis* command 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*", for example ".c". Prefixes of "s." resulting from use of source code control are also dealt with. The *whereis* command then attempts to locate the desired program in a list of standard places. If you specify **–b, –s** or **–m** then *whereis* searches only for binaries, sources, or manual sections respectively (or any two thereof). You can use the **–u** option 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 that have no documentation.

Finally, you can use the **–B –M** and **–S** options to change or otherwise limit the places where *whereis* searches. The **–f** option terminates the last such directory list and signal the start of file names.

EXAMPLE

The following commands find all the files in **/usr/bin** that are not documented in **/usr/man/man11** with source in **/usr/src/cmd:**

```
cd /usr/bin
whereis –u –M /usr/man/man11 –S /usr/src/cmd –f *
```

FILES

/usr/src/*
/usr/man/*
/lib, /etc, /usr/{lib,bin,ucb,lbin}

BUGS

Since the program uses *chdir*(2) to run faster, path names given with the **–M, –S,** and **–B** must be full; that is, they must begin with a "/".

NAME

who – Reports who is on the system

SYNOPSIS

who [–uTlpdbrtasqH] [*file*]

who am i

DESCRIPTION

The *who* command can list the user 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 UNICOS user. It examines the file **/etc/utmp** to obtain its information. If *file* is given, that file is examined instead. Usually, *file* will be **/etc/wtmp**, which contains a history of all the logins since the file was last created.

The *who* command with the **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*]

The *name* is the user's login name. 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. 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*(4F)). This can contain commentary information.

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 Lists information about those users who are currently logged on the system.

–T Causes the *state* of the terminal line to be printed.

–l 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 doesn't exist.

–p 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*(4F).

–d 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 Indicates the time and date of the last reboot.

–r Indicates the current *run-level* of the *init* process. Following the run-level and date information are three fields which indicate the current state, the number of times that state was previously entered, and the previous state.

–t Indicates the last change to the system clock (via the *date*(1) command) by **root**. See *su*(1).

–a Processes **/etc/utmp** or the named *file* with all options turned on.

–s Is the default and lists only the *name*, *line* and *time* fields.

–q Prints the login names of the current users and gives the total number of users. This option is available on CRAY-2 systems only.

–H Prints a header. This option is available on CRAY-2 only.

FILES

/etc/utmp
/etc/wtmp
/etc/inittab

SEE ALSO

date(1), login(1), mesg(1), su(1), init(1M)
wait(2) in the UNICOS System Calls Reference Manual, publication SR-2012
inittab(4F), utmp(4F) in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014

NAME

write – Lets you write to another user

SYNOPSIS

write *user* [*line*]

DESCRIPTION

The *write* command copies lines from your terminal to that of another user. When first called, it sends the message:

> Message from *yourname* (tty##) [*date*] ...

to the other person. 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 or an interrupt is sent. 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 (such as, **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 to another user's terminal may be denied or granted by use of the *mesg*(1) command. Writing to others is normally allowed by default. Certain commands, in particular *pr*(1) inhibits 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 a line. Each person should end a message with a distinctive signal (that is, (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 !

MESSAGES

User not logged in
: The person you are trying to *write* to is not logged in.

SEE ALSO

mail(1), mesg(1), pr(1), sh(1), who(1)

NAME

xargs – Constructs argument lists and executes a command

SYNOPSIS

xargs [*flags*] [*command* [*initial-arguments*]]

DESCRIPTION

The *xargs* command combines the fixed *initial-arguments* with arguments read from standard input to execute *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 the user'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 *arguments*. This process is repeated until there are no more *arguments*. When there are flag conflicts (such as **–l** with **–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 when the *–l* option is used.

–i*replstr* Insert mode: The *command* is executed for each line from standard input, taking the entire line as a single argument, 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 ignored. Constructed arguments may not be larger than 255 characters. Option **–x** is forced when this option is used. { } 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, otherwise *xargs* terminates execution.

–t Trace mode: The *command* and each constructed argument list are echoed to standard error 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. Underscore (_) is assumed for the logical **EOF** string if **−e** is not coded. The value **−e** with no *eofstr* specified turns off the logical **EOF** string capability (underscore is taken literally). *Xargs* reads standard input until either end-of-file or the logical **EOF** string is encountered.

Xargs will terminate if it receives a return code of −1 from *command* 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 *mv*(1) 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

find(1), sh(1)

NAME

yacc – Yet another compiler-compiler

SYNOPSIS

yacc [–vdlt] *grammar*

DESCRIPTION

The *yacc* command 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. It produces the **y.tab.c** file as output.

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*.

Run-time 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.

Valid options for *yacc* are as follows:

–v The **y.output** file is prepared, which contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.

–d The **y.tab.h** file 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.

–l The code produced in **y.tab.c** does not contain any **#line** constructs. This should only be used after the grammar and the associated actions are fully debugged.

–t Changes the default to include run-time debugging code when **y.tab.c** is compiled

Whether or not the **–t** option was used, the run-time debugging code is under the control of **YYDEBUG**, a preprocessor symbol. If **YYDEBUG** has a nonzero value, the debugging code is included. If its value is 0, the code is not included. The size and execution time of a program produced without the run-time debugging code is smaller and slightly faster.

FILES

y.output	
y.tab.c	
y.tab.h	Defines for token names
yacc.tmp,	
yacc.debug, yacc.acts	Temporary files
/usr/lib/yaccpar	Parser prototype for C programs

MESSAGE

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.

LIMITATIONS

Because file names are fixed, at most one *yacc* process can be active in a given directory at a time.

SEE ALSO

lex(1)

READER COMMENT FORM

UNICOS User Commands Reference Manual SR-2011

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME______________________________

JOB TITLE ___________________________

FIRM _______________________________

ADDRESS ___________________________

CITY____________ STATE______ ZIP______

DATE______________________________

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY
RESEARCH, INC.

Attention: PUBLICATIONS
1345 Northland Drive
Mendota Heights, MN 55120

FOLD

STAPLE

READER COMMENT FORM

UNICOS User Commands Reference Manual SR-2011

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME____

JOB TITLE ______________________________________

FIRM ____

ADDRESS _______________________________________

CITY____________________ STATE________ ZIP________

DATE____

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY
RESEARCH, INC.

Attention: PUBLICATIONS
1345 Northland Drive
Mendota Heights, MN 55120

FOLD

STAPLE