NAME

`intro` – Introduces user utilities and commands

IMPLEMENTATION

All Cray Research systems

STANDARDS

The manual entry for each utility includes a section that lists the standard or standards in which the utility being described is defined. Do not infer, however, from the reference to a standard that the entire UNICOS set of utilities has necessarily been validated to conform to that standard. Validation of conformance to these standards is an issue discussed in other Cray Research documents.

In this manual, the reference to a standard provides you with information about the portability of code using that utility. For example, if the entry for the utility states that the utility is defined in the XPG4 standard, you can expect a given utility to be found in any vendor's system that conforms to the XPG4 standard. On the other hand, if the entry for the utility states that it is a Cray Research extension, you cannot expect it to be found in other vendors' systems.

The specific meanings of the terms in the STANDARDS section are as follows:

Term	Description
XPG4	Defined in X/Open Company Ltd., Single UNIX Specification Spec 1170.
POSIX	Defined in POSIX IEEE Std 1003.2-1992, *Shell and Utilities*.
AT&T extension	Not defined in the POSIX standard; it originated from one or more of the software releases from AT&T.
BSD extension	Not defined in the POSIX standard; it originated from the Fourth Berkeley Software Distribution under license from The Regents of the University of California.
FSF extension	Not defined in the POSIX standard; it originated from the Free Software Foundation (FSF) under terms of the GNU General Public License.
CRI extension	Not defined in the POSIX standard; added by Cray Research.

DESCRIPTION

This manual presents, in alphabetical order, the UNICOS user utilities and commands for all Cray Research systems.

Command Syntax Terminology

The following terms identify the components of a UNICOS utility:

Utility Component	Definition
Command	The name of an executable file in lowercase letters.

Option	A command-line element indicated by a hyphen and usually followed by a single lowercase letter.
Option-argument	A character string supplying information for the preceding option. Although there are some exceptions, option-arguments are usually not optional.
Operand	A command-line element to be passed to the utility; not associated with an option. Operands can be optional.

Items enclosed in square brackets, [ ], are optional. *White space* refers to any number of horizontal spaces or tabs.

For a more detailed description of conventions, see *UNICOS Command Conventions*, Cray Research publication CP–2058.

EXIT STATUS

On termination, each command returns 2 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 procedure (see `wait`(2) and `exit`(2)). The former byte is 0, indicating normal termination; the latter is usually 0, indicating successful execution, and nonzero indicating 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 status, and is described only where special conventions are involved.

BUGS

Many commands do not use the aforementioned syntax.

SEE ALSO

`getopt`(1), `getopts`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`getopt`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

UNICOS Command Conventions, Cray Research publication CP–2058

NAME

acctcom – Searches and prints process accounting files

SYNOPSIS

acctcom [-a] [-b] [-c] [-d] [-f] [-h] [-i] [-k] [-m] [-p] [-q] [-r] [-t] [-v] [-w] [-y] [-A] [-B] [-D] [-F] [-J] [-M] [-N] [-P] [-T] [-U] [-V] [-W] [-X] [-Y] [-Z] [-e *time*] [-g *group*] [-j *jid*] [-l *line*] [-n *pattern*] [-o *ofile*] [-s *time*] [-u *user*] [-C *sec*] [-E *time*] [-H *factor*] [-I *chars*] [-O *sec*] [-S *time*] [*files*]

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The acctcom utility reads data *files*, in the format described by acct(5), and writes selected records to standard output. You can specify *files* to be read; otherwise, the standard input or /usr/adm/acct/day/pacct is read. Each record represents the execution of one process.

The acctcom utility accepts three types of options: output file options, selection options, and printing options.

Output Options

-o *ofile* Copies selected process records in the input data format to *ofile*, an output file you specify. Suppresses standard output printing.

Selection Options

-e *time* Selects processes existing at or before *time*, given in the format [D*dd*:]*hh*[:*mm*[:*ss*]]. (See EXAMPLES.) The letter D flags the presence of the relative day offset parameter, which allows acctcom to select records from a previous day. (To determine the day offset, use the -W option.)

-g *group* Selects only processes belonging to *group*. You can designate the *group* by either the group ID or group name.

-j *jid* Selects only processes that have a job ID that matches the *jid* argument.

-l *line* Selects only processes that belong to terminal /dev/*line*.

-n *pattern* Selects only commands matching *pattern* that may be a regular expression, as in ed(1), except that a + symbol indicates one or more occurrences.

-s *time* Selects processes existing at or after *time*, given in the format [D*dd*:]*hh*[:*mm*[:*ss*]]. See the -e option and EXAMPLES for more information on this format. Using the same *time* for both -s and -e shows the processes that existed at *time*.

`-u` *user* Selects only processes that belong to *user*. May be specified by a user ID, a login name that is then converted to a user ID, a # symbol designating only those processes executed with super-user privileges, or a question mark (`?`) designating only those processes associated with an unknown user ID. To avoid interpretation by the shell, the question mark must be escaped.

`-C` *sec* Selects only processes with total CPU time (system plus user time) exceeding *sec* seconds.

`-E` *time* Selects processes ending at or before *time*, given in the format [`D`*dd*`:`]*hh*[`:`*mm*[`:`*ss*]]. See the `-e` option and EXAMPLES for more information on this format.

`-H` *factor* Selects only processes that exceed *factor*. Factor is the "hog factor" (as explained in the description of printing option `-h`).

`-I` *chars* Selects only processes that transfer more characters than the cutoff number given by *chars*.

`-O` *sec* Selects only processes with CPU system time exceeding *sec* seconds.

`-S` *time* Selects processes that start at or after *time*, given in the format [`D`*dd*`:`]*hh*[`:`*mm*[`:`*ss*]]. See the `-e` option for more information on this format.

Printing Options

`-a` Shows some average statistics about the processes selected. The statistics are printed after the output records.

`-b` Reads backward, showing latest commands first. This option has no effect when the standard input is read.

`-c` Prints additional I/O counts for buffered and raw blocks transferred.

`-d` Prints the number of clock periods past the last second mark to the actual start time.

`-f` Prints the `fork/exec` flag in octal and system exit status in decimal in the output. The lower 8 bits from the exit status of the process are reported in the exit status (`STAT`) column. For more information about these bits, see `wait`(2).

`-h` Prints the fraction of total available CPU time, instead of mean memory size, consumed by the process during its execution. This is known as the "hog factor" and is computed as follows:

(total CPU time)/(elapsed time)

`-i` Prints columns that contain the I/O counts in the output.

`-k` Prints total kcore-minutes instead of memory size. This is an integral of memory usage over time. One kcore-minute is 1024 words used for 1 minute.

`-m` Prints mean core size. This is the default print option. If you do not specify any other print options, `-m` is selected. If do specify other print options and you want mean core size to print, you must specify `-m`.

`-p` Prints process ID and parent process ID.

`-q` Prints only the average statistics as with the `-a` option. Does not print any output records.

`-r` Prints the CPU factor (user time/(system time + user time)).

`-t` Prints separate system and user CPU times.

`-v` Excludes column headings from the output.

`-w` Prints wait times.

`-y` Prints total system call time.

`-A` Print a column with account IDs.

`-B` Prints breakdown of multitasking time. If the default Hardware Performance Monitor (HPM) group is changed to anything but 1, the `WAIT SEMA` field will always be 0. This is because wait semaphore time is accumulated by HPM counter group 1.

`-D` Prints only device accounting information. This option must not be used with other print options.

`-F` Prints secondary data segments (SDS) usage information.

`-J` Prints a column with job IDs.

`-M` Prints additional memory usage data.

`-N` Prints `nice` field value, or scheduling priority for use of CPU time. The range of values typically is 1 through 40, with 1 being super user and higher `nice` values being lower priorities.

`-P` Prints Cray MPP usage information where a CRAY T3D system is attached.

`-T` Prints only end-of-job termination data. This option must not be used with other print options.

`-U` Shows all user IDs in numeric format.

`-V` Combines some I/O fields; must be specified with the `-c`, `-w`, or `-D` option.

`-W` Prints the start and end dates and each date change found in the file. Ignores all other print and selection options. This option is useful if your data spans more than 1 day (that is, not 00:00 to 00:00) and if more than 1 day of data is present in the `pacct` file. The day number of the date change is printed and can be used with the time selects. (See Example 2.)

`-X` Prints the process start date. The date follows the last date printed on the line and will be in the format: *day month date year* (for example, `Sun Sep 16 1993`).

`-Y` Prints the process end date. The date follows the last data printed on the line and will be in the format: *day month date year* (for example, `Sun Sep 16 1993`).

`-Z` Skips (does not print) first seven fields (must be specified with one of the print options (`-cdfhikmprtwyABDFJMNT`)).

files Input file(s) you specify. These are one or more of the `/usr/adm/acct/day/pacct*` files. 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/acct/day/pacct` is read; otherwise, the standard input is read. Therefore, when executing `acctcom` using the Network Queuing System (NQS), you must specify *files*.

Any file arguments specified are read in their respective order. Each file is usually read forward; that is, in chronological order by process completion time. The `/usr/adm/acct/day/pacct` file is the current file to be examined.

The output fields are as follows:

Field Name	Option	Definition
`COMMAND NAME`	`! Z`	ASCII command name OR #command name if executed with super-user privileges
`USER`	`! Z`	ASCII user name
`TTYNAME`	`! Z`	ASCII tty name (`?` in this field indicates that a process is not associated with a known terminal)
`START TIME`	`! Z`	Start time of process in clock format (that is, 10:01:25)
`END TIME`	`! Z`	End time of process in clock format (that is, 10:10:15)
`REAL (SECS)`	`! Z`	Elapsed time of process in seconds
`CPU (SECS)`	`! t`	CPU time used by the process in seconds OR
`(SECS) SYS`	`-t`	System time used by the process in seconds
`(SECS) USER`	`-t`	User time used by the process in seconds
`CHARS TRNSFD`	`-i`	Number of characters transferred
`PHYS REQS`	`-i`	Number of physical IO requests
`CPU FACTOR`	`-r`	User time divided by the CPU time
`HOG FACTOR`	`-h`	CPU time divided by the elapsed time
`MEAN SIZE(K)`	`-m`	Average amount of memory used by the process in kilowords
`KCORE MIN`	`-k`	Amount of memory used by the process in kilowords * minutes
`F STAT`	`-f`	The record flags, `F` (the `fork/exec` flag: `1` for `fork` without `exec`), and exit status
`JOB ID`	`-J`	Job identifier
`ACCT ID`	`-A`	Account identifier
`GRP ID`	`-G`	Group identifier
`LOGIO REQS`	`-c`	Number of logical IO requests
`PHYS BLKS`	`-cV`	Number of physical blocks transferred OR
`PHYS MVD: BUF`	`-c`	Number of buffered physical blocks transferred

Field Name	Option	Definition
PHYS MVD: RAW	-c	Number of raw physical blocks transferred
PID	-p	Process identifier
PPID	-p	Parent process identifier
START FRACT	-d	Fractional second part of the start time (.*xx*)
IOWAIT COMB	-wV	Combined I/O wait time in seconds OR
IOWAIT LOCKED	-w	Locked in memory I/O wait time in seconds
IOWAIT UNLOCK	-w	Unlocked in memory I/O wait time in seconds
IOWAIT TERM	-w	Total I/O wait time in seconds
WAIT SWAP	-w	I/O wait time in seconds
SCTIME SECS	-y	System call time in seconds
SDS MWSECS	-F	SDS memory integral in megawords * seconds
SDS REQS	-F	Number of SDS logical requests
SDS CHARS	-F	Number of SDS characters transferred
MPP PE'S	-P	Number of Cray MPP process elements used
MPP BB'S	-P	Number of Cray MPP barrier bits used
MPP (SECS)	-P	Time the Cray MPP was used in seconds
WAIT SEMA	-B	Time spent waiting for semaphores in seconds
USERTM CPU	-B	User time each CPU executed in seconds
IOWAIT LCKMEM	-M	I/O wait memory divided by I/O wait time in kilowords
HIMEM	-M	Largest amount of memory in kilowords
SWAPS	-M	Number of I/O swaps
NICE	-N	Nice value of the process
PROCESS START DATE	-W	Date that the process started in `ctime` format
PROCESS END DATE	-W	Date that the process ended in `ctime` format

NOTES

The `-d` option no longer displays the system run queue time.

BUGS

The `acctcom` utility reports only 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.

EXAMPLES

Example 1: The following example generates a list of commands executed by user `samuel` by examining all current process accounting files. The output includes system and user CPU times. In this example, if the `pacct` files are not specified in the order shown, the commands may not be reported in ascending time order.

```
acctcom -u samuel -t /usr/adm/acct/day/pacct?* /usr/adm/acct/day/pacct
```

Example 2: The following example shows how, using the printing option `-W`, the day number of the date change is printed.

```
acctcom -W

Day  0:  Mon Apr  3 10:20:11 1995 - first record
Day  1:  Tue Apr  4 00:00:00 1995 - date change
Day  4:  Fri Apr  7 10:20:00 1995 - date change
Day  4:  Fri Apr  7 14:43:10 1995 - last record
```

Example 3: The following example shows how you would select and print data from day 4, 10:20 A.M. in Example 2; you would use the same format to specify dates and times when using selection options `-e`, `-E`, `-s`, `-S`.

```
acctcom -s D4:10:20
```

Example 4: The following example shows how you would select and print data from the `pacct` file for an interval on day 4 between 8:00 A.M. and 4:00 P.M.

```
acctcom -S D4:08:00:00 -E D4:16:00:00
```

FILES

`/etc/group`	Group file that contains group names and group IDs
`/etc/udb`	User validation file that contains user control limits
`/usr/adm/acct/day/pacct`	Process accounting file that contains resource usage information for processes running on the system

SEE ALSO

`ps`(1), `su`(1)

`acct`(2), `wait`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`acct`(5), `utmp`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`acct`(8), `acctcms`(8), `acctcon`(8), `acctmerg`(8), `acctprc`(8), `acctsh`(8), `fwtmp`(8), `runacct`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

adb – Invokes the absolute debugger

SYNOPSIS

adb [-w] [*objfile* [*corefile*]]

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The adb utility invokes the absolute debugger, which lets you look at core files resulting from crashed systems or aborted programs. adb is an interactive, general-purpose debugger with access to global symbols. It lets you display output in a variety of formats, patch files, and run programs with embedded breakpoints.

The adb utility accepts the following options and operands:

-w Indicates that both *objfile* and *corefile* should be created if necessary and opened for reading and writing so that files can be modified using adb requests.

objfile 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 *objfile* is a.out.

corefile A core image file produced after executing *objfile*; the default for *corefile* is core. See core(5) for a description of core files.

Generally, requests to adb are in the following format:

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

address and *count* are expressions (see the Expressions subsection). adb maintains a current address called *dot*. If *address* is present, *dot* is set to *address*; otherwise, *dot* refers to the address of the last item printed. For example, the following request sets *dot* to octal 100 and prints the instruction at that address:

```
0100?i
```

The following request prints 15 octal numbers starting at address *dot*:

```
.,15/o
```

Initially *dot* is set to 0. 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 subsection.

For most requests, *count* specifies how many times the request is executed; default *count* is 1.

The `adb` utility ignores `QUIT`, and `INTERRUPT` causes `adb` to return to the next `adb` request. Use the `$Q`, `$q`, or `<CONTROL-d>` keys to exit from `adb`.

Expressions

Addresses are represented by expressions. Expressions consist of decimal, octal, and hexadecimal integers, and symbols from the program being tested. Integers are assumed to be octal by default. Expressions consist of the following characters:

`.` Value of *dot*.

`+` Value of *dot* incremented by the current increment.

`^` Value of *dot* decremented by the current increment.

`"` Last *address* typed.

integer An octal number. If you have used a `$d` request, *integer* is a decimal number unless the number begins with a leading `0x` (for hexadecimal) or a leading `0` (for octal).

integer`.`*fraction*
A 64-bit floating-point octal number.

'cccccccc' The ASCII value of up to 8 characters. A backslash (\) may be used to escape a `'` symbol.

`<` *name* 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 uppercase letters or digits. If *name* is a register name, the register value is obtained from the user structure in *corefile*. The register names are `a0` through `a7`, `s0` through `s7`, `p`, `vl`, `vm`, `v000` through `v777`, `b00` through `b77`, `t00` through `t77`, `sb0` through `sb7`, `sm00` through `sm37`, and `st0` through `st7`.

symbol Sequence of uppercase or lowercase letters, underscores or digits, not starting with a digit. A backslash (\) may be used to escape other characters. The value of *symbol* is taken from the symbol table in *objfile*.

(*exp*) Value of the expression *exp*.

You can also use monadic and dyadic operators. Dyadic operators are left-associative and are less binding than monadic operators.

`*`*exp* Contents of the location addressed by *exp* in *corefile*

`@`*exp* Contents of the location addressed by *exp* in *objfile*

`-`*exp* Integer negation

~ exp Bitwise complement

exp1`+`*exp2* Integer addition

exp1`-`*exp2* Integer subtraction

exp1`*`*exp2*	Integer multiplication
exp1`%`*exp2*	Integer division
exp1`&`*exp2*	Bitwise conjunction (and)
exp1`|`*exp2*	Bitwise disjunction (or)
exp1`#`*exp2*	*exp1* rounded up to the next multiple of *exp2*

Commands

Most commands to `adb` consist of a verb, followed by a modifier or list of modifiers. The following verbs are available. (The verbs `?` and `/` may be followed by `*`; see the Addresses subsection for further details.)

Verb	Description
`?`	Prints the contents from *objfile*
`/`	Prints the contents from *corefile*
`=`	Prints the current value of *dot*
`$`	Designates a miscellaneous request
`:`	Manages a child process
`!`	Creates a shell to read the rest of the line following `!`
`>`	Designates an assignment request

The following list shows the combinations of requests and modifiers you can issue to `adb`. Use `?` with the [`?`/]`l`, [`?/`]`w`, and [`?/`]`m` requests if you want to include the contents from *objfile*; use `/` if you want to include the contents from *corefile*.

`newline` Repeats the previous request with a *count* of 1 (works only with the ? and / requests).

[`?/`]`l` *value mask*
: Masks words starting at *dot* with *mask* and compares them with *value* until *count* matches are 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 request is `W`, it writes 8 bytes. Word-aligned addresses are required when writing to a subprocess address space.

[`?/`]`m` *b1 exp1 f1*[`?/`]
: Records new values for (*b1*, *exp1*, and *f1*). 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 `/`, the file (*objfile* or *corefile*, respectively) is used for subsequent requests. (For example, `/m?` causes `/` to refer to *objfile*.)

`>name` Assigns *dot* to the variable or register specified.

`$`*modifier* Specifies miscellaneous requests. The available *modifier*s are as follows:

`<`*file* Reads commands from the file *file* and returns.

`>`*file* Sends output to the file *file*, which is created if it does not exist. If no file name is given, returns to `stdout`.

`r` Prints the general registers, prints the instruction addressed by pc, and sets *dot* to pc. `b00` is also displayed.

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

`c` c stack backtrace; attempts a backtrace for all languages. If *address* is given, it is taken as the address of the current frame. If *count* is given, only the first *count* frames are printed.

`w` Sets the page width for output to *address* (default 80).

`s` Sets the limit for symbol matches to *address* (default 4096 bytes).

`o` Specifies that all integers input are regarded as octal. This is the default.

`d` Switches input integers as specified. To switch from

octal to decimal, enter:	`012$d`
octal to hexadecimal, enter:	`020$d`
hexadecimal to decimal, enter:	`0xa$d`
decimal to hexadecimal, enter:	`16$d`

`p` Changes the register set address. 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, $0 \leq n < var[VARQ]$.

`q` Exits from `adb`.

`u` Prints all nonzero variables in octal.

`m` Prints the address map.

`v` Prints vector registers at *address* for *count*.

`j` Prints `b` registers at *address* for *count*.

`k` Prints `t` registers at *address* for *count*.

`:`*modifier* Manages a subprocess. Available *modifier*s 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 request *c* is executed.

`d` Deletes breakpoint at *address*.

`r` Runs *objfile* 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 request. An argument starting with `<` or `>` causes the standard input or output to be established for the request. All signals are enabled on entry to the subprocess.

`c`*s* Continues the subprocess 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* Continues the subprocess the same as for `c`, except that the subprocess is single stepped *count* times. If there is no current subprocess, *objfile* 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` Terminates the current subprocess, if any.

Formats

A *format* consists of one or more characters that describe the format of the output. Each format character may be preceded by an integer and an asterisk, for example, indicating a repeat count for the format character (3*x). While parsing 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 Prints in signed octal.
`Q` 8 Prints long signed octal.
`d` 8 Prints 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. Numeric formats:
- 1 Same as `b`
- 2 Prints 2-byte parcels in octal
- 4 Prints 4-byte halfwords in octal

`c` 1 Prints the addressed character.
`C` 1 Prints the addressed character using the following escape convention. Character values 000 through 040 are printed as @ followed by the corresponding character in the range 0100 through 0140. The character @ is printed as @@.
`s` *n* Prints the addressed characters until a 0 character is reached.
`S` *n* Prints a string by 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 length 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`.
`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* decrements by 1 word; nothing is printed.

+ *dot* is incremented by 1; nothing is printed.
- *dot* is decremented by 1; nothing is printed.
space Increments *dot* by one word.

Variables

The `adb` utility provides several uppercase variables. Named variables are set initially by `adb`, but they are not subsequently used. You can access locations by using the `adb`-defined variables. The `adb` program reads the header of the core image file to find the values for these variables. If the second file specified does not seem to be a core file, or if it is missing, the header of the executable file is used instead.

`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

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`

Addresses

All addresses are byte addresses, except for the register number in a `$v` display. The interpretation of an address depends on its context. If a subprocess is being debugged, addresses are interpreted in the usual way (as described below) in the address space of the subprocess.

For the UNICOS operating system, 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.

EXIT STATUS

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

EXAMPLES

The following example shows a run of an adb debugger on a simple C program, which examines the symbol table at the end of an executable file. User input is shown in bold.

```
$ cat junk.c
# include <stdio.h>
main(){ printf("hello world\n");}
$ ./a.out/
hello world
$ adb  - a.out
$m
? map    ‘-’
b1 = 0   e1       = 40000000000000000f1 =  0
b2 = 0   e        = 0                     f2 = 0
/ map    ‘a.out’
b1 = 0   e1       = 326220                f1 = 70000
b2 = 0   e2       = 326220                f2 = 70000
/m 0 050000000000 0
0,4/00na
0:     0411    021547
20:    014624  016253
40:    03217   0
60:    0       01
100:
0100+0215470+0146240,30/O^tt010*Cna
364030:    0270000000040000003217   .@‘@‘@a@‘@‘@f@o
364040:   0141000001400000000001   B@‘@‘@@@‘@‘@‘@a
364050:                   03334200   @‘@‘@‘@‘@‘@m8@‘
364060:    0221012525410124400000   $AUXAR@‘@‘
364070:   0142400001440000000001   E@‘@‘H@‘@‘@‘@a
364100:                   03641500   @‘@‘@‘@‘@‘@oC@@
364110:    0221062224751723400000   $FIOON@‘@‘
364120:   0142400002000000000001   E@‘@a@‘@‘@‘@‘@a
364130:                   03641600   @‘@‘@‘@‘@‘@oC@‘
364140:    0221322404252221430503   $ZPERF1C
364150:   0141000002040000000001   B@‘@a@h@‘@‘@‘@a
364160:                   03334300   @‘@‘@‘@‘@‘@m8@@
364170:    0221423106656335060564   $bdmstat
364200:   0141000002040000000001   B@‘@a@h@‘@‘@‘@a
364210:                   03334400   @‘@‘@‘@‘@‘@m9@‘
364220:    0221463406456335060564   $fpistat
364230:   0141000002040000000001   B@‘@a@h@‘@‘@‘@a
364240:                   03334500   @‘@‘@‘@‘@‘@m9@@
364250:    0221573446456335060564   $oristat
```

```
364260:   0141000001640000000200   B@‘@‘h@‘@‘@‘@‘
364270:                   03334600   @‘@‘@‘@‘@‘@m9@‘
364300:    022164302711473l272000   $target@‘
364310:   0141000001640000000001   B@‘@‘h@‘@‘@‘@a
364320:                   03354600   @‘@‘@‘@‘@‘@mY@‘
364330:
$q
$ #
$ # run the program
$ ./a.out
hello world
$ # debug program, fixing invalid output
$ adb a.out -
write:b
:r
a.out: running
breakpoint       write:  s0  021564,0   $sysc$trap
(<a6+1)*010,2/OOna
414030:                          01                      044703
414050:                          014                     041364
414070:
041050/W 017
414050:                          14=                     17
0447030/XX
447030:                          68656c6c6f20776f        726c640a00000000
0447030/W 0x736f206c6f6e6720
447030:                          641453306615710073557=  715571006615733463440
0447040/W 0x7375636b65720a00
447040:                          7115431005000000000000=  715653066554534405000
:c
a.out: running
so long sucker
delbp: No such process
process                          terminated
$q
```

FILES

a.out	Executable file
core	Program memory dump

SEE ALSO

`ptrace`(2), `signal`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`ctime`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`a.out`(5), `core`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

addbss – Increases the amount of BSS space in an executable file

SYNOPSIS

addbss *file* [[*newfile*] *incr*]

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The addbss utility reports or changes the amount of BSS space for a program. The additional BSS space starts as free space in the memory manager.

The addbss utility accepts the following operands:

file Specifies an executable file.

newfile Specifies the new file, which contains *incr* more BSS space than *file*.

incr Specifies the amount of Kwords (1Kword is 1024 decimal) added to the BSS space of *file*.

If only one argument is present (the *file* argument), the amount of BSS space for that executable file is printed. *file* should represent an executable file. The format of an executable file is described in a.out(5). If two arguments are present (*file* and *incr*), *incr* Kwords are added to *file*'s BSS space. If three arguments are present (*file*, *newfile*, and *incr*), *file* is copied to *newfile* with a BSS size that is *incr* Kwords larger.

The *incr* argument must be at least 10.

NOTES

If this utility is installed with a privilege assignment list (PAL), a user with one of the following active categories is allowed to perform the actions shown:

Active Category	Action
system, secadm	Allowed to increase BSS space for any executable file. In a privileged administrator shell environment, shell-redirected I/O is not subject to file protections.
sysadm	Allowed to increase BSS space for any executable file subject to security label restrictions. Shell-redirected I/O is subject to security label restrictions.

If the PRIV_SU configuration option is enabled, the super user is allowed to increase BSS space for any executable file. Shell-redirected I/O on behalf of the super user is not subject to file protections.

SEE ALSO

a.out(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

admin – Creates and administers SCCS files

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The admin utility is used to create new Source Code Control System (SCCS) files and change parameters of existing ones. Arguments to admin, which may appear in any order, consist of keyletter arguments (keyletter arguments begin with the - symbol) 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 non-SCCS files (last component of the path name does not begin with s.) and unreadable files are silently ignored. If a name of - is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

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

-n Indicates that a new SCCS file is to be created.

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

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

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

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

`c` *ceil* Specifies the highest release (that is, "ceiling") that may be retrieved by a `get`(1) command for editing. Must be a number greater than 0 but less than or equal to 9999. The default value for an unspecified `c` flag is 9999.

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

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

`i` [*str*] Causes the `No id keywords (ge6)` message issued by `get`(1) or `delta`(1) to be treated as a fatal error. In the absence of this flag, the message is only a warning. The message is issued if no SCCS identification keywords (see `get`(1)) are found in the text retrieved or stored in the SCCS file. If 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* Specifies a *list* of releases to that 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 the releases skipped when a delta is made in a new release (for example, in making delta 5.1 after delta 2.7, releases 3 and 4 are skipped). These null deltas serve as anchor points so that branch deltas may be created from them later. The absence of this flag causes skipped releases to be nonexistent in the SCCS file, preventing branch deltas from being created from them in the future.

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

`m` *mod* Specifies the 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* Specifies the *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* Deletes 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.

`-a` *login* Specifies a *login* name, or numerical UNIX system group ID, to be added to the list of users that may make deltas (changes) to the SCCS file. A group ID is equivalent to specifying all *login* names common to that group ID. Several `-a` keyletters may be used on a single `admin` command line. As many *login*s, or numerical group IDs, as desired may be on the list simultaneously. If the list of users is empty, then anyone may add deltas. If *login* or group ID is preceded by a `!` symbol, they are to be denied permission to make deltas.

`-e` *login* Specifies 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.

`-m` *mrlist* Inserts the list of modification requests (MR) numbers into the SCCS file as the reason for creating the initial delta in a manner identical to `delta`(1). The `v` flag must be set and the MR numbers are validated if the `v` flag has a value (the name of an MR number validation program). Diagnostics will occur if the `v` flag is not set or MR validation fails.

`-y`[*comment*] Inserts the *comment* text 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 following 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 (for example, a new SCCS file is being created).

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

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

`-z` Recomputes the SCCS file checksum and stores it in the first line of the SCCS file (see keyletter `-h`).

NOTE: Use of this keyletter on a truly corrupted file may prevent future detection of the corruption.

files Specifies the files to be created or changed.

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` utility 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, 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 checksum. Another `admin -h` is recommended to ensure the SCCS file is valid.

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

MESSAGES

Use help(1) for explanations.

EXAMPLES

The following example creates a new SCCS file named s.example.c. The file example.c is used for initial input and the SCCSID line is required for future deltas (see the EXAMPLES section of delta(1)).

```
$ cat example.c
static char SCCSID[] = "%Z%%M%      %I%   %G% %U%";
main()
{
        printf("Hello, world!\n");
}
$ admin -n -iexample.c -fi"`grep SCCSID example.c`" s.example.c
$
```

FILES

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

SEE ALSO

cdc(1), comb(1), delta(1), ed(1), get(1), help(1), prs(1), rmdel(1), sact(1), sccs(1), sccsdiff(1), unget(1), val(1), vc(1), what(1)

chown(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

sccsfile(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`airlogger` – Logs AIR messages in the system log

SYNOPSIS

`airlogger` [`-s` *msg_type*] [`-p` *product*] [`-P` *subproduct*] *message*

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `airlogger` utility logs automated incident reporting (AIR) messages in a system log file. The utility sends messages to the `airlog`(3C) library routine, which formats the messages and passes them to the `syslog`(3C) routine. `syslog` writes the messages in a UNICOS system log maintained by the `syslogd`(8) command.

The `airlogger` utility accepts the following options and operands:

`-s` *msg_type* Specifies the message type. The *msg_type* argument can be one of the following:

`start`	Indicates normal daemon initiation
`term`	Indicates normal daemon termination
`panic`	Indicates abnormal daemon termination
`crit`	Indicates that a disaster has occurred
`warn`	Contains (nonfatal) information that indicates possible future disaster
`atten`	Contains information that should be displayed for the operator
`info`	Contains useful information to be logged.
`pulse`	Contains a daemon heartbeat
`fork`	Indicates that the daemon created a child process
`user`	Contains user-entered text
`conf`	Contains configuration information

`-p` *product* Specifies the product to which the message pertains. The *product* can be one of the following:

`unicos`	UNICOS kernel
`nqs`	Network Queuing System (NQS)
`tcp`	TCP/IP
`tape`	Online tape subsystem
`dmf`	Data Migration Facility (DMF)
`nfs`	Network file system (NFS)
`acct`	System accounting
`disk`	CRI disks
`superl`	Open Systems Interconnection (OSI) product
`share`	Fair-share scheduler

`cron` `cron` daemon

`-P` *subproduct* Specifies information that further delineates the origin of the message. The *subproduct* argument is a comma-separated string. For example, if the *product* were `nqs`, a possible subproduct string could be `qfdaemon,readq,end`.

message Text to be logged.

The `airlog` library routine creates the format of the log entry by listing the arguments and adding an identifying key, the contents of which are defined based on the type argument. See the `airlog`(3C) man page for more information.

FILES

`/usr/logs/airlog` AIR system log file

SEE ALSO

`airlog`(3C), `syslog`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`syslogd`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

alias – Defines or displays aliases

SYNOPSIS

alias [-t] [-x] [*alias-name*[=*string*] ...]

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4
AT&T extensions (-t and -x options)

DESCRIPTION

The alias utility creates or redefines alias definitions or writes the values of existing alias definitions to standard output. An alias definition provides a string value that replaces a command name when it is encountered (see sh(1)).

An alias definition affects the current shell execution environment and the execution environments of the subshells of the current shell. The alias definition does not affect the parent process of the current shell or any utility environment that the shell invokes.

The alias utility supports the following options and operands:

-t	Sets or lists tracked aliases.
-x	Sets or lists exported aliases.
alias-name	Writes the alias definition to standard output.
alias-name=*string*	Assigns the value of *string* to the alias *alias-name*.

If no operands are given, all alias definitions are written to standard output.

The format for displaying aliases is as follows:

"<*name*>=<*value*>\n"

The *value* string is written with appropriate quoting so that it is suitable for reinput to the shell.

NOTES

The alias utility is a built-in utility to the standard shell (sh(1)). An executable version of this utility is available in /usr/bin/alias.

The csh(1) utility has a built-in alias utility that has slightly different characteristics. See csh(1).

EXIT STATUS

The alias utility exits with one of the following values:

0 Successful completion.

>0 One of the *alias-name* operands specified did not have an alias definition or an error occurred.

SEE ALSO

csh(1), sh(1), unalias(1)

NAME

amlaw – Displays maximum theoretical parallel processing speedups

SYNOPSIS

amlaw [*ncpu* [*pc-values*]]

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The amlaw utility displays the maximum theoretical speedup for selected pairs (number of CPUs, percent parallelism) using Amdahl's Law.

The amlaw utility accepts the following arguments:

ncpu *ncpu* is the number of CPUs; it must be greater than 1.

pc-values *pc-values* is the percent parallelism; each value must be greater than 0 and less than or equal to 99.9999999. (*pc-values* that are less than 1 are multiplied by 100.0.) Up to 19 *pc-values* may be used.

A table of sample speedups is generated, using selected *ncpu* and *pc-values*, when any of the following conditions occur:

- No arguments are specified.
- Only *ncpu* is specified.
- *ncpu* is 0.
- The only *pc-value* specified is 0.
- More than one *pc-value* is specified.

EXAMPLES

The following command line generates the speedup information shown below:

```
$ amlaw  8  94.45
8 CPUs, 94.45% Parallelism:  Max Theoretical speedup is  5.76
```

The following command line generates the speedup table shown below:

```
$ amlaw  16  97  97.1  97.2  97.3  97.4  97.5
        Max Theoretical Speedup for 16 CPUs

          %   Speedup   |      %   Speedup
        ----  -------   |     ---  -------
        97.0   11.034   |    97.3  11.388
        97.1   11.150   |    97.4  11.511
        97.2   11.268   |    97.5  11.636
```

SEE ALSO

Optimizing Code on Cray PVP Systems, Cray Research publication SG–2192

NAME

apropos – Locates commands by keyword

SYNOPSIS

apropos *keyword* ...

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The apropos utility displays the man page name, section number, and a short description for each man page that contains the *keyword* in its NAME line. This information is contained in the /usr/man/whatis file. Each keyword is considered separately and the case of letters is ignored. Words that contain *keyword* also are displayed; for example, if *keyword* is compile, apropos displays all instances of compiler also.

The apropos utility accepts the following operand:

keyword The character string that is searched for in the NAME line of available man pages.

The function of the apropos utility is identical to that of the -k option of the man(1) utility.

If the line displayed by apropos starts *filename*(*section*)..., you can type man *section filename* to display the man page for *filename*.

FILES

/usr/man/whatis Database

SEE ALSO

man(1), whatis(1)

NAME

ar – Archive and library maintainer for portable archives

SYNOPSIS

```
ar -d [-l] [-s] [-v] archive files
ar -m [-a] [-b] [-i] [-l] [-s] [-v] [posname] archive files
ar -p [-s] [-v] archive [files]
ar -q [-c] [-l] [-s] [-u] [-v] archive files
ar -r [-a] [-b] [-c] [-i] [-l] [-s] [-u] [-v] [posname] archive files
ar -t [-s] [-v] [-z] archive [files]
ar -x [-o] [-s] [-v] [-z] [-C] [-T] archive [files]
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4
AT&T extensions (-a, -b, -i, -l, -m, -o, -q, and -s options)

DESCRIPTION

The ar utility maintains groups of files combined into a single archive file. Once an archive has been created, new files can be added, and existing files can be extracted, deleted, or replaced. The 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.

When ar creates an archive, it creates headers in a format that is portable across all Cray Research systems. The portable archive format and structure are described in detail in the ar(5) man page.

The following options are supported:

-a Places new modules after *posname*. This option only works when the -r or -m options are specified.

-b Places new modules before *posname*. This option only works when the -r or -m options are specified.

-c Suppresses the diagnostic message that is written to standard error by default when the archive file *archive* is created.

-d Deletes *files* from *archive*.

-i Equivalent to -b.

-l Places temporary files in the local (current working) directory rather than in the default temporary directory, TMPDIR.

`-m` Moves the named *files* to the end of the archive. If `-a`, `-b`, or `-i` are specified, the *posname* argument must be present and, as in `-r`, specify where the files are to be moved.

`-o` Sets the "last modified" date to the date recorded in the archive. This option only works if the `-x` option also is specified.

`-p` Writes the contents of the files from *archive* to the standard output. If no *files* are specified, the contents of all files in the archive are written in the order of the archive.

`-q` Quickly appends the named *files* to the end of the archive file. The positioning options `-a`, `-b`, and `-i` are not valid. No checks are performed as to whether or not the added members are already in the archive. This option is useful to avoid quadratic behavior when creating a large archive piece-by-piece.

`-r` Replaces or adds *files* to *archive*. If the archive named by *archive* does not exist, a new archive file is created. Files that replace existing files do not change the order of the archive.

If the `-a`, `-b`, or `-i` options are not specified, *files* that do not replace existing files are appended to the archive. Otherwise, the *posname* operand must be present and *files* are positioned according to the option and *posname* specified.

`-s` Forces regeneration of the archive symbol table, even if `ar` is not invoked with a command that will modify the archive contents (see `ranlib`(1)).

`-t` Writes a table of contents of *archive* to the standard output. The files specified by the *file* operands are included in the written list. If no *file* operands are specified, all files in *archive* are included in the order of the archive.

`-u` Updates older files. When used with the `-r` or `-m` options, files within the archive will be replaced only if the corresponding *file* has a modification time that is at least as new as the modification time of the file within the archive.

`-v` Gives verbose output. When used with the option characters `-d`, `-m`, `-r`, or `-x`, writes a detailed file-by-file description of the archive creation and maintenance activity.

When used with `-p`, writes the name of the file to the standard output before writing the file itself to the standard output.

When used with `-t`, includes a long listing of information about the files within the archive.

`-x` Extracts the files named by the *files* operands from *archive*. The contents of the archive file are not changed. If no *files* operands are given, all files in the archive are extracted. If the file name of a file extracted from the archive is longer than that supported in the directory to which it is being extracted, the results are undefined. The modification time of each file extracted is set to the time the file is extracted from the archive.

`-z` Provides backward-compatibility for those archives incorrectly constructed in UNICOS 7.0 and UNICOS 8.0. This option is used when the message `malformed archive` is generated. If this message is produced using the `-z` option, invoke `ar` again without the option. If this message still appears, the archive is truly malformed. This option can only be used with the `-t` and `-x` options.

`-C` Prevents extracted files from replacing existing files of the same name.

`-T` Allows file name truncation of extracted files whose archive names are longer than the file system can support. By default, extracting a file with a name that is too long is an error; a diagnostic message is written and the file is not extracted.

The following operands are supported:

posname An archive member name used as a reference point in positioning other files in the archive.

archive The path name of the archive file.

files Path names. Only the last component is used when comparing against the names of files in the archive. If two or more *files* operands have the same last path name component (base name), both are added to the archive. In the case of such files, however, each file operand matches only the first archive file having a name that is the same as the last component of the file operand. The archive format used by `ar` does not truncate valid file names of files added to, or replaced in, the archive.

NOTES

All *files* operands can be path names. However, files within archives are identified by a file name, which is the last component of the path name used when the file was entered into the archive. The comparison of *files* operands to the names of files in the archives is performed by comparing the last component of the operand to the name of the archive file.

ENVIRONMENT VARIABLES

Variable	Description
`RANLIB`	Absolute path name of the utility that optimizes the archive file. The default is `opt/ctl/bin/ranlib`. If set to a `NULL` string, optimization of the archive is not performed.
`RANLIBFLAGS`	Option arguments to the executable file specified by `RANLIB`.

EXIT STATUS

The `ar` utility exits with one of the following values:

0 Successful completion.

>0 An error occurred.

MESSAGES

A phase error indicates that one or more files of the archive are no longer available, and that the archive itself can no longer be used. It is possible that some individual parts of the archive may be salvaged by extracting them.

Most other error messages are self-explanatory.

EXAMPLES

Example 1: The following example creates library `mylib.a` from object files `file1.o`, `file2.o`, and `file3.o`. After the creation of the library, the library's table of contents is printed along with verbose output:

```
$ ar -r mylib.a file1.o file2.o file3.o
$ ar -tv mylib.a
```

Example 2: The following example adds `newfile.o` after `file1.o` in the archive `mylib.a` with verbose output:

```
$ ar -rav file1.o mylib.a newfile.o
```

Example 3: The following example extracts all files from the archive `mylib.a`:

```
$ ar -x mylib.a
```

FILES

`TMPDIR`	Directory containing temporary files for user

SEE ALSO

`bld`(1), `segldr`(1),

`stat`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`tmpnam`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`a.out`(5), `ar`(5), `relo`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`ranlib`(1) Online only

NAME

asa – Interprets ASA carriage control characters

SYNOPSIS

asa [*files*]

IMPLEMENTATION

All Cray Research systems
SPARC systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The asa utility interprets the output of any programs that use ASA carriage control characters. It processes either the *files* that 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:

<space>	The rest of the line is output without change.
0	A <newline> is output, followed by the rest of the input line. This results in double spacing.
-	Two <newline> symbols are output, followed by the rest of the input line. This results in triple spacing.
1	Advance to the top of the next page, then output the rest of the input line.
+	Return to the column position 1; then overwrite the previous line with the rest of the input line.

Lines beginning with other than the preceding characters are treated as if they began with a <space>. 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.

NOTES

If this utility is installed with a privilege assignment list (PAL), a user with one of the following active categories is allowed to perform the actions shown:

Active Category	Action
system, secadm	Allowed to interpret any input file. In a privileged administrator shell environment, shell-redirected I/O is not subject to file protections.
sysadm	Allowed to interpret any input file subject to security label restrictions. Shell-redirected I/O is subject to security label restrictions.

If the PRIV_SU configuration option is enabled, the super user is allowed to interpret any input file. Shell-redirected I/O on behalf of the super user is not subject to file protections.

EXIT STATUS

The asa utility exits with one of the following values:

0 All input files were output successfully.

>0 An error occurred.

EXAMPLES

Example 1: The following example uses asa as a filter to view the output of a Fortran program that uses ASA carriage control characters. myfort is the Fortran executable file, datafile is the input data to myfort, and textfile is the output file:

```
$ myfort < datafile | asa > textfile
```

Example 2: asa can be used with an executable file as follows:

```
$ a.out | asa > file1
```

The output, properly formatted and paginated, would be directed to a file file1. Fortran output sent to the file could be viewed by entering the following:

```
$ asa file
```

SEE ALSO

fsplit(1), lp(1), nasa(1)

NAME

ascheck – Validates the array services configuration

SYNOPSIS

ascheck [-F] [-Kl *key*] [-Kr *key*] [-p *port*] [-q ...] [-s *server* [-D]] [-t *value*]

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The ascheck command validates the configuration of all the array services daemons known to the local array services daemon. ascheck checks to ensure that the various array and machine definitions are consistent, both on the individual servers and with corresponding definitions on the other servers. If any problems are found, they are described in detail, and suggestions are provided to correct the situation.

It should be noted that ascheck is not a syntax checker for the individual array services configuration files; that function is handled by the -c option of arrayd(8). Instead, ascheck is used to check the semantics of a syntactically correct array services configuration file.

The ascheck command relies on information provided by array services. Thus, for complete coverage, an array services daemon should be running on every machine that is mentioned in the local configuration file(s).

The checks that ascheck makes include the following:

- Ensuring that the local array services daemon is accessible.
- Verifying that all of the array services daemons known to the local daemon are also available.
- Ensuring that each array services daemon is using the correct version of the array services library.
- Ensuring that the LOCAL IDENT value used by a particular array services daemon is not different from the machine ID used by the operating system on that daemon's machine.
- Suggesting that a machine ID be set in the operating system on those systems that have an array services daemon but have not yet set a machine ID.
- Ensuring that no two arrays on a particular server have the same ARRAY IDENT value.
- Ensuring that the SERVER IDENT value, which may be declared for a particular ARRAY/MACHINE definition, matches the LOCAL IDENT value of the corresponding server.
- Warning if the list of machines defined for a particular array does not match the list of machines for an array with the same name on a different server.
- Warning about any server that has two or more arrays defined but has not specified a default array.

The ascheck command takes several options, primarily to control the selection of the local server and to select the desired amount of output.

-F	When used with -s, indicates that array services requests should be forwarded to the specified server via the server on the current machine rather than sent directly.
-Kl *key*	Use *key* for the local authentication key when communicating directly with a remote array services daemon.
-Kr *key*	Use *key* for the remote authentication key when communicating directly with a remote array services daemon.
-p *port*	Specifies the port address of the local arrayd server. Defaults to the value of the ARRAYD_PORT environment variable if present, or 5434 otherwise.
-q	Produces less verbose (quieter) output. Repeated occurrences (either -q -q . . . or -qq . . .) may further decrease the amount of output.
-s *server*	Specifies the host name or IP address of the local arrayd server. Defaults to the value of the ARRAYD environment variable if present, or localhost otherwise.
-D	When used with -s, indicates that array services requests should be sent directly to the specified server, rather than being forwarded to that server by the array services daemon running on the current machine. This is the default behavior.
-t *value*	Specifies the time-out value (in seconds) used for waiting on individual responses from the array daemon.

NOTES

The array services daemon (arrayd(8)) must be running on all machines that are to be examined. It does not necessarily have to be running on the machine that executes ascheck if an alternate server was specified in some way.

SEE ALSO

arrayd(8)

arrayd.conf(5) Online only

NAME

at, batch – Executes commands at a later time

SYNOPSIS

```
at [-m] [-f file] [-q queuename] -t time
at [-m] [-f file] [-q queuename] timespec ...
at -r at_job_id ...
at -l -q queuename
at -l [at_job_id ...]
batch
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The at and batch utilities read commands from standard input to be executed at a later time. The at utility lets you specify when the commands should be executed; jobs queued with batch execute when system load level permits.

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

Users whose names appear in the /usr/lib/cron/at.allow file are permitted to use at. When that file does not exist, the /usr/lib/cron/at.deny file is checked to determine whether a user should be denied access to at. If neither file exists, only root will be allowed to submit a job. Null file at.allow means that no users are allowed to use at; null file at.deny means that no users are denied the use of at. The allow/deny files consist of one user name per line.

The at utility accepts the following options:

-f *file* Specifies the path name of a file to be used as the source of the at job, rather than standard input.

-l Lists all jobs scheduled for the invoking user if no *at_job_id* operands are specified. If *at_job_id*s are specified, lists only information for these jobs.

-m Sends mail to the invoking user after the *at-job* has run, announcing its completion. Standard output and standard error produced by the *at-job* are mailed to the user as well, unless redirected elsewhere. Mail is sent even if the job produces no output. If you omit -m, the job's standard output and standard error are mailed to the user, unless redirected elsewhere.

`-q` *queuename*

Queues the command in queue *queuename*. The default queue is `a`. *queuename* can be one of 25 different queues (that is, `a`, `b`, `d–z`.). Queue `b` is defined as the batch queue; jobs in this queue run whenever the system administrator-defined maximum level is not exceeded. Jobs in all other queues run at the time specified on the command line. The character `c` is not allowed after the `-q` option. When used with the `-l` option, the search is limited to that particular queue.

`-r` Removes the jobs with the specified *at_job_id* operands that were previously scheduled by the `at` utility.

`-t` *time* Submits the job to be run at the specified *time*. The option-argument is of the form:

[[*CC*]*YY*]*MMDDhhmm*[*.SS*]

where each two digits represent the following:

CC First two digits of the year (the century).

YY Second two digits of the year.

MM Month of the year (01–12).

DD Day of the month (01–31).

hh Hour of the day (00–23).

mm Minute of the hour (00–59).

SS Second of the minute (00–61).

If you do not specify *CC* or *YY*, the current year is assumed. If you specify *YY* but not *CC*, *CC* will become 19, if *YY* is in the range 69–99. *CC* will become 20, if *YY* is in the range 00–68.

timespec Consists of a *time* followed by an optional *date* and an optional *increment*, or the special name, `now`.

You may specify the time as 1, 2, or 4 digits. A 1-digit or 2-digit number indicates hours. A 4-digit number indicates hours and minutes. Alternatively, you may alternatively specify time as two numbers separated by a colon, meaning *hour*`:`*minute*. An `am` or `pm` suffix may be appended; otherwise, 24-hour clock time is understood. The suffix `gmt`, `utc` or `zulu`, may be used to indicate GMT. The special names `noon` and `midnight` 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 3 characters). Two special "days," `today` and `tomorrow`, are recognized. If date is not specified, and if the given hour is greater than the current hour, `today` is assumed. `tomorrow` is assumed if the specified hour is less than the current hour. If the specified month is less than the current month (and no year is given), next year will be 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.) The keyword `next` is equivalent to an increment number of `+1`. For example, the following are equivalent commands:

```
at 2pm +1 week
at 2pm next week
```

Valid time specifiers include the following:

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

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

`batch` submits a batch job. It is equivalent to `at -q b -m now`.

`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` utilities. You can also get job numbers by typing `at -l`. Unless you are the super user, you can remove only your own jobs.

At the time of submission, `at` jobs are run at the user's current security label.

NOTES

If this utility is installed with a privilege assignment list (PAL), a user who is assigned the following privilege text upon execution of this command is allowed to perform the action shown:

Privilege Text	Action
`showall`	Allowed to manage all jobs.

If this utility is installed with a PAL, a user with one of the following active categories is allowed to perform the action shown:

Active Category	Action
`system`, `secadm`, `sysadm`, `sysops`	Allowed to manage all jobs.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to manage all jobs.

EXIT STATUS

The `at` utility exits with one of the following values:

0 The `at` utility successfully submitted, removed, or listed a job or jobs.

>0 An error occurred.

The batch utility exits with one of the following values:

0 The batch utility successfully submitted a job.

>0 An error occurred.

MESSAGES

The at utility reports various syntax errors and times out-of-range.

EXAMPLES

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

The following sequence can be used at a terminal:

```
$ batch
make filename >outfile
<CONTROL-d>
```

Example 2: The following sequence, which demonstrates the redirecting of 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
!
```

Example 3: To use the at utility to schedule the execution of a shell script or executable binary file (stored in *execfile*), use shell input redirection (<), as in the following example:

```
$ at 0815am Jan 24 < execfile
```

Example 4: To have a job reschedule itself, invoke at from within the shell procedure by including in the shell file code similar to the following:

```
$ 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

SEE ALSO

`kill`(1), `mail`(1), `nice`(1), `ps`(1), `sh`(1)

`chown`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`proto`(5), `queuedefs`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`cron`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

awk, nawk – Pattern scanning and processing language

SYNOPSIS

awk [-F *ERE*] [-v *assignment*]... *program* [*argument*]...

awk [-F *ERE*] -f *progfile*... [-v *assignment*]... [*argument*]...

nawk [-F *ERE*] [-v *assignment*]... *program* [*argument*]...

nawk [-F *ERE*] -f *progfile*... [-v *assignment*]... [*argument*]...

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The awk utility scans each input file for lines that match any of a set of patterns specified in *program*. Each pattern in *program* may have an associated action that will be performed when a line of a file matches the pattern. The set of patterns may appear either literally as *program* or in a file specified by using the -f option. This version of awk provides capabilities that were not available with previous versions.

awk accepts the following options and arguments:

-F *ERE* Defines the input field separator to be the Extended Regular Expression *ERE*. (Currently, only Basic Regular Expressions are supported.)

-v *assignment*... *assignment*s in the form *x=xvalue y=yvalue* can be passed to awk; *x* and *y* are awk built-in variables. The specified variable assignment occurs prior to executing the awk program, including the actions associated with BEGIN patterns, if any. You can specify multiple occurrences of the -v option.

-f *progfile* File that contains the set of pattern-action statements. If you specify multiple instances of this option, the concatenation of the files specified as *progfile* is used as the awk program.

program Set of patterns for which awk scans file. To protect the *program* string from the shell, enclose it in a single quotation marks.

argument Either of the following types of *argument*s can be specified:

assignment *assignment*s in the form *x=xvalue y=yvalue* can be passed to awk; *x* and *y* are awk built-in variables.

file Input files. If no files exist, the standard input is read. The file name `-` specifies the standard input. Each input 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 usually made up of fields separated by white space. (To changed the default, use the `FS` built-in variable or the `-F` *ERE* option.) The fields are denoted `$1`, `$2`, ...; `$0` refers to the entire line.

A pattern-action statement has the form:

pattern `{` *action* `}`

You can omit either pattern or action. If no action with a pattern exists, the matching line is printed. If no pattern with an action exists, the action is performed on every input line.

Patterns are arbitrary Boolean combinations (`!`, `||`, `&&`, and parentheses) of relational expressions and Extended Regular Expressions. A relational expression is one of the following:

"*expression relop expression*"
"*expression matchop regular_expression*"
expression `in` *array-name*
`(`*expression*`,`*expression*`, ... )` `in` *array-name*

A *relop* is any of the six relational operators in C, and a *matchop* is either `~` (contains) or `!~` (does not contain). An *expression* is an arithmetic expression, a relational expression, the special expression

var `in` array

or a Boolean combination of these.

You can use the special patterns `BEGIN` and `END` to capture control before the first input line has been read and after the last input line has been read, respectively. These keywords do not combine with any other patterns.

Regular expressions are as in `egrep` (see `egrep`(1)). In patterns, they must be surrounded by slashes. Isolated Extended 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 pattern.

You can use an Extended Regular Expression to separate fields by using the `-F` *ERE* option or by assigning the expression to the built-in variable `FS`. The default is to ignore leading `<blank>`s and to separate fields by `<blank>`s and/or `<tab>` characters. However, if `FS` is assigned a value, leading `<blank>`s are no longer ignored.

Other built-in variables include the following:

`ARGC`	Command-line argument count
`ARGV`	Command-line argument array
`FILENAME`	Name of the current input file

FNR	Ordinal number of the current record in the current file
FS	Input field separator Extended Regular Expression (default is <blank> and <tab> characters)
NF	Number of fields in the current record
NR	Ordinal number of the current record
OFMT	Output format for numbers (default is %.6g)
OFS	Output field separator (default is <blank> character)
ORS	Output record separator (default is <newline> character)
RS	Input record separator (default is <newline>)
SUBSEP	Separates multiple subscripts (default is 034)

An action is a sequence of statements. A statement can be one of the following:

```
if ( expression ) statement [ else statement ]
while ( expression ) statement
do statement while ( expression )
for ( expression ; expression ; expression ) statement
for ( var in array ) statement
delete array [subscript]
break
continue
{ [ statement ] ... }
expression          # commonly variable = expression
print [ expression-list ] [ >expression ]
printf format [ , expression-list ] [ >expression ]
next                # skip remaining patterns on this input line
exit [expr] # skip the rest of the input; exit status is expr
return [expr]
```

Statements are terminated by semicolons, <newline> characters, or right braces. An empty expression-list stands for the whole input line. Expressions take on string or numeric values as appropriate, and are built using the operators +, -, *, /, % and concatenation (indicated by a <blank>). The 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 or 0. Array subscripts can be any string, not necessarily numeric, allowing for associative memory. String constants are quoted (").

The print statement prints its arguments on the standard output, or on a file if >*expression* is present, or on a pipe if | *cmd* is present. The current output field separator separates arguments. The output record separator terminates arguments. The printf statement formats its expression list according to the format in printf(3C).

awk has a variety of built-in functions: arithmetic, string, input/output, and general.

The arithmetic functions are atan2, cos, exp, int, log, rand, sin, sqrt, and srand. int truncates its argument to an integer. rand returns a random number between 0 and 1. srand(expr) sets the seed value for rand to expr or uses the time of day if expr is omitted.

The string functions are as follows:

gsub(*ERE*,*repl*,[*in*])
: Behaves like the sub string function, except that it replaces successive occurrences of the Extended Regular Expression (such as the ed(1) global substitute command) in $0 or in the *in* argument, when specified.

index(*s*,*t*)
: Returns the position in string *s* where string *t* first occurs, or 0 if it does not occur at all.

int
: Truncates to an integer value.

length[([*s*])]
: Returns the length of its argument taken as a string, or of the whole line, $0, if no argument exists.

match(*s*,*ERE*)
: Returns the position in string *s* where the Extended Regular Expression occurs, or 0 if it does not occur at all. RSTART is set to the starting position (which is the same as the returned value), and RLENGTH is set to the length of the matched string.

rand
: Random number on (0, 1).

split(*s*,*a*,*fs*)
: Splits the string *s* into array elements *a*[1], *a*[2], . . ., *a*[*n*], and returns *n*. The separation is done with the Extended Regular Expression *fs* or with the field separator FS if *fs* is not given.

srand
: Sets the seed for rand

sprintf(*fmt*,*expr*,*expr*,...)
: Formats the expressions according to the printf(3C) format given by *fmt* and returns the resulting string.

sub(*ERE*,*repl*,*in*)
: Substitutes the string *repl* in place of the first instance of the Extended Regular Expression *ERE* in string *in* and returns the number of substitutions. If you omit *in*, awk substitutes in the current record ($0).

substr(*s*,*m*,*n*)
: Returns the *n* -character substring of *s* that begins at position *m*.

tolower(*s*)
: Return a string based on the string *s*. Each character in *s* that is an uppercase letter specified to have a tolower mapping by the LC_CTYPE category of the current locale is replaced in the returned string by the lowercase letter specified by its mapping. Other characters in *s* are unchanged in the returned string.

toupper(*s*)
: Return a string based on the string *s*. Each character in *s* that is a lowercase letter is specified to have a toupper mapping by the LC_CTYPE category of the current locale is replaced in the returned string by the uppercase letter specified by its mapping. Other characters in *s* are unchanged in the returned string.

The input/output and general functions are as follows:

`close(`*filename*`)`	Closes the file or pipe named *filename*.
cmd `| getline`	Pipes the output of *cmd* into `getline`; each successive call to *getline* returns the next line of output from *cmd*.
`getline`	Sets `$0` to the next input record from the current input file.
`getline <`*file*	Sets `$0` to the next record from *file*.
`getline` *x*	Sets variable *x* instead.
`getline` *x* `<`*file*	Sets *x* from the next record of *file*.
`system(`*cmd*`)`	Executes *cmd* and returns its exit status.

All forms of `getline` return 1 for successful input, 0 for end of file, and −1 for an error.

`awk` also provides user-defined functions. You can define such functions (in the pattern position of a pattern-action statement) as follows:

```
function name(args,...) { stmts }
func name(args,...) { stmts }
```

Function arguments are passed by value if scalar and by reference if array name. Argument names are local to the function; all other variable names are global. You can nest function calls, and functions may be recursive. You can use the `return` statement to return a value.

NOTES

If this utility is installed with a privilege assignment list (PAL), a user with one of the following active categories is allowed to perform the actions shown:

Active Category	Action
`system`, `secadm`	In a privileged administrator shell environment, shell-redirected I/O is not subject to file protections.
`sysadm`	Shell-redirected output is subject to security label restrictions.

If the `PRIV_SU` configuration option is enabled, shell-redirected I/O on behalf of the super user is not subject to file protections.

WARNINGS

Currently, only Basic Regular Expressions are supported. If fields are involved, input white space is not preserved on output.

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.

Separate pattern-action statements by either a semicolon or a `<newline>`. This is an incompatibility with the old version of `awk`.

EXIT STATUS

The awk utility exits with one of the following values:

0 All input files were processed successfully.

>0 An error occurred.

The exit status can be altered within the program by using an exit expression.

EXAMPLES

Example 1: The following awk program prints lines longer than 72 characters:

```
length > 72
```

Example 2: The following awk program prints first two fields in opposite order:

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

Example 3: The following awk program prints the first two fields in opposite order with input fields separated by comma and/or <blank>s and <tab>s:

```
BEGIN { FS = ",[ \t]*|[ \t]+" }
      { print $2, $1 }
```

Example 4: The following awk program adds up the first column, and then prints the sum and average:

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

Example 5: The following awk program prints fields in reverse order:

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

Example 6: The following awk program prints all lines between start/stop pairs:

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

Example 7: The following awk program prints all lines whose first field differs from the previous one:

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

Example 8: The following awk program simulates the echo(1) utility:

```
BEGIN {
      for (i = 1; i < ARGC; i++)
            printf "%s", ARGV[i]
      printf "\n"
      exit
      }
```

Example 9: The following awk program prints a file, filling in page numbers starting at 5:

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

Assuming this program is in a file named prog, the following command line prints the file input and numbers its pages starting at 5:

```
awk -f prog n=5 input
```

SEE ALSO

echo(1), ed(1), egrep(1), grep(1), lex(1), sed(1)

printf(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

The AWK Programming Language, Aho, Kerninghan, Weinberger, Addison-Wesley, 1988

sed & awk, Dale Doherty, O'Reilly & Associates, Inc., 1990

NAME

banner – Makes posters

SYNOPSIS

banner *strings*

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The banner utility prints the *strings* argument (each string can be up to 10 characters long) in large letters on standard output. You can include spaces in an argument by surrounding them with quotation marks. The maximum number of characters that can be accommodated in a line is implementation-dependent; excess characters are simply ignored.

EXAMPLES

Example 1: Outputs the words `hi there` as one string:

```
banner "hi there"
```

Example 2: Outputs `hi` as one string and `there` as a second string.

```
banner hi there
```

SEE ALSO

echo(1)

NAME

basename, dirname – Prints parts of path names on standard output

SYNOPSIS

basename *string* [*suffix*]

dirname *string*

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The basename utility deletes any prefix ending in / and the *suffix* (if present in *string*) from *string*, and prints the result on the standard output.

The dirname utility converts *string* to the name of the directory that contains the file name corresponding to the last pathname component in *string*. The result is printed on the standard output.

NOTES

The *suffix* must match identically to the suffix in *string*.

EXIT STATUS

The basename and dirname utilities exit with one of the following values:

0 Successful completion.

>0 An error occurred.

EXAMPLES

Example 1: The following shell script, invoked with the /usr/src/cmd/cat.c argument, compiles the specified file and moves the output to a file named cat in the current directory:

```
cc $1
mv a.out $(basename "$1" .c)
```

Example 2: The following example sets shell variable NAME to /usr/src/cmd:

```
NAME=$(dirname /usr/src/cmd/cat.c)
```

SEE ALSO

sh(1)

NAME

bc – An arbitrary precision calculator language

SYNOPSIS

bc [-l] [-w] [-s] [*file* ...]

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4
FSF XPG4 (-s and -w options)

DESCRIPTION

bc is a language that supports arbitrary precision numbers with interactive execution of statements. This bc utility replaces the bc utility that was included in UNICOS releases prior to UNICOS 8.3. The old bc utility has been renamed obc(1).

There are some similarities in the bc syntax to the C programming language. A standard math library is available by command-line option. If requested, the math library is defined before processing any files. bc starts by processing code from all the files listed on the command line in the order listed. After all files have been processed, bc reads from the standard input. All code is executed as it is read. (If a file contains a command to halt the processor, bc will never read from the standard input.)

This version of bc contains several extensions beyond traditional bc implementations and the POSIX draft standard. Command-line options can cause these extensions to print a warning or to be rejected. This document describes the language accepted by this processor. Extensions will be identified as such.

The bc utility accepts the following options and operand:

-l Defines the standard math library.

-w Gives warnings for extensions to POSIX bc.

-s Processes exactly the POSIX bc language.

file The file or files to be processed.

The most basic element in bc is the number. Numbers are arbitrary precision numbers. This precision is both in the integer part and the fractional part. All numbers are represented internally in decimal and all computation is done in decimal. (This version truncates results from divide and multiply operations.) There are two attributes of numbers, the *length* and the *scale*. The length is the total number of significant decimal digits in a number and the scale is the total number of decimal digits after the decimal point. For example:

```
.000001 has a length of 6 and scale of 6.
1935.000 has a length of 7 and a scale of 3.
```

Numbers are stored in two types of variables, simple variables and arrays. Both simple variables and array variables are named. Names begin with a letter followed by any number of letters, digits and underscores. All letters must be lowercase. (Full alphanumeric names are an extension. In POSIX `bc`, all names are a single lowercase letter.) The type of variable is clear by the context, because all array variable names will be followed by brackets (`[]`).

There are four special variables, *scale*, *ibase*, *obase*, and *last*. *scale* defines how some operations use digits after the decimal point. The default value of *scale* is 0. *ibase* and *obase* define the conversion base for input and output numbers. The default for both input and output is base 10. *last* (an extension) is a variable that has the value of the last printed number. These will be discussed in further detail where appropriate. All of these variables may have values assigned to them as well as used in expressions.

Comments in `bc` start with the characters `/*` and end with the characters `*/`. Comments may start anywhere and appear as a single space in the input. (This causes comments to delimit other input items. For example, a comment can not be found in the middle of a variable name.) Comments include any newlines (end-of-line) between the start and the end of the comment.

Expressions and Statements

The numbers are manipulated by expressions and statements. Since the language was designed to be interactive, statements and expressions are executed as soon as possible. There is no "main" program. Instead, code is executed as it is encountered. (Functions, discussed in detail later, are defined when encountered.)

A simple expression is just a constant. `bc` converts constants into internal decimal numbers using the current input base, specified by the variable *ibase*. (There is an exception in functions.) The legal values of *ibase* are 2 through 16 (F). Assigning a value outside this range to *ibase* will result in a value of 2 or 16. Input numbers may contain the characters 0–9 and A–F. (NOTE: They must be uppercase. Lowercase letters are variable names.) Single-digit numbers always have the value of the digit, regardless of the value of `ibase` (for example, A = 10). For multidigit numbers, `bc` changes all input digits greater or equal to *ibase* to the value of *ibase*–1. This makes the number `FFF` always be the largest 3-digit number of the input base.

Full expressions are similar to many other high-level languages. Since there is only one kind of number, there are no rules for mixing types. Instead, there are rules on the scale of expressions. Every expression has a scale. This is derived from the scale of original numbers, the operation performed and in many cases, the value of the variable *scale*. Legal values of the variable *scale* are 0 to the maximum number representable by a C integer.

In the following descriptions of legal expressions, *expr* refers to a complete expression and *var* refers to a simple or an array variable. A simple variable is just a *name* and an array variable is specified as *name*[*expr*]. Unless specifically mentioned, the *scale* of the result is the maximum scale of the expressions involved.

`-` *expr* The result is the negation of the expression.

`++` *var* The variable is incremented by one and the new value is the result of the expression.

`--` *var* The variable is decremented by one and the new value is the result of the expression.

var ++ The result of the expression is the value of the variable, and then the variable is incremented by one.

var -- The result of the expression is the value of the variable, and then the variable is decremented by one.

expr + *expr* The result of the expression is the sum of the two expressions.

expr - *expr* The result of the expression is the difference of the two expressions.

expr * *expr* The result of the expression is the product of the two expressions.

expr / *expr* The result of the expression is the quotient of the two expressions. The scale of the result is the value of the variable *scale*.

expr % *expr* The result of the expression is the "remainder," and it is computed in the following way. To compute *a*%*b*, first *a*/*b* is computed to *scale* digits. That result is used to compute *a*-(*a*/*b*)**b* to the scale of the maximum of *scale*+*scale*(b) and *scale*(a). If *scale* is set to zero, and both expressions are integers, this expression is the integer remainder function.

expr ^ *expr* The result of the expression is the value of the first raised to the second. The second expression must be an integer. (If the second expression is not an integer, a warning is generated, and the expression is truncated to get an integer value.) The scale of the result is *scale* if the exponent is negative. If the exponent is positive, the scale of the result is the minimum of the scale of the first expression times the value of the exponent and the maximum of *scale* and the scale of the first expression (for example, *scale*(*a*^*b*) = *min*(*scale*(*a*)***b*, *max*(*scale*, *scale*(*a*))). It should be noted that *expr*^0 will always return the value of 1.

(*expr*) This alters the standard precedence to force the evaluation of the expression.

var = *expr* The variable is assigned the value of the expression.

var <op>= *expr* This is equivalent to *var* = *var* <op> *expr* with the exception that the *var* part is evaluated only once. This can make a difference if *var* is an array.

Relational expressions are a special kind of expression that always evaluate to 0 or 1, 0 if the relation is false and 1 if the relation is true. These may appear in any legal expression. (POSIX bc requires that relational expressions are used only in if, while, and for statements and that only one relational test may be done in them.) The relational operators are as follows:

expr1 < *expr2* The result is 1 if *expr1* is strictly less than *expr2*.

expr1 <= *expr2* The result is 1 if *expr1* is less than or equal to *expr2*.

expr1 > *expr2* The result is 1 if *expr1* is strictly greater than *expr2*.

expr1 >= *expr2* The result is 1 if *expr1* is greater than or equal to *expr2*.

expr1 == *expr2* The result is 1 if *expr1* is equal to *expr2*.

expr1 != *expr2* The result is 1 if *expr1* is not equal to *expr2*.

Boolean operations are also legal. (POSIX bc does **not** have boolean operations). The result of all boolean operations are 0 and 1 (for false and true) as in relational expressions. The boolean operators are as follows:

!*expr* The result is 1 if *expr* is 0.

expr && *expr* The result is 1 if both expressions are nonzero.

expr || *expr* The result is 1 if either expression is nonzero.

The expression precedence is as follows: (lowest to highest)

```
|| operator, left associative
&& operator, left associative
! operator, nonassociative
Relational operators, left associative
Assignment operator, right associative
+ and - operators, left associative
*, / and % operators, left associative
^ operator, right associative
unary - operator, nonassociative
++ and -- operators, nonassociative
```

This precedence was chosen so that POSIX-compliant bc programs will run correctly. This will cause the use of the relational and logical operators to have some unusual behavior when used with assignment expressions. Consider the expression:

```
a = 3 < 5
```

Most C programmers would assume this would assign the result of 3 < 5 (the value 1) to the variable *a*. What this does in bc is assign the value 3 to the variable *a* and then compare 3 to 5. It is best to use parentheses when using relational and logical operators with the assignment operators.

A few more special expressions are provided in bc. These have to do with user-defined functions and standard functions. They all appear as *name*(*parameters*). See the Functions subsection for user-defined functions. The standard functions are as follows:

length (*expression*)
: The value of the length function is the number of significant digits in the expression.

read ( )
: The read function (an extension) will read a number from the standard input, regardless of where the function occurs. WARNING: This can cause problems with the mixing of data and program in the standard input. The best use for this function is in a previously written program that needs input from the user, but never allows program code to be input from the user. The value of the read function is the number read from the standard input using the current value of the variable ibase for the conversion base.

`scale ( expression )`
: The value of the `scale` function is the number of digits after the decimal point in the expression.

`sqrt ( expression )`
: The value of the `sqrt` function is the square root of the expression. If the expression is negative, a run-time error is generated.

Statements

Statements (as in most algebraic languages) provide the sequencing of expression evaluation. In `bc`, statements are executed "as soon as possible." Execution happens when a newline is encountered, and there is one or more complete statement. Due to this immediate execution, newlines are very important in `bc`. In fact, both a semicolon and a newline are used as statement separators. An improperly placed newline will cause a syntax error. Because newlines are statement separators, it is possible to hide a newline by using the backslash character (\). The sequence \<*nl*>, where <*nl*> is the newline, appears to `bc` as white space instead of a newline. A statement list is a series of statements separated by semicolons and newlines. The following is a list of `bc` statements and what they do. (Items enclosed in brackets (`[]`) are optional parts of the statement.)

`expression`
: This statement does one of two things. If the expression starts with <*variable*> <*assignment*> ..., it is considered to be an assignment statement. If the expression is not an assignment statement, the expression is evaluated and printed to the output. After the number is printed, a newline is printed. For example, *a*`=1` is an assignment statement, and `(`*a*`=1)` is an expression that has an embedded assignment. All numbers that are printed are printed in the base specified by the variable *obase*. The legal values for *obase* are 2 through `BC_BASE_MAX`. (See the Limits subsection.) For bases 2 through 16, the usual method of writing numbers is used. For bases greater than 16, `bc` uses a multicharacter digit method of printing the numbers, in which each higher base digit is printed as a base-10 number. The multicharacter digits are separated by spaces. Each digit contains the number of characters required to represent the base-10 value of *obase*`-1`. Since numbers are of arbitrary precision, some numbers may not be printable on a single output line. These long numbers will be split across lines using the \ as the last character on a line. The maximum number of characters printed per line is 70. Due to the interactive nature of `bc`, printing a number causes the side effect of assigning the printed value to the special variable *last*. This allows the user to recover the last value printed without having to retype the expression that printed the number. Assigning to *last* is legal and will overwrite the last printed value with the assigned value. The newly assigned value will remain until the next number is printed or another value is assigned to *last*.

`string`
: The string is printed to the output. Strings start with double quotation marks and contain all characters until the next double quotation marks. All characters are taken literally, including any newline. No newline character is printed after the string.

`print` *list* The `print` statement (an extension) provides another method of output. The *list* is a list of strings and expressions separated by commas. Each string or expression is printed in the order of the list. No terminating newline is printed. Expressions are evaluated and their value is printed and assigned the variable `last`. Strings in the `print` statement are printed to the output and may contain special characters. Special characters start with the backslash character (\). The special characters recognized by `bc` are `b` (bell), `f` (form feed), `n` (newline), `r` (carriage return), `t` (tab), and \ (backslash). Any other character following the backslash will be ignored. This still does not allow the double quote character to be part of any string.

`{ statement_list }`
This is the compound statement. It allows multiple statements to be grouped together for execution.

`if (`*expression*`) then` *statement1* `[else` *statement2*`]`
The `if` statement evaluates the expression and executes *statement1* or *statement2* depending on the value of the expression. If the expression is nonzero, *statement1* is executed. If *statement2* is present, and the value of the expression is 0, then *statement2* is executed. (The `else` clause is an extension.)

`while (`*expression*`)` *statement*
The `while` statement will execute the statement while the expression is nonzero. It evaluates the expression before each execution of the statement. Termination of the loop is caused by a zero expression value or the execution of a break statement.

`for ([`*expression1*`] ; [`*expression2*`] ; [`*expression3*`])` *statement*
The `for` statement controls repeated execution of the statement. *expression1* is evaluated before the loop. *expression2* is evaluated before each execution of the statement. If it is nonzero, the statement is evaluated. If it is zero, the loop is terminated. After each execution of the statement, *expression3* is evaluated before the reevaluation of *expression2*. If *expression1* or *expression3* is missing, nothing is evaluated at the point at which it would be evaluated. If *expression2* is missing, it is the same as substituting the value 1 for *expression2*. (The optional expressions are an extension. POSIX `bc` requires all three expressions.) The following is equivalent code for the `for` statement:

```
expression1;
while (expression2) {
   statement;
   expression3;
}
```

`break` The `break` statement causes a forced exit of the most recent enclosing `while` statement or `for` statement.

`continue` The `continue` statement (an extension) causes the most recent enclosing for *statement* to start the next iteration.

`halt` The `halt` statement (an extension) is an executed statement that causes the `bc` processor to quit only when it is executed. For example, `if (0 == 1) halt` will not cause `bc` to terminate, because the halt is not executed.

`return` Returns the value 0 from a function. (See the Functions subsection.)

`return (`*expression*`)`
Returns the value of the expression from a function. (See the Functions subsection.)

Pseudo Statements

These statements are not statements in the traditional sense. They are not executed statements. Their function is performed at "compile" time.

`limits` Prints the local limits enforced by the local version of `bc`. This is an extension.

`quit` When the `quit` statement is read, the `bc` processor is terminated, regardless of where the `quit` statement is found. For example, `if (0 == 1)`, `quit` will cause `bc` to terminate.

`warranty` Prints a longer warranty notice. This is an extension.

Functions

Functions provide a method of defining a computation that can be executed later. Functions in `bc` always compute a value and return it to the caller. Function definitions are *dynamic* in the sense that a function is undefined until a definition is encountered in the input. That definition is then used until another definition function for the same name is encountered. The new definition then replaces the older definition. A function is defined as follows:

`define` *name* `(`*parameters*`) {` *newline*
auto_list *statement_list* `}`

A function call is just an expression of the form *name*`(`*parameters*`)`.

Parameters are numbers or arrays (an extension). In the function definition, zero or more parameters are defined by listing their names separated by commas. Numbers are only call-by-value parameters. Arrays are only call-by-variable. Arrays are specified in the parameter definition by the notation *name*`[]`. In the function call, actual parameters are full expressions for number parameters. The same notation is used for passing arrays as for defining array parameters. The named array is passed by variable to the function. Since function definitions are dynamic, parameter numbers and types are checked when a function is called. Any mismatch in number or types of parameters will cause a run-time error. A run-time error will also occur for the call to an undefined function.

The *auto_list* is an optional list of variables that are for local use. The syntax of the auto list (if present) is `auto` *name*`, ... ;`. (The semicolon is optional.) Each *name* is the name of an auto variable. Arrays may be specified by using the same notation as used in parameters. These variables have their values pushed onto a stack at the start of the function. The variables are then initialized to zero and used throughout the execution of the function. At function exit, these variables are popped so that the original values (at the time of the function call) of these variables are restored. The parameters are really auto variables that are initialized to a value provided in the function call. Auto variables are different than traditional local variables in that if function A calls function B, B may access function A's auto variables by just using the same name, unless function B has called them auto variables. Due to the fact that auto variables and parameters are pushed onto a stack, `bc` supports recursive functions.

The function body is a list of `bc` statements. Again, statements are separated by semicolons or newlines. Return statements cause the termination of a function and the return of a value. There are two versions of the return statement. The first form, `return`, returns the value 0 to the calling expression. The second form, `return (`*expression*`)`, computes the value of the expression and returns that value to the calling expression. There is an implied `return (0)` at the end of every function. This allows a function to terminate and return 0 without an explicit return statement.

Functions also change the usage of the variable *ibase*. All constants in the function body will be converted using the value of *ibase* at the time of the function call. Changes of *ibase* will be ignored during the execution of the function except for the standard function `read`, which will always use the current value of *ibase* for conversion of numbers.

Math Library

If `bc` is invoked with the `-l` option, a math library is preloaded and the default scale is set to 20. The math functions will calculate their results to the scale set at the time of their call. The math library defines the following functions:

`s (`*x*`)` The sine of *x* in radians.

`c (`*x*`)` The cosine of *x* in radians.

`a (`*x*`)` The arctangent of *x*.

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

`e (`*x*`)` The exponential function of raising `e` to the value *x*.

`j (`*n*`,`*x*`)` The Bessel function of integer order *n* of *x*.

Differences

This version of `bc` was implemented from the POSIX P1003.2/D11 draft and contains several differences and extensions relative to the draft and traditional implementations. It is not implemented in the traditional way using `odc`(1). This version is a single process which parses and runs a byte code translation of the program. There is an undocumented option (`-c`) that causes the program to output the byte code to the standard output instead of running it. It was mainly used for debugging the parser and preparing the math library.

A major source of differences is extensions, where a feature is extended to add more functionality, and additions, where new features are added. The following is the list of differences and extensions.

`LANG` environment
: This version does not conform to the POSIX standard in the processing of the `LANG` environment variable and all environment variables starting with `LC_`.

`names`
: Traditional and POSIX `bc` have single–letter names for functions, variables and arrays. They have been extended to be multicharacter names that start with a letter and may contain letters, numbers, and the underscore character.

`Strings`
: Strings are not allowed to contain NUL characters. POSIX says all characters must be included in strings.

`last`
: POSIX `bc` does not have a *last* variable.

`comparisons`
: POSIX `bc` allows comparisons only in the `if` statement, the `while` statement, and the second expression of the `for` statement. Also, only one relational operation is allowed in each of those statements.

`if statement, else clause`
: POSIX `bc` does not have an `else` clause.

`for statement`
: POSIX `bc` requires all expressions to be present in the `for` statement.

`&&, ||, !`
: POSIX `bc` does not have the logical operators.

`read function`
: POSIX `bc` does not have a read function.

`print statement"`
: POSIX `bc` does not have a `print` statement.

`continue statement`
: POSIX `bc` does not have a `continue` statement.

`array parameters`
: POSIX `bc` does not have `array` parameters. Other implementations of `bc` may have call by value array parameters.

`=+, =-, =*, =/, =%, =^`
: POSIX `bc` does not require these "old style" assignment operators to be defined. This version may allow these "old style" assignments. Use the `limits` statement to see if the installed version supports them. If it does support the "old style" assignment operators, the statement `a =- 1` will decrement `a` by 1 instead of setting `a` to the value –1.

`spaces in numbers`
: Other implementations of `bc` allow spaces in numbers. For example, `x=1 3` would assign the value 13 to the variable *x*. The same statement would cause a syntax error in this version of `bc`.

errors and execution

This implementation varies from other implementations in terms of what code will be executed when syntax and other errors are found in the program. If a syntax error is found in a function definition, error recovery tries to find the beginning of a statement and continue to parse the function. Once a syntax error is found in the function, the function will not be callable and becomes undefined. Syntax errors in the interactive execution code will invalidate the current execution block. The execution block is terminated by an end-of-line that appears after a complete sequence of statements. For example,

```
a = 1
b = 2
```

has two execution blocks and

```
{ a = 1
  b = 2 }
```

has one execution block. Any run-time error will terminate the execution of the current execution block. A run-time warning will not terminate the current execution block.

Interrupts — During an interactive session, the SIGINT signal (usually generated by the <CONTROL-c> character from the terminal) will cause execution of the current execution block to be interrupted. It will display a run-time error indicating which function was interrupted. After all run-time structures have been cleaned up, a message will be printed to notify the user that bc is ready for more input. All previously defined functions remain defined, and the value of all nonauto variables are the value at the point of interruption. All auto variables and function parameters are removed during the cleanup process. During a noninteractive session, the SIGINT signal will terminate the entire run of bc.

Limits

The following are the limits currently in place for this bc processor. Some of them may have been changed by an installation. Use the limits statement to see the actual values.

BC_BASE_MAX — The maximum output base is currently set at 999. The maximum input base is 16.

BC_DIM_MAX — This is currently an arbitrary limit of 65,535 as distributed. Your installation may be different.

BC_SCALE_MAX — The number of digits after the decimal point is limited to INT_MAX digits. Also, the number of digits before the decimal point is limited to INT_MAX digits.

BC_STRING_MAX — The limit on the number of characters in a string is INT_MAX characters.

exponent — The value of the exponent in the raise operation (^) is limited to LONG_MAX.

multiply — The multiply routine may yield incorrect results if a number has more than LONG_MAX / 90 total digits. For 32-bit longs, this number is 23,860,929 digits.

`code size`	Each function and the "main" program are limited to 10,240 bytes of compiled byte code each. This limit (`BC_MAX_SEGS`) can be changed easily to have more than 10 segments of 1024 bytes.
`variable names`	The current limit on the number of unique names is 32,767 for each of simple variables, arrays, and functions.

EXAMPLES

Example 1: In `/bin/sh`, the following will assign the value of pi to the shell variable `pi`.

```
pi=$(echo "scale=10; 4*a(1)" | bc -l)
```

Example 2: The following is the definition of the exponential function used in the math library. This function is written in POSIX bc.

```
scale = 20

/* Uses the fact that e^x = (e^(x/2))^2
   When x is small enough, we use the series:
     e^x = 1 + x + x^2/2! + x^3/3! + . . .
*/

define e(x) {
  auto  a, d, e, f, i, m, v, z

  /* Check the sign of x. */
  if (x<0) {
    m = 1
    x = -x
  }

  /* Precondition x. */
  z = scale;
  scale = 4 + z + .44*x;
  while (x > 1) {
    f += 1;
    x /= 2;
  }

  /* Initialize the variables. */
  v = 1+x
  a = x
  d = 1

  for (i=2; 1; i++) {
    e = (a *= x) / (d *= i)
    if (e == 0) {
      if (f>0) while (f--)  v = v*v;
      scale = z
      if (m) return (1/v);
      return (v/1);
    }
    v += e
  }
}
```

Example 3: The following is code that uses the extended features of bc to implement a simple program for calculating checkbook balances. This program is best kept in a file so that it can be used many times without having to retype it at every use.

```
scale=2
print "\nCheck book program!\n"
print "  Remember, deposits are negative transactions.\n"
print "  Exit by a 0 transaction.\n\n"

print "Initial balance? "; bal = read()
bal /= 1
print "\n"
while (1) {
  "current balance = "; bal
  "transaction? "; trans = read()
  if (trans == 0) break;
  bal -= trans
  bal /= 1
}
quit
```

The following is the definition of the recursive factorial function:

```
define f (x) {
  if (x <= 1) return (1);
  return (f(x-1) * x);
}
```

FILES

In most installations, bc is completely self-contained. Where executable size is of importance or the C compiler does not deal with very long strings, bc will read the standard math library from the file /usr/lib/lib.b.

DIAGNOSTICS

If any file on the command line cannot be opened, bc will report that the file is unavailable and terminate. Also, there are compile-time and run-time diagnostics that should be self-explanatory.

SEE ALSO

awk(1), obc(1), odc(1)

NAME

bdiff – Compares very large files for differences

SYNOPSIS

bdiff *file1 file2* [*n*] [-s]

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The bdiff utility is used in a manner analogous to that of diff(1) to find lines that must be changed in two files to bring them into agreement. Its purpose is to allow the processing of files that are too large for diff. The bdiff utility 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 bdiff utility accepts the following arguments and options:

file1 First file to compare; required.

file2 Second file to compare; required. If *file1* or *file2* is -, bdiff reads the standard input.

n Causes bdiff to split remainder of file into *n*-line segments, rather than the default 3500-line segments. *n* must be numeric. This option is useful when 3500-line segments are too large for diff, causing bdiff to fail.

-s Specifies that diagnostics are not to be printed by bdiff (however, this does not suppress possible exclamations by diff(1)).

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

The bdiff utility's output is exactly like 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 the smallest sufficient set of file differences.

NOTES

The bdiff utility will exit with a nonzero status even if there are differences between the input files. This behavior is necessary because bdiff is invoked by delta(1).

MESSAGES

Use help(1) for explanations.

FILES

`/tmp/bd`*name* Temporary working file

SEE ALSO

`delta`(1), `diff`(1), `help`(1)

NAME

bftp – Provides user interface to the background file transfer program

SYNOPSIS

/usr/ucb/bftp

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The bftp command is the user interface to the background file transfer program (BFTP). You can use bftp to submit a request for a future file transfer by using the standard Internet file transfer protocol (FTP), which is described in RFC 959.

Because bftp makes use of third-party FTP, the source and the destination hosts do not have to be operational at the time the request is submitted. Transfers are scheduled locally through the system batch processor using the at(1) utility. Therefore, the exact time at which a file transfer occurs depends on how often jobs are run from the system batch queue.

For more information on BFTP, see RFC 1068.

The bftp command recognizes the following commands:

append true | false
: Appends the transferred file to the existing file if the destination file exists and this option is true. If this option is false and the file name is already in use, the existing file is replaced if write permission on the destination host is enabled.

command copy | delete | move
: Sets the source file disposition to copy, move (which is equivalent to copy and delete), or simply delete the source file. The default is copy. These commands set the file transfer mode.

d-dir *destination-directory*
: Establishes the destination directory into which the file will be transferred. The directory name must end with a directory-delimiter character.

d-file *filename*
: Sets the destination file name, which is the name of the file when the transfer is complete.

d-host *hostname*
: Sets the destination host name, which is the host to which the file will be transferred.

d-password *password*
: Specifies the password on the destination host.

`d-user` *user*
: Specifies the user name (login) on the destination host.

`explain` Displays a short explanation on the use of `bftp`.

`find` Locates and displays a previously submitted BFTP job. `bftp` prompts for the (optional) *RequestID* and the *RequestKeyword*. After a request is located and displayed, you can change and resubmit it, or cancel it.

`help-all`
: Displays a description of each command.

`help` [*command*]
: Prints an informative message about the meaning of *command*. If you do not specify an argument, `bftp` prints a list of the commands and their descriptions.

`init` Discards all information about a request. This command resets `bftp` to the default startup values.

`interval`
: Displays starting retry interval (in minutes) and number of retries.

`mailbox` *mailbox-name*
: Sends the completion notification message to this address.

`mode stream | block | compress`
: Sets the FTP mode. The default value is `stream`.

`multiple true | false`
: When `true`, transfers all files that match the pattern set in `s-file` to be transferred. Matching is performed on the source host.

`prompt` Prompts for all of the necessary information to set up a transfer.

`quit` Ends the current `bftp` session.

`r-delete` *request-file*
`r-list`
`r-load` *request-file*
`r-store` *request-file*
: The request commands manage request files, which are used to save BFTP requests for future use.

`s-acct` and `d-acct`
: Sets the login account name for the transaction. Usually, this option is not needed.

`s-dir` *source-directory*
: Sets the source directory from which the file will be transferred. The directory name must end with a directory-delimiter character.

`s-file` *source-filename*
: Sets the source file name of the file to be transferred.

`s-host` *hostname*
Sets the source host name, which is the host from which the file will be transferred.

`s-password` *password*
Specifies the password on the source host.

`s-port | d-port` *n*
Sets the FTP transaction source or destination port number, which is usually 21.

`s-user` *user*
Specifies the user name (login) on the source host.

`show` Displays the current parameter values.

`simple-example`
Displays an example that shows how to submit a request.

`stru file | record | page`
Sets the FTP structure.

`submit` Places the request in the batch queue. `bftp` prompts for the *StartTime* and the *RequestKeyword*.

`time` *StartTime*
Sets the date and time at which the file will be transferred.

`transfer`
Performs the requested transfer now.

`type image | ascii | ebcdic | local | binary`
Sets the FTP file type. The `ascii` and `ebcdic` types have further parameters of `nonprint`, `telnet`, and `carriage-control`. The default type is `ascii nonprint`. The representation type may be one of network ASCII, EBCDIC, image, or local byte size with a specified byte size (usually for PDP-10s and PDP-20s). The network ASCII and EBCDIC types have a further subtype that specifies whether vertical format control (new-line characters, form feeds, and so on) will be passed through (nonprint), provided in TELNET format, or provided in ASA carriage control format.

`unique true | false`
Sets unique mode. If `unique` is `true`, the destination file name is guaranteed to be unique. This prevents the replacement of old files with new files of the same name.

`verbose true | false`
Sets verbose mode. When `verbose` is `true`, `bftp` prints each command used in the transaction during the verify and the transfer operations.

`verify` Validates the request. The current request is checked to determine whether or not all needed information was entered. Then `bftp` connects to the specified hosts to determine whether or not the specified parameters are supported.

FILES

The bftp command creates the following files that keep track of requests that are in progress:

b*123456789*.cmd
b*123456789*.lis
b*123456789*.msg
b*123456789*.req

The following files are saved by using the r-store command:

s.*request-name*

Usually, bftp stores request files in the home directory of the user who is logged on. To have bftp store these files in another directory, use the system setenv command to set $BFTPDIR. For example:

```
setenv BFTPDIR ~yourname/.bftp
```

SEE ALSO

at(1), crontab(1), ftp(1B)

cron(8), ftpd(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

bg – Runs jobs in the background

SYNOPSIS

bg [*job_id* ...]

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The bg utility resumes suspended jobs from the current shell execution environment (see sh(1)) by running the jobs as background jobs. If the job specified by *job_id* is already a running background job, the bg utility has no effect and exits immediately.

Using bg to place a job into the background causes its process ID to become "known in the current shell execution environment," as if it had been started as an asynchronous list.

The bg utility supports the following operand:

job_id Specifies the job to be resumed as a background job. If you omit *job_id*, the most recently suspended job is used. For a description of the *job_id* format, see sh(1).

For information about running a job in the foreground, see fg(1).

NOTES

The bg utility is a built-in utility to the standard shell (sh(1)). An executable version of this utility is available in /usr/bin/bg.

EXIT STATUS

The bg utility exits with one of the following values:

0 Successful completion.

>0 An error occurred.

SEE ALSO

fg(1), jobs(1), sh(1)

NAME

bld – Maintains relocatable libraries

SYNOPSIS

bld *key*[*opts*] [*position-obj*] *bldname* [*args*]
bld R *bldname old-obj-name new-obj-name*

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The bld utility collects relocatable modules into a relocatable library. The library can be maintained and manipulated with the bld utility.

The bld utility line has two formats as shown in the SYNOPSIS section. The first format is used to physically manipulate modules in a library. The second fomrat is used to change the names of modules in the library; only the <R> key (without *opts*) is used in this version.

A *position-obj* or any other object is defined as *file*:*module*. *file* may be a full path name. For example, the following refers to the module MODULE that came from the file file.o:

```
file.o:MODULE
```

In the example, file.o (or file.o:) refers to all modules in the file file.o, and :MODULE refers to the module MODULE, which has no file name associated with it. If MODULE contains a '$', the '$' must be escaped so it is protected from being expanded by the shell.

The bld utility accepts the following arguments:

key Keyletters are as follows:

- d Deletes the named objects from the library.
- m Moves the named modules to the end of the library. If a positioning character is present, the *position-obj* argument must be present and specifies where the modules are to be moved.
- p Prints the named modules in the library in binary format to stdout.
- q Takes the relocatable modules specified in *args* and appends them to the end of the library. Optional positioning characters are not valid. The command does not check whether the added modules are already in the library.
- r Takes the relocatable modules specified in *args* and replaces them in the library.

  WARNING: The command does not check for duplicate names in the argument list.

`t` Prints a table of contents of the library. If no names are specified, all modules in the library are used for the table of contents. If names are specified, only those modules are used for the table of contents.

`x` Extracts the named modules. If no names are specified, all modules in the library are extracted. The modules are placed in a file whose name is the same as the *file* or, if none, the *module*. In neither case does `x` alter the library file.

`R` Renames an object. (This key is used only with the second form of the command line; no *opts* may be used.) This lets users change the name of an object or part of the name of an object. For example:

`bld R bld.a foo.o bar.o` changes the file name of every module that came from `foo.o` to `bar.o`.

`bld R bld.a foo.o:BAR bar.o` changes the file name associated with the module `BAR` to `bar.o`.

`bld R bld.a :BAR foo.o:BAR` adds the file name `foo.o` to module `BAR`.

`bld R bld.a foo.o:BAR :BAR` removes the file name `foo.o` from module `BAR`

`S` Keeps files in sync and ensures that there is a one-to-one mapping with an `ar`(1) library. It may add, replace, or delete modules in the build library based on the `ar` library.

opts Option letters are as follows:

`a` Places new modules after *position-obj*. This option works only with the `<r>` and `<m>` keys.

`b` Places new modules before *position-obj*. This option works only with the `<r>` and `<m>` keys.

`c` Suppresses the message that is produced by default when *bldname* is created.

`e` Displays all entry points in the specified modules. This option works only with the `<t>` key.

`i` Places new modules before *position-obj*. This option works only with the `<r>` and `<m>` keys.

`l` Places temporary files in the local current working directory `/tmp`. By default, temporary files are placed in the directory specified by `TMPDIR`.

`v` Gives a verbose module-by-module description of the making of a new library file from the old library and the constituent modules. When used with `t`, this option gives a long listing of all information about the modules. When used with `x`, it precedes each module with a name.

`z` Inhibits the association of a file from which the module comes. This feature allows for the replacement of a module of the same name from a different compilation unit (`.o`). This is useful in Fortran applications in which subroutines are generated by `cft77` as multiple module units. This option may be used only when the `r` and `q` options are specified. This option should be used only to emulate the operation of the COS `BUILD` utility. Its use is inappropriate within the context of a UNICOS makefile or nmakefile. See the EXAMPLES section.

position-obj Specifies that new modules are to be placed after (`a`) or before (`b` or `i`) *position-obj*. Otherwise, new files are placed at the end. *position-obj* is required when the `a`, `b`, or `i` options are used.

bldname Specifies the library. If the library file *bldname* exists, it must be a `bld`-formatted archive. If the `q`, `r`, or `s` keys are specified and *bldname* does not exist, `bld` will create the library *bldname*.

args Names of files containing relocatable modules when using the `r` or `q` keys. Otherwise, names of modules in the library, *bldname*.

old-obj-name Old name of the module.

new-obj-name New name of the module.

NOTES

The `bld` utility has the following limitations:

- Currently, the `bld` utility does not interpret CRAY T3D object files. Use `ar`(1) to process CRAY T3D object files.
- If the file names in the `bld` archive have subdirectories and if the subdirectories do not exist, `bld` will error exit when extracting files from a `bld` archive.
- When the default (no `z` *opt*) is used, a file name is *permanently* associated with the one or more modules contained in the file. Once generated into a `bld` archive, the file name is retained, even after extraction. To clear the file name from the module, use the `<R>` key.
- The `-z` option is not supported with Fortran 90.
- If `bld` cannot access one of the files in an argument list, then none of the files will be added to the library. This behavior differs from the `ar`(1) utility, which adds all of the files on the argument list that can be successfully accessed.

EXAMPLES

Example 1: The following example shows the use of the `z` *opt* in a Fortran application in which subroutines are generated by as multiple module units. In this example, `filea.o` contains the modules `SUBA` and `SUBB`:

```
$ bld qvz bldname filea.o
```

Example 2: By compiling file `fileb.f`, which contains subroutine modules `SUBB` and `SUBC`, `SUBB` and `SUBC` can be replaced using the following command:

```
$ bld rvz bldname fileb.o
```

This will overlay `SUBB` even though it was created in a different `.o` file. When the default (no `z` *opt*) is used, a file name is *permanently* associated with the one or more modules contained in the file. Once generated into a `bld` archive, the file name is retained, even after extraction. To clear the file name from the module, use the `<R>` key.

Example 3: The following example shows a variety of `bld` uses. The shell prompt is indicated by a `$`.

```
$ ls
malloc.c memmgr.c memmgr.h
$ cc -c malloc.c
$     #Initial insertion of malloc.o
$ bld q lib.a malloc.o
bld: creating lib.a
$ cc -c memmgr.c
$     #Initial insertion of memmgr.o
$ bld q lib.a memmgr.o
$     #Table of contents
$ bld t lib.a
malloc.o:malloc$c
memmgr.o:memmgr$c
$     #Verbose table of contents
$ bld tv lib.a
  Date      Time    Compiler  Version   OS Vers.  Object Name
01/27/91  10:59:28  Std C     2X065406    7.0     malloc.o:malloc$c
01/27/91  11:00:41  Std C     2X065406    7.0     memmgr.o:memmgr$c
$     #Assume that malloc.c has changes
$ cc -c malloc.c
$     #Verbose replace
$ bld rv lib.a malloc.o
r - malloc.o:malloc$c
$     #Verbose table of contents
$ bld tv lib.a
  Date      Time    Compiler  Version   OS Vers.  Object Name
01/27/91  11:04:08  Std C     2X065406    7.0     malloc.o:malloc$c
01/27/91  11:00:41  Std C     2X065406    7.0     memmgr.o:memmgr$c
$     #Notice the updated time of only malloc.o
$ rm malloc.o memmgr.o
$ ls
lib.a malloc.c memmgr.c memmgr.h
$     #Extraction of memmgr.o
$ bld x lib.a memmgr.o
```

```
$ ls
lib.a malloc.c memmgr.c memmgr.h memmgr.o
$     #The extraction did not modify lib.a
$ bld tv lib.a
  Date      Time    Compiler  Version   OS Vers.  Object Name
01/27/91  11:04:08  Std C     2X065406    7.0     malloc.o:malloc$c
01/27/91  11:00:41  Std C     2X065406    7.0     memmgr.o:memmgr$c
$     #Move malloc.o after memmgr.o
$ bld ma memmgr.o:memmgr\$c lib.a malloc.o
$ bld tv lib.a
  Date      Time    Compiler  Version   OS Vers.  Object Name
01/27/91  11:00:41  Std C     2X065406    7.0     memmgr.o:memmgr$c
01/27/91  11:04:08  Std C     2X065406    7.0     malloc.o:malloc$c
$     #Removing memmgr.o
$ bld d lib.a memmgr.o
$ bld tv lib.a
  Date      Time    Compiler  Version   OS Vers.  Object Name
01/27/91  11:04:08  Std C     2X065406    7.0     malloc.o:malloc$c
```

FILES

TMPDIR/bld*	Temporary files

BUGS

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

SEE ALSO

ar(1), nm(1), segldr(1)

a.out(5), bld(5), relo(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

builddefs – Reads a definitions file that has embedded keywords to produce a keyword file and a definitions file without embedded keywords

SYNOPSIS

builddefs *infile deffile*

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The builddefs utility reads a definitions file that has embedded keywords to produce a keyword file and a definitions file without embedded keywords. A pair of keywords in the form ++*term* marks each definition in the input file. Synonyms for the term, if any, also begin with ++ and follow the keyword line (do not mark synonyms in pairs). You should enter all keywords and synonyms into the definitions file and use the correct capitalization conventions. The keywords and synonyms are recorded in the keyword file in lowercase characters.

Example:

```
++access control list
++ACL
  An access control list (ACL) contains the individual/group
  identifiers and the individual/group access modes for the subjects
  who will be allowed to access the file.  See also discretionary
  access control.
++access control list
```

The builddefs utility checks all keywords for keywording errors. The following rules apply to keywording:

- The two ++ symbols that appear in columns 1 and 2 of the intermediate definitions file identify a keyword. The keyword immediately follows the ++ symbols, with no intervening blank spaces or tabs. Empty keywords (that is, ++ with no following text) are not allowed.
- A keyword consists of up to 48 characters. If a keyword is longer than 48 characters, it will be truncated.
- Each definition must have two keywords (a matching pair). The first keyword indicates the start of the definition. The second keyword indicates the end of the definition.
- Synonyms for a keyword are in the form ++*synonym* and are limited to 48 characters. Do not mark synonyms in pairs.

The builddefs utility accepts the following arguments:

infile Input file.

deffile Output file.

Running the builddefs utility produces the output file *deffile* and a keyword file *deffile*_k.

NOTES

The builddefs utility lets you modify the Cray Research definitions file (CRAYdefs_i) or add local definitions files. The following subsections describe these procedures.

Modifying the Cray Research Definitions File

The following procedure shows how to use builddefs to modify the Cray Research definitions file that the define(1) utility uses.

If you modify the Cray Research definitions file, your changes will be lost during the installation of a UNICOS revision or update. In this case, back up the modified definitions file, install the UNICOS revision or update, and then reapply the modified definitions file.

1. Copy the CRAYdefs_i definitions file, which has embedded keywords, from the default definitions directory, /usr/lib/define, to your working directory.
2. Edit the file to make the desired changes.
3. Run builddefs on the edited file as follows:

   ```
   builddefs CRAYdefs_i CRAYdefs
   ```

   CRAYdefs_i is the input file, and CRAYdefs is the output file. This utility produces two files: CRAYdefs, which is a definitions file without embedded keywords, and CRAYdefs_k, which is a keyword file.
4. Set the DEFINEDIR environment variable to the working directory that contains the CRAYdefs and CRAYdefs_k files. For the Korn and standard shells, set the variable as follows:

   ```
   DEFINEDIR=directoryname
   export DEFINEDIR
   ```

   For the C shell, set the variable as follows:

   ```
   setenv DEFINEDIR directoryname
   ```
5. Test the modified file by using the define(1) utility on selected terms.
6. Install the CRAYdefs_i, CRAYdefs, and CRAYdefs_k files in the /usr/lib/define directory.
7. If necessary, reset the DEFINEDIR environment variable. For the Korn and standard shells, reset the variable as follows:

   ```
   unset DEFINEDIR
   ```

For the C shell, reset the variable as follows:

```
unsetenv DEFINEDIR
```

Creating a Local Definitions File

The following procedure shows how to use `builddefs` to create a local definitions file for use by the `define`(1) utility. When multiple definitions files are in the definitions directory, the `define`(1) utility reads the files in alphabetical order; that is, it will search file `aaaa` before file `bbbb`.

Your local definitions files, installed in the default `define` directory `/usr/lib/define`, might be removed during the installation of a UNICOS revision or update if your site begins the installation process from a clean partition. In this case, back up your local definitions files, install the UNICOS revision or update, and then reinstall your local definitions files. Prepare an input file that contains embedded keywords according to the keywording rules contained in the DESCRIPTION section.

1. Run `builddefs` on the local file as follows:

   `builddefs` *sitedefs*`_i` *sitedefs*

   The *sitedefs*`_i` argument is the input file, and *sitedefs* is the output file. This command produces two files: *sitedefs*, which is a definitions file without embedded keywords, and *sitedefs*`_k`, which is a keyword file.

3. If you want to test how your local file works, set the `DEFINEDIR` environment variable to the working directory (*directoryname*) that contains the new formatted definition file. If you do not want to test the local file, skip to step 6. For the Korn and standard shells, set the variable as follows:

   ```
   DEFINEDIR=directoryname
   export DEFINEDIR
   ```

   For the C shell, set the variable as follows:

   `setenv DEFINEDIR` *directoryname*

4. Copy the `CRAYdefs` file by using the following command:

   `cp CRAYdefs CRAYdefs_k` *directoryname/*

5. Test the new file by using the `define`(1) utility on selected terms.
6. Install the *sitedefs* and *sitedefs*`_k` files in the `/usr/lib/define` directory. The `define`(1) utility reads the files in this directory and searches them in sequence for search string matches.
7. If necessary, reset the `DEFINEDIR` environment variable. For the Korn and standard shells, reset the variable as follows:

   ```
   unset DEFINEDIRR
   ```

   For the C shell, reset the variable as follows:

   ```
   unsetenv DEFINEDIR
   ```

FILES

`builddefs.cat`	`builddefs` message catalog
`builddefs.exp`	`builddefs` explain message catalog
`builddefs.msg`	`builddefs` message text file

SEE ALSO

`define`(1)

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`cal` – Prints calendar

SYNOPSIS

`cal` [[*month*] *year*]

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `cal` utility prints a calendar for the specified year. If a month is also specified, a calendar just for that month is printed. If neither is specified, a calendar for the present month is printed. *year* can be from 1 through 9999. *month* is a number from 1 through 12. The calendar produced is for England and the United States.

NOTES

The year is always considered to start in January.

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

EXIT STATUS

The following exit values are returned:

0 Successful completion.

>0 An error occurred.

EXAMPLES

An unusual calendar is printed for September 1752. That is the month 11 days were skipped to make up for lack of leap year adjustments. To see this calendar, type the following:

```
cal 9 1752
```

NAME

calendar – Reminder service

SYNOPSIS

```
calendar
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The `calendar` utility consults the `calendar` file in your 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 all users who have a file `calendar` in their login directory and sends them any positive results by `mail`(1). Usually, this is done daily by administrative facilities in the UNICOS operating system.

EXIT STATUS

The following exit values are returned:

0 Successful completion.

>0 An error occurred.

BUGS

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

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

EXAMPLES

In this example, suppose you have previously created a file named `calendar` that contains the following lines:

```
2/2    Report A due
2/3    Department meeting
2/4    Time card
2/4    Lunch with Lori
2/5    Report B due
```

If you enter the `calendar` command on February 3, you will receive the following output:

```
2/3    Department meeting
2/4    Time card
2/4    Lunch with Lori
```

FILES

`/usr/lib/calprog`	Program that figures out today's and tomorrow's dates
`/etc/passwd`	List of login directories
`/tmp/cal*`	Temporary files
`$HOME/calendar`	Personal calendar file

SEE ALSO

`mail`(1)

NAME

cat – Concatenates and prints files

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4
AT&T extensions (-s, -v, -t, and -e options)

DESCRIPTION

The cat utility reads files you specify on the command line in sequence and writes them to standard output.

The cat utility accepts the following options and operand:

-s Makes cat silent about nonexistent files.

-u Causes the output to be unbuffered. The default is buffered output.

-v Causes nonprinting characters (except for <tab>s, <newline>s, and <form-feed>s) to be printed visibly. ASCII control characters (octal 000–037) are printed as ^*n*, where *n* is the corresponding ASCII character in the range octal 100–137 (@, A, B, C, ..., X, Y, Z, [, \,], ^, and _); the DEL character (octal 0177) is printed ^?. Other nonprintable characters are printed as M-*x*, where *x* is the ASCII character specified by the low-order 7 bits.

When used with the -v option, the following options may be used:

-t Prints tabs as ^I's and form feeds to as ^L's when used with the -v option. If you do not specify the -v option, cat ignores this option.

-e Prints a $ character at the end of each line (before the <newline> character); you must use this option with the -v option.

If the -v option is not specified, the -t and -e options are ignored.

files If you do not specify an input file, or if you specify a hyphen (-), cat reads from standard input. Multiple occurrences of the argument - are accepted as file operands.

NOTES

If this utility is installed with a privilege assignment list (PAL), a user with one of the following active categories is allowed to perform the actions shown:

Active Category	Action
`system`, `secadm`	In a privileged administrator shell environment, shell-redirected I/O is not subject to file protections.
`sysadm`	Shell-redirected output is subject to security label restrictions.

If the `PRIV_SU` configuration option is enabled, shell-redirected I/O on behalf of the super user is not subject to file protections.

CAUTIONS

Command formats such as the following may cause the original data in *file1* to be lost:

```
$ cat file1 file2 >file1
```

EXIT STATUS

The `cat` utility exits with one of the following values:

0 All input files were output successfully.

>0 An error occurred.

EXAMPLES

Example 1: The following command line writes the contents of *file* to the standard output:

```
$ cat file
```

Example 2: The following command line concatenates the first two files and places the result in the third:

```
$ cat file1 file2 >file3
```

SEE ALSO

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

NAME

caterr – Processes message text files

SYNOPSIS

caterr [-c *catfile*] [-e] [-s[-P *cpp_opts*]] [-Y *x*,*pathname*] [*msgfile*]

IMPLEMENTATION

UNICOS systems

UNICOS/mk systems

IRIX systems

DESCRIPTION

A *message catalog* is a binary file that contains the run-time source of error messages output by UNICOS software products. A message catalog is produced from a message text file that contains messages (tagged with $msg tags) and message explanations (tagged with $nexp or $exp tags).

Before it can be accessed at run time, a message text file must be converted to a message catalog binary file by the caterr processor and the gencat(1) catalog generator.

The caterr utility converts the error message text source in *msgfile* into the format used as input to gencat(1), the error message catalog generation utility. If *msgfile* is not specified or if a dash (-) is specified, caterr reads from the standard input.

The -c option to the caterr utility calls gencat(1) after processing is complete. Using the -c option allows a catalog to be generated from a message text file in one step. It is recommended that you use caterr with the -c option. The gencat(1) utility exists as a separate utility to maintain compatibility with industry standards for message catalog processing. No advantage exists in calling gencat(1) separately. By default, caterr looks for gencat(1) in the /usr/bin/gencat file.

A single invocation of caterr can process either the messages or the explanations in the input files, but not both. The caterr utility processes the messages by default. Use the -e option to specify processing of the explanations.

The caterr utility calls the text formatting utility nroff(1) to process formatted explanations as part of its processing of the message text file. nroff(1) uses message macro definitions to format the explanation text. By default, on UNICOS and UNICOS/mk systems, caterr looks for nroff(1) in the /usr/bin/nroff file and for the message macros in the /usr/lib/tmac/tmac.sg file. On IRIX systems, caterr looks for nroff(1) in the /usr/bin/nroff file and for the message macros in the /usr/share/lib/tmac/tmac.sg file.

If no options are specified, caterr processes *msgfile* by using the tools in the default locations. The output, suitable for input to gencat(1), is sent to stdout.

The caterr utility accepts the following options and arguments:

-c *catfile* (Catalog) Calls gencat(1) to update or create a catalog with the information in the processed *msgfile*. If the -c option is used, caterr invokes gencat(1) to update the specified catalog by using the generated output. If *catfile* does not exist, it is created. Using the -c option makes it unnecessary to call gencat(1) separately; the message catalog is generated in one step.

-e (Explanations) Processes the explanations in *msgfile*. Without the -e option, caterr processes the messages in *msgfile*.

-s[-P *cpp_opts*]
(Symbolic names) Calls the C language preprocessor (cpp(1)) to preprocess symbolic message names into message numbers. The mapping of names to numbers must be specified in a header file name in the input file. On UNICOS and UNICOS/mk systems, caterr looks for cpp(1) first in the /usr/gen/lib/cpp directory. If it does not find it there, it looks in /lib/cpp. On IRIX systems, caterr looks for cpp(1) in the /lib/cpp directory.

Options can be passed to cpp by specifying the -P suboption to the -s option. Place the options to be passed to cpp within double quotation marks (" "). The entire string within the quotation marks is passed to cpp for execution. The -P suboption can be specified only if the -s option also is specified.

-Y *x*,*pathname*
Specifies the version of the nroff(1) and gencat(1) tools and of the tmac.sg message macros that caterr calls. If the -Y option is not specified, caterr calls the version of nroff(1) in /usr/bin/nroff, the version of gencat(1) in /usr/bin/gencat, and the version of the message macros in /usr/lib/tmac/tmac.sg (UNICOS and UNICOS/mk systems) or /usr/share/lib/tmac/tmac.sg (IRIX systems). If you need to specify alternative paths for all three tools that caterr calls, you can specify the -Y option up to three times in the same command line.

The -Y option takes two arguments: a path name and a key letter that specifies which software (nroff(1), gencat(1), or the message macros) is located at that path name. The key letter is specified first, followed by a comma (,), followed by the path name. The alternative tool path specified with *pathname* must be a full path.

The -Y option accepts the following key letters:

c Specifies that the path name following the comma is the path name for gencat(1).

m Specifies that the path name following the comma is the path name for the message macros.

n Specifies that the path name following the comma is the path name for nroff(1).

msgfile Specifies the name of the file containing the message text source to be processed.

EXAMPLES

Example 1: In the following example, `caterr` processes the messages in file `ldr.msg`. The output, sent to `stdout`, is suitable for input to `gencat`(1).

```
caterr ldr.msg
```

Example 2: In the following example, `caterr` invokes `gencat`(1) to update the messages in the `ldr.cat` catalog with the information in file `ldr.msg`.

```
caterr -c ldr.cat ldr.msg
```

Example 3: In the following example, `caterr` uses the message macros in the file `/usr/me/errmsg/tmac.sg` to produce a catalog of explanations suitable for processing by `gencat`(1). The input file is `ldr.msg`; the output is sent to `stdout`.

```
caterr -e -Y m,/usr/me/errmsg/tmac.sg ldr.msg
```

Example 4: In the following example, `caterr` uses the message macros in the current directory and invokes `gencat`(1) from `/bin/gencat` to update the explanation catalog `ldr.exp` with the information in `ldr.msg`.

```
caterr -e -c ldr.exp -Y m,tmac.sg -Y c,/bin/gencat ldr.msg
```

Example 5: In the following example, `caterr` calls `nroff` from `/usr/me/errmsg/nroff` and uses the message macros in the current directory. The input file is `ldr.msg`. Explanations suitable for processing by `gencat`(1) are output to `stdout`.

```
caterr -e -Y n,/usr/me/errmsg/nroff -Y m,tmac.sg ldr.msg
```

Example 6: In the following example, `caterr` calls alternative versions of all three tools. It uses the versions of `nroff`(1) and the message macros in the current directory, and it calls `gencat`(1) from `/bin/gencat`. Using these tools, the explanations in the `ldr.exp` file are updated with the information in the `ldr.msg` file.

```
caterr -e -c ldr.exp -Y c,/bin/gencat -Y m,tmac.sg -Y n,nroff ldr.msg
```

Example 7: In the following example, `caterr` invokes `gencat`(1) to update the messages in the `ldr.cat` catalog with the information in the `ldr.msg` file. The `caterr` utility calls `cpp`(1) to preprocess symbolic message names, and passes the `-M` option to `cpp`(1) for execution.

```
caterr -c ldr.cat -s -P "-M" ldr.msg
```

Example 8: In the following example, `caterr` invokes `gencat`(1) to update the `ldr.cat` catalog. Because no message text file name is specified, the input to `caterr` is read from the standard input.

```
caterr -c ldr.cat
```

SEE ALSO

`catxt`(1), `explain`(1), `gencat`(1), `whichcat`(1)

`catgetmsg`(3C), `catgets`(3C), `catmsgfmt`(3C), `catopen`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`nl_types`(5), `msg`(7D) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

Cray Message System Programmer's Guide, Cray Research publication SG–2121

NAME

catxt – Extracts message explanations from a message text file

SYNOPSIS

catxt [-n *outfile*] [-s[-P *cpp_opts*]] *infile*

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The catxt utility extracts the message explanations from a message text file to ready it for printing as an error message supplement document.

Typically, the input file *infile* is the message text file that resides in the developer's program library (PL). This file usually contains the text of error messages issued to users and the text of message explanations available to users through the explain(1) command. The error messages are associated with a $msg tag in the message text file. The explanations are associated with a $nexp or $exp tag in the message text file. Printed error message supplements contain the information associated with the $nexp tags.

The catxt utility accepts the following options and arguments:

-n *outfile* Specifies the *outfile* for extracted text. To produce an error message supplement, you must extract the text associated with $nexp tags from *infile*. The catxt utility performs this extraction. If the -n option is not specified, the output is sent to stdout.

-s[-P *cpp_opts*]
(Symbolic names) Calls the C language preprocessor (cpp(1)) to preprocess symbolic message names into message numbers. The mapping of names to numbers must be specified in an include file name in the input file. catxt looks for cpp(1) first in the /usr/gen/lib/cpp directory. If it does not find it there, it looks in the /lib/cpp file.

Options can be passed to cpp(1) by specifying the -P suboption to the -s option. Place the options to be passed to cpp(1) within double quotation marks (" "). The entire string within the quotation marks is passed to cpp(1) for execution. The -P suboption can be specified only if the -s option also is specified.

infile Specifies the message text file used as input to catxt.

NOTES

In addition to extracting explanations, catxt also changes the $nexp string into the .MS (message start) macro. This macro must be present for the output file to print in the proper format.

MESSAGES

The catxt utility issues the following messages:

`No explanation number processed yet.`
: This message is issued in conjunction with several of the warnings and errors that follow in this section. It indicates that the problem reported in the associated message occurred at the beginning of the file, before any explanations were processed successfully. Use this message to locate the problem and correct it.

`Last explanation number processed is '`*num*`'.`
: This message is issued in conjunction with several of the warnings and errors that follow in this section. It indicates that the problem reported in the associated message occurred after the specified explanation number. Use this message to locate the problem and correct it.

`***WARNING*** Encountered $nexp with no explanation number.`
: Every `$nexp` tag must be followed by an explanation number. If you receive this message, there is a `$nexp` tag without a number in the input file. Begin searching for this tag after the explanation number specified in the second line of the message (one of the first two messages in this section).

`***WARNING*** Encountered $nexp followed by character(s), a number is expected.`
: Each `$nexp` tag must be followed by an explanation number. If you receive this message, a `$nexp` tag exists that is followed by characters rather than numbers. Begin searching for this tag after the explanation number specified in the second line of the message (one of the first two messages in this section).

`***WARNING*** Incorrectly formed .ME line.`
: Each explanation must end with a `.ME` macro. The `.ME` macro must be on a line by itself. If you receive this message, a `.ME` macro in the input file is followed by other characters. Begin searching for this macro after the explanation number specified in the second line of the message (one of the first two messages in this section).

`***WARNING:nexp number` *n* `does not have an ending ".ME"***`
: The `catxt` utility verifies that there is a closing `.ME` macro for each occurrence of the `$nexp` tag. If it finds a `$nexp` tag without a `.ME` macro at the end of the explanation, it issues this warning message. If you receive this warning, add the `.ME` macro to message number *n* in the input file *infile* and rerun `catxt`.

`***ERROR*** Input file` *filename* `is not in the message text file format`
: The `catxt` utility verifies that key elements of the message text file format are present in the input file. If they are not present, it issues this error message. If you receive this error message, verify that the format of the file conforms to message text file guidelines. For a detailed description of the format of the message text file, see the *Cray Message System Programmer's Guide*, Cray Research publication SG–2121.

`***ERROR*** The message text file and the output file for nroffable`

`explanations are identical! Use a different output file.`

If you receive this message, the `catxt` utility has detected that the input file (message text file) and the output file you have specified have the same name. Proceeding with the utility under these circumstances will destroy the input file. To avoid the destruction of the input file, choose a different name for the output file.

SEE ALSO

`msg`(7D) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

Cray Message System Programmer's Guide, Cray Research publication SG–2121

NAME

cb – C program beautifier

SYNOPSIS

cb [-j] [-l *length*] [-s] [*files*]

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The cb utility 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 utility accepts the following options:

-j Causes split lines to be put back together.

-l *length* Causes cb to split lines that are longer than *length*. *length* must be between 10 and 120 characters. The default *length* is 120.

-s Changes the style of the code to conform to the style of Kernighan and Ritchie in *The C Programming Language*.

files Specifies files to read in.

CAUTIONS

The source code that is to be run under cb should be free of compilation errors.

Punctuation that is hidden in preprocessor statements causes indentation errors.

SEE ALSO

cc(1)

NAME

cd – Changes working directory

SYNOPSIS

cd [*directory*]

cd -

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The cd utility changes your working directory to *directory*. If you omit *directory*, the value of the shell variable 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*.

NOTES

The cd utility is a built-in utility to the standard shell (sh(1)). An executable version of this utility is available in /usr/bin/cd.

EXIT STATUS

The cd utility exits with one of the following values:

0 The directory was successfully changed.

>0 An error occurred.

SEE ALSO

pwd(1), sh(1)

chdir(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

cdc – Changes the delta commentary of an SCCS delta

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The cdc command changes the delta commentary of the *SID* specified by the -r option of each named Source Code Control System (SCCS) file. A *delta commentary* is the modification request (MR) and comment information normally specified by using the delta(1) command (-m and -y options).

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 the WARNINGS section); each line of the standard input is taken to be the name of an SCCS file to be processed.

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

All described option-arguments apply independently to each named file:

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

-m[*mrlist*] If the SCCS file has the v flag set (see admin(1)), a list of (MR) numbers to be added and/or deleted in the delta commentary of the *SID* specified by the -r option may be supplied. 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). To delete an MR, precede the MR number with the ! character (see the EXAMPLES section). 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 option).

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

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

`-y`[*comment*] Arbitrary text used to replace the *comment*(s) already existing for the delta specified by the `-r` option.

files Files to be changed.

The previous comments are kept and preceded by a comment line stating that they were changed. A null *comment* has no effect.

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

Permissions needed to modify the SCCS file 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.

WARNINGS

If SCCS file names are supplied to the `cdc` command through the standard input (`-` on the command line), the `-m` and `-y` options must also be used.

MESSAGES

Error messages from SCCS are printed. Use `help`(1) for explanations.

EXAMPLES

The following example 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 -m"bl78-12345 !bl77-54321 bl79-00001" -ytrouble s.file
```

The following example does the same thing:

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

FILES

`x.`*file* See `delta`(1)

`z.`*file* See `delta`(1)

SEE ALSO

admin(1), comb(1), delta(1), get(1), help(1), prs(1), rmdel(1), sact(1), sccsdiff(1), unget(1), val(1), vc(1), what(1)

sccsfile(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

cflow – Generates C-language flowgraph

SYNOPSIS

cflow [-d *num*] [-D *name*[=*def*]]... [-i *incl*] [-I *dir*]... [-r] [-U *dir*]... *file*...

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The cflow utility analyzes a collection of C, yacc, lex, assembler, and object files and builds a graph charting the external function references. Files suffixed with .y, .l, and .c are processed by yacc, lex, and the C compiler as appropriate. The results of the preprocessed files, and files suffixed with .i, are then run through the first pass of lint. Files suffixed with .s are assembled. Assembled files, and files suffixed with .o, have information extracted from their symbol tables. The results are collected and turned into a graph of external references that is written on the standard output.

Each line of output begins with a reference number, followed by a suitable number of tabs indicating the level, then the name of the global symbol followed by a colon and its definition. Normally only function names that do not begin with an underscore are listed (see the -i options). 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. Definitions extracted from object files indicate the file name and location counter under which the symbol appeared (for example, *text*). Leading underscores in C-style external names are deleted.

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

As an example, suppose the following code is in file.c:

```
int   i;

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

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

The command

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

produces the output

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

When the nesting level becomes too deep, the output of `cflow` can be piped to the `pr`(1) utility, using the `-e` option, to compress the tab expansion to something less than every eight spaces.

In addition to the `-D`, `-I`, and `-U` options (which are interpreted just as they are by `cc`(1)), 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).

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

NOTES

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

DIAGNOSTICS

Complains about multiple definitions and only believes the first.

SEE ALSO

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

NAME

`chacid` – Changes account ID of disk files

SYNOPSIS

`chacid` [`-v`] [`-S`] [`-h`] *files*
`chacid` [`-s` *id*] [`-v`] [`-h`] *files*

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `chacid` utility allows users to set the account ID of a disk file (or symbolic link). The ID must be one of the user's valid accounts. Only the owner of a file may change that file's account ID. An appropriately authorized user may set the account ID of a file that is owned by another user and may specify any account ID value.

When used without options, `chacid` displays the current account ID. Options `-s` and `-S` are mutually exclusive.

The `chacid` utility accepts the following options and arguments:

`-s` *id* Sets the account ID of the specified *files* to *id*.

`-v` Operates in verbose mode.

`-S` Sets the account ID of files to the current user's default account ID.

`-h` When this option is specified and the file is a symbolic link, the requested operation (display the current value or change the account ID) is done on the link; that is, it does not follow the link to the destination file. If this option is not specified, the link is followed and the operation is done on the destination file.

files Specifies the file (or files) whose account ID will be set.

NOTES

If this utility is installed with a privilege assignment list (PAL), a user who is assigned the following privilege text upon execution of this command is allowed to perform the actions shown:

Privilege Text	Action
`anyown`	Allowed to change the account ID of a file that is owned by another user and to specify any account ID value.
`own`	Allowed to change the account ID of a file that is owned by another user.
`any`	Allowed to specify any account ID value.

If this utility is installed with a PAL, a user with one of the following active categories is allowed to perform the actions shown:

Active Category	Action
`system`, `secadm`	Allowed to change the account ID of any file to any value.
`sysadm`	Allowed to change the account ID of any file to any value, subject to security label restrictions on the file's path. Allowed to specify any account ID value.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to change the account ID of any file to any value.

EXAMPLES

To determine the current account ID of the file `notes`, enter the following:

```
chacid notes
```

To change the account ID of the file `notes` to `proj1`, enter the following:

```
chacid -s proj1 notes
```

FILES

`/etc/acid`	Account ID information file that contains account names and account IDs
`/etc/udb`	User validation file that contains user control limits

SEE ALSO

`newacct`(1), `privtext`(1)

`chacid`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

UNICOS Resource Administration, Cray Research publication SG–2302

NAME

checknr – Checks nroff and troff input files; reports possible errors

SYNOPSIS

checknr [-fs] [-a .*x1* .*y1* .*x2* .*y2* ... .*xn* .*yn*] [-c .*x1* .*x2* .*x3* ... .*xn*] [*filename* ...]

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

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

- Font changes using \f*x* ... \fP.
- Size changes using \s*x* ... \s0.
- Macros that come in open ... close forms (for example, the .TS and .TE macros), which must always come in pairs.

The checknr utility knows about the ms(7D) and me(7D) macro packages.

The checknr utility is intended to be used on documents that are prepared with checknr in mind. It expects a certain document writing style for \f and \s commands, in that each \f*x* must be terminated with \fP and each \s*x* must be terminated with \s0. While it will work to go directly into the next font or explicitly specify the original font or point size, and many existing documents actually do this, such a practice will produce complaints from checknr. Since it is probably better to use the \fP and \s0 forms anyway, you should think of this as a contribution to your document preparation style.

The checknr utility accepts the following options:

-f	Ignores \f font changes.
-s	Ignores \s size changes.
-a .*x1* .*y1* ...	Adds pairs of macros to the list. The pairs of macros are assumed to be those (such as .DS and .DE) that should be checked for balance. The -a option must be followed by groups of six characters, each group defining a pair of macros. The six characters are a period, the first macro name, another period, and the second macro name. For example, to define the pair .BS and .ES, use -a.BS.ES.
-c .*x1* ...	Defines commands that checknr would otherwise complain about as undefined.
filename	nroff file to be checked.

NOTES

There is no way to define a 1-character macro name using the `-a` option.

SEE ALSO

`eqn`(1), `nroff`(1), `troff`(1)

`me`(7D), `ms`(7D) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

chgrp – Changes the group ownership of a file

SYNOPSIS

chgrp [-R] [-h] *group file* ...

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4
AT&T extensions (-h option)

DESCRIPTION

The chgrp utility changes the group ID of each *file* to *group*.

The chgrp utility accepts the following options and operands:

-R Recursively changes file group IDs. When symbolic links are encountered, the group of the target is changed, but no recursion takes place.

-h If the file is a symbolic link, changes the group of the symbolic link. Without this option, the group of the file or directory referenced by the symbolic link is changed.

group May be either a decimal group ID or a group name found in the group file.

file ... Files or directories to be changed. For each *file* operand that specifies a directory, chgrp changes the group ID of the directory and all files in the file hierarchy below it. When symbolic links are encountered, they are not traversed.

Only an appropriately authorized user may change the group of a file that is owned by another user. Unless users are appropriately authorized, they must be a member of the specified group to change the group of a file.

Unless the user is appropriately authorized, chgrp clears the set-user-ID and set-group-ID file mode bits.

NOTES

When the path name supplied to the chgrp utility specifies a multilevel symbolic link (the name of a multilevel directory), the group is changed only on the root of the multilevel directory tree. While this affects all subsequently created labeled subdirectories, it does not affect labeled subdirectories currently in existence. To ensure that the group of all labeled subdirectories change, the user must manually change the group on each labeled subdirectory found in root of the multilevel directory.

If this utility is installed with a privilege assignment list (PAL), a user with one of the following active categories is allowed to perform the actions shown:

Active Category	Action
`system`, `secadm`	Allowed to change the group of any file to any value. The set-user-ID and set-group-ID file mode bits are not cleared.
`sysadm`	Allowed to change the group of any file to any value. The set-user-ID and set-group-ID file mode bits are not cleared. Shell-redirected I/O is subject to security label restrictions.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to change the group of any file to any value. If the user is the super user or has the `suidgid` permission, the set-user-ID and set-group-ID file mode bits are not cleared.

EXIT STATUS

The `chgrp` utility exits with one of the following values:

0 All requested changes were made.

>0 An error occurred.

EXAMPLES

The following example changes the group to `dev` for the `example.c` file:

```
chgrp dev example.c
```

FILES

`/etc/udb`	User validation file that contains user control limits
`/etc/group`	Group file that contains group names and group IDs

SEE ALSO

`chmod`(1), `chown`(1), `id`(1)

`chown`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`group`(5), `passwd`(5), `udb`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

General UNICOS System Administration, Cray Research publication SG–2301

NAME

chkey – Changes your encryption key

SYNOPSIS

```
chkey
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The chkey command prompts users for their secure RPC password, and uses it to encrypt a new encryption key for the user to be stored in the publickey(5) database.

SEE ALSO

keylogin(1)

publickey(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

keyserv(8), newkey(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

chkpnt – Checkpoints a process, multitask group, or job

SYNOPSIS

```
chkpnt -j id [-k] [-v] [-f file]
chkpnt -p id [-k] [-v] [-f file]
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The chkpnt utility creates a restart file containing all of the necessary state information to restart all of the processes selected by the *id* option. The target *id* must belong to the current user, unless the current user is the super user.

The chkpnt utility accepts the following options:

`-j` *id*
`-p` *id* Indicates whether the *id* specified is a job (or interactive session) ID (`-j`) or a process ID (`-p`).

`-k` Indicates that the specified *id* (job or process) will be killed if the checkpoint is successful.

`-v` Causes additional informational messages to be printed.

`-f` *file* Indicates the path name to be used for the restart file. If the user does not specify a path name, a default path name will be created by appending *id* to the name `pid` or `jib`.

NOTES

If this utility is installed with a privilege assignment list (PAL), a user with one of the following active categories is allowed to perform the actions shown:

Active Category	Action
system, secadm, sysadm	Allowed to checkpoint any process or job.

If the PRIV_SU configuration option is enabled, the super user is allowed to checkpoint any process or job.

The following restrictions apply to jobs and processes that are to be checkpointed:

- Only an appropriately authorized user may checkpoint a process or job that is owned by another user.
- The active security label of the user must equal the active security label of every affected process.
- Processes with open pipes may be checkpointed and restarted successfully if the following two conditions are met:
  - All openings of the pipe file must be contained within the process collection being checkpointed.

  - All I/O operations on the pipe must be atomic with respect to the chkpnt system call. This condition is a limit on the size of an I/O operation: either PIPE_BUF bytes, or (v_maxpipe * 4096) bytes. PIPE_BUF is found in the sys/param.h file. v_maxpipe is a member of the var structure in the sys/var.h file.

- All files that a process was using when it was checkpointed must be present when the process is restarted.
- Processes using shared memory segments (CRAY T90 series systems only) cannot be checkpointed or restarted.
- Processes using online tapes cannot be checkpointed or restarted.

EXAMPLES

The following example illustrates how to use the chkpnt utility to checkpoint a process.

The user is running a.out in the background and enters a chkpnt utility to checkpoint the process. The -p option specifies the PID and the -k option kills the process (pid.23634). The user then issues an ls command to list information about pid.23634; the capital R at the beginning of the long listing indicates a restart file has been created.

```
unicos$ a.out &
23634

unicos$ chkpnt  -p  23634  -k

unicos$ ls  -l  pid.23634
Rr--------  1  (id) (group) (size) (date) pid.23634
```

Later, the user enters a restart utility to recover the process. The ps listing confirms the process has been reactivated.

```
unicos$ restart  pid.23634

unicos$ ps
    PID    TTY  TIME COMMAND
  23634   p031  0:13 a.out
  23510   p031  0:00 sh
  23758   p031  0:00 ps
```

SEE ALSO

chkpnt_util(1), chkptint(1), restart(1)

chkpnt(2), restart(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

General UNICOS System Administration, Cray Research publication SG–2301

NAME

chkpnt_util – Verifies restartability of restart files

SYNOPSIS

chkpnt_util [-f *filch*] [-n] [-o *name*] [-v] *restart-file*

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The chkpnt_util utility verifies a restart file in the same manner that the kernel would verify the file during an attempted restart. The utility issues the same cryptic messages as the kernel, but offers a verbose mode that locates the problem. The utility also enables you to find the name of a restart-referenced nc1 or sfs file.

The chkpnt_util accepts the following options and operand:

-f *filch* Associates path names to the files identified by specified vnodes. Items in the *filch* string are separated by commas. Each item is a pair separated by a colon. The first in the pair is the former vnode pointer, used to identify the node within the restart file. The second in the pair is the new file name.

-n Prints the name of every file referenced in the restart file. This option uses a lot of execution time.

-o *name* Writes a new restart file with the specified name.

-v Produces verbose output.

restart-file Specifies the restart file.

NOTES

Only an appropriately authorized user may provide super-user authority to write a restart file. Similarly, authority is required for the openi(2) system call.

EXAMPLES

The following example shows the output produced when you specify `chkpnt_util` with the `-n` and `-v` options:

```
# chkpnt_util -n -v  ofile
chkpnt_util: restart header
chkpnt_util: session header, sid = 63
chkpnt_util: process executing sh
chkpnt_util:   process pcomm structure, ppid 3392
chkpnt_util:     task proc structure, pid 3395
chkpnt_util:   process ucomm structure, uc_magic 020001600407
chkpnt_util:       vnode for executable text
chkpnt_util:         vnode structure from vp = 01251570
chkpnt_util:         -- inum 5305 on minor 48
chkpnt_util:         -- file name "/bin/sh"
chkpnt_util:       vnode for current directory
chkpnt_util:         vnode structure from vp = 01526754
chkpnt_util:         -- inum 106 on minor 36
chkpnt_util:         -- file name "/tmp/jtmp.000057a"
chkpnt_util:       open file number 0
chkpnt_util:         file structure, from fp = 01204253
chkpnt_util:         vnode structure from vp = 01254320
chkpnt_util:         -- inum 1530 on minor 48
chkpnt_util:         -- file name "/dev/tty"
....
chkpnt_util:       open file number 31
chkpnt_util:         file structure, from fp = 01204551
chkpnt_util:         vnode structure from vp = 01536764
chkpnt_util:         -- inum 1061919 on minor 27
chkpnt_util:         -- file name "/frost/u4/rig/.sh_history"
chkpnt_util: file has been modified
chkpnt_util: **** condition prevents restart
```

To obtain detailed information about specific files, select the vnodes and associated new file names from the verbose output and enter the following command:

```
chkpnt_util -f 01526754:/tmp/jtmp.000057b, 1536764:/dev/null
```

SEE ALSO

`chkpnt`(1), `restart`(1)

`chkpnt`(2), `openi`(2), `restart`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

General UNICOS System Administration, Cray Research publication SG–2301

NAME

chkptint – Registers current session to be checkpointed upon shutdown and/or periodically

SYNOPSIS

chkptint -s *sec*

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The chkptint utility notifies the Unified Resource Manager (URM) that a checkpoint of the current session should be taken every *sec* seconds and/or at shutdown, depending on the URM configuration. This utility should be used in conjunction with the rmgr(1) utility View Restart subcommand, which allows you to see if you have any checkpointed interactive sessions.

The purpose of the URM checkpoint feature is to improve user recoverability by allowing you to periodically checkpoint your active session to protect against lost work due to an uncontrolled system shutdown and to request that an interactive session be checkpointed as part of a controlled system shutdown.

The chkptint utility requires the following options:

-s *sec* A positive integer *sec* indicates that a checkpoint of the current session should be done at shutdown and the frequency at which periodic checkpoints should be taken. Zero *sec* indicates that checkpointing should not be done for the current session.

The units of the -s option are either wall-clock or CPU seconds; the unit selected is specified in a URM configuration parameter.

NOTES

The URM checkpoint facility is disabled by default; checkpointing of sessions is not performed unless this facility is enabled in URM. A URM configuration parameter, chkpt_switch, can be set to select whether checkpointing is done at shutdown only or both periodically and at shutdown. If URM is running with checkpoint at shutdown only, a nonzero value for *sec* also indicates that checkpointing is desired at shutdown. For information on enabling checkpointing in URM, see *UNICOS Resource Administration*, Cray Research publication SG–2302.

The checkpointing of sessions through chkptint follows all the current limitations of chkpnt(2). For a list of restrictions, see chkpnt(2).

If you restart a saved session, this session does not show up in the output of the finger(1B) or who(1) utility. The reason is that init(8) is the process that updates the tables used by these commands and restarting a previously running session does not notify the init(8) process.

MESSAGES

```
chkptint:  Could not obtain the current session ID
using the getjtab system call.

chkptint:  The current session ID returned by the getjtab system call
does not match the job table data.

chkptint:  The current user ID returned by the getjtab system call
does not match the job table data.
```

EXAMPLES

To set a checkpoint interval frequency of 30 minutes:

```
$ chkptint -s 1800
```

To automatically turn off checkpointing:

```
$ chkptint -s 0
```

SEE ALSO

chkpnt(1), finger(1B), restart(1), rmgr(1), who(1)

chkpnt(2), getjtab(2), restart(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

init(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

UNICOS Resource Administration, Cray Research publication SG–2302

NAME

chmod – Changes mode of files or directories

SYNOPSIS

chmod [-R] *mode files*

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The chmod utility changes the mode of the specified files (or directories) according to *mode*.

The chmod utility accepts the following option and operands:

-R Recursively descends through directory arguments, setting the mode for each file. When symbolic links are encountered, the mode of the target is changed, but no recursion occurs.

mode Specifies the permissions and other attributes of a file. The mode may be absolute or symbolic.

files Files to be changed.

Absolute changes to modes are stated with octal numbers:

chmod *nnnn files*

n is a number from 0 through 7.

An absolute mode is given as an octal number constructed from the OR of the following modes:

Code	Description
4000	Sets user ID on execution.
20#0	Sets group ID on execution if # is 7, 5, 3, or 1. Enables mandatory locking if # is 6, 4, 2, or 0. (Secure sites: see secure site information that follows.) If the file is a directory, this bit is ignored; you may set or clear it using only the symbolic mode.
1000	Sticky bit (applicable only to directories). See chmod(2) for more information.
0400	Sets read permission for owner.
0200	Sets write permission for owner.
0100	Sets execute (search in directory) permission for owner.
0070	Sets read, write, and execute (search in directory) permission for group.

`0007` Sets read, write, and execute (search in directory) permission for others.

Symbolic changes are stated with mnemonic characters:

`chmod [`*who*`] `*operator*` [`*permissions(s)*`] `*files*

who is one or more characters that corresponds to user, group, or other (`u`, `g`, or `o`, respectively); *operator* is +, –, or =, signifying the assignment of permissions; and *permission(s)* is one or more characters that correspond to type of permission.

Symbolic changes are stated using letters that correspond both to access classes and to the individual permissions themselves. Permissions to a file may vary, depending on your user identification number (UID) or group identification number (GID). Permissions are described in three sequences, each having three characters:

User	Group	Other
rwx	rwx	rwx

The preceding lines (meaning that user, group, and others all have reading, writing, and execution permission to a given file) demonstrate two categories for granting permissions: the access class and the permissions themselves.

A command making *file* readable and writable by the group using the symbolic method would appear as follows:

`chmod g+rw `*file*

who can be stated as one or more of the following letters:

`u` User's permissions

`g` Group's permissions

`o` Others permissions

`a` All (equivalent to specifying `ugo`).

If you omit *who*, `chmod` will use the file creation mask (see `umask`) to determine whose permissions are to be changed.

operator is one of `+`, `-`, or `=`, signifying how permissions are to be changed:

`+` Adds permissions.

`-` Removes permissions.

`=` Assigns permissions absolutely.

Unlike other symbolic operations, = has an absolute effect in that it resets all other bits. Omitting *permission(s)* is useful only with = to remove all permissions.

permission(s) is any compatible combination of the following letters:

r Reading permission

w Writing permission

x Execution permission

X Search/execute permission if the file is a directory or if current mode has at least one of the execute bits set.

s User or group set-ID is turned on

l Mandatory locking occurs during access (if any execution bits are set, group set-ID is turned on).

t Sets the sticky bit (see chmod(2) for more information)

Multiple symbolic modes separated by commas can be specified, although no spaces may intervene between these modes. Operations are performed in the order given. Multiple symbolic letters following a single operator cause the corresponding operations to be performed simultaneously. The letter s is meaningful only with u or g; thus, if a is used with s, only u and g are affected. t works only with u.

Mandatory file and record locking refers to a file's ability to have its read or write permissions locked while a program is accessing that file. You cannot permit group execution and enable a file to be locked on execution at the same time. You can turn on the set-group-ID and enable a file to be locked on execution at the same time.

The following examples are, therefore, illegal usages and will elicit error messages:

```
chmod g+x,+l file

chmod g+s,+l file
```

Only an appropriately authorized user may change the mode of a file that he or she does not own. Only an appropriately authorized user may alter the t permission (mode 1000). Otherwise, chmod clears the t permission, but does not return an error. To enable a file's set-group-ID mode bit, you must belong to the file's owning group and group execution must be allowed.

NOTES

The chmod utility permits you to produce useless modes if they are not illegal (for example, making a text file executable).

When the path name supplied to the chmod utility specifies a multilevel symbolic link (the name of a multilevel directory), the mode is changed only on the root of the multilevel directory tree. While this affects all subsequently created labeled subdirectories, it does not affect labeled subdirectories currently in existence. To ensure that the mode of all labeled subdirectories change, the user must manually change the mode on each labeled subdirectory found in root of the multilevel directory.

If this utility is installed with a privilege assignment list (PAL), a user with one of the following active categories is allowed to perform the actions shown:

Active Category	Action
`system`, `secadm`	Allowed to change the mode of any file. The set-user-ID and set-group-ID file mode bits are not cleared.
`sysadm`	Allowed to change the mode of any file. The set-user-ID and set-group-ID file mode bits are not cleared. Shell-redirected I/O is subject to security label restrictions.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to change the mode of any file. If the user is the super user or has the `suidgid` permission, the set-user-ID and set-group-ID file mode bits are not cleared.

EXIT STATUS

The `chmod` utility exits with one of the following values:

0 All requested changes were made.

>0 An error occurred.

EXAMPLES

Example 1: The following examples show how to deny execution permission to all. The absolute (octal) example permits only reading permissions:

```
chmod a-x file
chmod 444 file
```

Example 2: The following examples make a file readable and writable by the group and others:

```
chmod go+rw file
chmod 066 file
```

Example 3: The following example locks a file during access:

```
chmod +l file
```

Example 4: The following examples enable all to read, write, and execute the file; and they turn on the set-group-ID:

```
chmod +rwx,g+s file
chmod 2777 file
```

SEE ALSO

`ls`(1), `umask`(1)

`chmod`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

chown – Changes owner of files or directories

SYNOPSIS

chown [-h] [-R] *owner*[:*group*] *files...*

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4
AT&T extensions (-h option)

DESCRIPTION

The chown utility changes the owner of each *file* to *owner*. The optional *group* operand changes the group.

The chown utility accepts the following options and operands:

-h If the file is a symbolic link, changes the owner of the symbolic link. Without this option, the owner of the file or directory referenced by the symbolic link is changed.

-R Recursively changes file user IDs, and if the *group* operand is specified, group IDs. When symbolic links are encountered, the owner of the target is changed, but no recursion takes place. For each *file* operand that specifies a directory, chown changes the user and group ID of the directory and all files in the file hierarchy below it.

owner[:*group*]
owner may be either a decimal user ID or a login name found in the password file. *group* may be either a decimal group ID or a group name found in the group file.

files... Files or directories to be changed.

Only an appropriately authorized user may change the owner of a file.

Unless the user is appropriately authorized, chown clears the set-user-ID and set-group-ID file mode bits.

NOTES

When the path name supplied to the chown utility specifies a multilevel symbolic link (the name of a multilevel directory), the owner is changed only on the root of the multilevel directory tree. While this affects all subsequently created labeled subdirectories, it does not affect labeled subdirectories currently in existence. To ensure that the owner of all labeled subdirectories change, the user must manually change the owner on each labeled subdirectory found in root of the multilevel directory.

If this utility is installed with a privilege assignment list (PAL), a user with one of the following active categories is allowed to perform the actions shown:

Active Category	Action
system, secadm	Allowed to change the owner of any file. The set-user-ID and set-group-ID file mode bits are not cleared.
sysadm	Allowed to change the owner of any file. The set-user-ID and set-group-ID file mode bits are not cleared. Shell-redirected I/O is subject to security label restrictions.

If the PRIV_SU configuration option is enabled, the super user is allowed to change the owner of any file. If the user is the super user or has the suidgid permission, the set-user-ID and set-group-ID file mode bits are not cleared. A user with the chown permbit is allowed to change the ownership of his or her files.

EXIT STATUS

The chown utility exits with one of the following values:

0 All requested changes were made.

>0 An error occurred.

FILES

/etc/udb	User validation file that contains user control limits
/etc/passwd	Password file that contains login names and user IDs
/etc/group	Group file that contains group names and group IDs

SEE ALSO

chgrp(1), chmod(1)

chown(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

group(5), passwd(5), udb(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

udbgen(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

General UNICOS System Administration, Cray Research publication SG–2301

NAME

chsh – Changes default login shell

SYNOPSIS

/usr/ucb/chsh *name* [*shell*]

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The chsh utility is similar to passwd(1) except that it changes the login shell field of your password file rather than the password entry.

The *name* argument is your login name.

The *shell* argument is compared with the list of legal user shells obtained from the getusershell(3C) library routine. Possible shells, depending on site configuration, may be one of the following:

```
/bin/sh
/bin/csh
```

The full path name is not necessary. Only an appropriately authorized user can specify a shell that is not in the allowed list of user shells or change the shell for another user. If you do not specify a shell, the login shell defaults to /bin/sh.

NOTES

If this utility is installed with a privilege assignment list (PAL), a user who is assigned the following privilege text upon execution of this command is allowed to perform the action shown:

Privilege Text	Action
chgany	Allowed to change the shell of any user to any value.

If this utility is installed with a PAL, a user with one of the following active categories is allowed to perform the action shown:

Active Category	Action
system, secadm, sysadm	Allowed to change the shell of any user to any value.

If the PRIV_SU configuration option is enabled, the super user is allowed to change the shell of any user to any value.

EXAMPLES

```
chsh bill /bin/csh
chsh jim /bin/sh
```

SEE ALSO

csh(1), passwd(1), privtext(1), sh(1)

getusershell(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

shells(5), udb(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

General UNICOS System Administration, Cray Research publication SG–2301

NAME

cksum – Writes file checksums and sizes

SYNOPSIS

cksum [*files...*]

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The cksum utility calculates and writes to standard output a cyclic redundancy check (CRC) for each input file, and it also writes to standard output the number of octets in each file. The CRC used is based on the polynomial used for CRC error checking in the networking standard ISO 8802-3. The standard input is used only if no *file* operands are specified.

For each file processed successfully, the cksum utility writes in the following format:

"<*checksum*> <*# of octets*> <*path name*>\n"

If you omit the *file* operand, the path name and its leading space is omitted.

The cksum utility accepts the following operand:

files... A path name of a file (any type) to be checked. If no *file* operands are specified, the standard input is used.

NOTES

The generating polynomial defines the encoding, as follows:

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

Mathematically, the CRC value that corresponds to a given file is defined by the following procedure:

1. The *n* bits to be evaluated are considered to be the coefficients of a mod 2 polynomial *M(x)* of degree *n*-1. These *n* bits are the bits from the file, with the most-significant bit being the most significant bit of the first octet of the file and the last bit being the least-significant bit of the last octet, padded with zero bits (if necessary) to achieve an integral number of octets, followed by one or more octets representing the length of the file as a binary value, least-significant octet first. The smallest number of octets that can represent this integer is used.
2. *M(x)* is multiplied by x^{32} (for example, shifted left 32 bits) and divided by *G(x)* using mod 2 division, producing a remainder *R(x)* of degree ≤ 31.

3. The coefficients of *R(x)* are considered to be a 32-bit sequence.
4. The bit sequence is complemented, and the result is the CRC.

If this utility is installed with a privilege assignment list (PAL), a user with one of the following active categories is allowed to perform the actions shown:

Active Category	Action
`system`, `secadm`	Allowed to generate a checksum of any file. In a privileged administrator shell environment, shell-redirected I/O is not subject to file protections.
`sysadm`	Allowed to generate a checksum of any file subject to security label restrictions. Shell-redirected I/O is subject to security label restrictions.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to generate a checksum of any file. Shell-redirected I/O on behalf of the super user is not subject to file protections.

EXAMPLES

Generate a checksum for each of the files `a.out`, `file.c`, and `obj.o`:

```
cksum a.out file.c obj.o
```

EXIT STATUS

The `cksum` utility exits with one of the following values:

0 All files were processed successfully.

>0 An error occurred.

SEE ALSO

ISO 8802-3: 1989, *Information processing systems – Local area networks – Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specification.*

A Tutorial on CRC Computations, Ramabadran and Gaitonde, *IEEE Micro*, August, 1988, p. 62.

Computation of Cyclic Redundancy Checks Via Table Lookup, Sarwate, Dilip V, *Communications of the ACM*, August, 1988, p. 1011.

NAME

`clear` – Clears terminal screen

SYNOPSIS

`/usr/ucb/clear`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

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

FILES

`/usr/lib/terminfo`	Terminal capability database

NOTES

Use the `tput clear` command, because the `clear` command might be removed in a future UNICOS 9.0 release.

SEE ALSO

`tput`(1)

NAME

cmp – Compares two files

SYNOPSIS

```
cmp [-l] file1 file2
cmp -s file1 file2
cmp -w file1 file2
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4
CRI extension (-w option)

DESCRIPTION

The cmp utility compares *file1* and *file2*. 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.

The cmp utility accepts the following options:

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

-s Displays nothing for files that differ; returns exit status only.

-w Word mode. Displays differences between 64-bit words. The octal offset of the word from the beginning of the file, an octal representation of the words, and an ASCII representation of the words are displayed for each word that differs.

file1
file2 The path names of the two files to be compared. If - is specified, the standard input is used.

NOTES

If this utility is installed with a privilege assignment list (PAL), a user with one of the following active categories is allowed to perform the actions shown:

Active Category	Action
system, secadm	In a privileged administrator shell environment, shell-redirected I/O is not subject to file protections.
sysadm	Shell-redirected output is subject to security label restrictions.

If the PRIV_SU configuration option is enabled, shell-redirected I/O on behalf of the super user is not subject to file protections.

EXIT STATUS

The cmp utility exits with one of the following values:

0 Files are identical.

1 Files are different.

2 An error occurred.

SEE ALSO

comm(1), diff(1)

NAME

col – Filters reverse line feeds

SYNOPSIS

col [-b] [-f] [-p] [-x]

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The col utility filters reverse line feeds, reading from the standard input and writing to the standard output. It performs the line overlays implied by reverse line feeds (ASCII code ESC-7), and by forward and reverse half-line feeds (ESC-9 and ESC-8). col is particularly useful for filtering multicolumn output made with the .rt command of nroff(1) and output resulting from use of the tbl(1) preprocessor.

The col utility accepts the following options:

-b Indicates that the output device is incapable of backspacing.

-f Enables half-line vertical motion.

-p Enables output of escape sequences as regular characters.

-x Suppresses output of tabs rather than white space.

If the -b option is given, col assumes that the output device in use is incapable of backspacing. In this case, if 2 or more characters are to appear in the same place, only the last one read will be output.

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

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

On input, the only control characters accepted are <space>, <backspace>, <tab>, <carriage-return>, <newline>, SI, SO, <vertical tab> (\013), and ESC followed by 7, 8, or 9. The <vertical tab> character is an alternative form of full reverse line feed, included for compatibility with some earlier programs of this type. All other nonprinting characters are ignored.

Normally, `col` ignores any unknown escape sequences found in its input; the `-p` option may be used to cause `col` to output these sequences as regular characters, subject to overprinting from reverse line motions. The use of this option is highly discouraged unless you are fully aware of the textual position of the escape sequences.

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

NOTES

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

EXIT STATUS

The following exit values are returned:

0 successful completion.

>0 An error occurred.

BUGS

The following are known `col` bugs:

- The `col` utility cannot back up more than 128 lines.
- Up to 800 characters, including backspaces, are allowed on a line.
- Local vertical motions that would result in backing up over the first line of the document are ignored. As a result, the first line must not have any superscripts.

SEE ALSO

`nroff`(1), `tbl`(1)

NAME

comb – Combines SCCS deltas

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The comb command generates a shell procedure (see sh(1)) which, when run, reconstructs the given Source Code Control System (SCCS) files.

The reconstructed files should be smaller than the original files. The options may be specified in any order, but all options 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 dash (-) is given as a name, 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 options 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 options are as follows. Each is explained as though only one named file is to be processed, but the effects of any option apply independently to each named 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 Causes the reconstructed file to be accessed at the release of the delta to be created for each get -e generated. Otherwise, the reconstructed file would be accessed at the most recent ancestor. Use of the -o option may decrease the size of the reconstructed SCCS file. It may also alter the shape of the delta tree of the original file.

-p*sid* Provides the SCCS identification string (*sid*) of the oldest delta to be preserved. All older deltas are discarded in the reconstructed file.

-s Causes comb to generate a shell procedure which, when run, produces a report. This report gives for each file: the file name, size (in blocks) after combining, original size (also in blocks), and percentage change computed by the following:

100*(*original–combined*)/*original*

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

file1 file2 Files to be compared.

MESSAGES

Error messages from SCCS are printed. Use `help`(1) for explanations.

BUGS

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

The `comb` command may produce shell scripts that contain `admin` commands which have an argument to the `-fi` option containing white space. This will cause the shell script to fail. To allow the shell script to work, place single quotes around the argument to the `-fi` option.

FILES

`s.COMB`	Name of the reconstructed SCCS file
`comb`*?????*	Temporary file

SEE ALSO

`admin`(1), `cdc`(1), `delta`(1), `get`(1), `help`(1), `prs`(1) `rmdel`(1), `sact`(1), `sccsdiff`(1), `sh`(1), `unget`(1), `val`(1), `vc`(1), `what`(1)

`sccsfile`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

comm – Selects or rejects lines common to two sorted files

SYNOPSIS

comm [-1] [-2] [-3] *file1 file2*

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The comm utility 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. File name - means that standard input is used.

The comm utility accepts the following options:

-1
-2
-3 Suppresses printing of the corresponding column. Thus, comm -12 prints only the lines common to the two files, comm -23 prints only lines in the first file but not in the second, and comm -123 prints nothing.

NOTES

If this utility is installed with a privilege assignment list (PAL), a user with one of the following active categories is allowed to perform the actions shown:

Active Category	Action
system, secadm	Allowed to manage any sorted files. In a privileged administrator shell environment, shell-redirected I/O is not subject to file protections.
sysadm	Allowed to manage any sorted files subject to security label restrictions. Shell-redirected I/O is subject to security label restrictions.

If the PRIV_SU configuration option is enabled, the super user is allowed to manage any sorted files. Shell-redirected I/O on behalf of the super user is not subject to file protections.

EXIT STATUS

The comm utility exits with one of the following values:

0 All input files were successfully output as specified.

>0 An error occurred.

SEE ALSO

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

NAME

command – Executes a simple command

SYNOPSIS

command [-p] *command_name* [*argument...*]
command [-v] *command_name*
command [-V] *command_name*

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The command utility causes the shell to treat the arguments as a simple command, suppressing the shell function lookup (see sh(1)).

In every other respect, if *command_name* is not the name of a shell function, the effect of command is the same as omitting command.

The command utility accepts the following options and operands:

-p Performs the command search using a default value for PATH that will find all of the standard utilities.

-v Writes a string to standard output that indicates the path name or command that the shell will use in the current shell execution environment, to invoke *command_name*.

-V Writes a string to standard output that indicates how the shell will interpret the name given in the *command_name* operand in the current shell execution environment.

argument A string treated as an argument to *command_name*.

command_name The name of a utility or a special built-in utility.

NOTES

The command utility described in this man page is a built-in utility to the standard shell (sh(1)). An executable version of this utility is available in /usr/bin/command.

EXIT STATUS

The command utility exits with one of the following values:

0 Successful completion.

>0 The *command_name* cannot be found or an error occurred.

126 The utility specified by *command_name* was found but could not be invoked.

127 An error occurred in the command utility or the utility specified by *command_name* could not be found.

Otherwise, the exit status of command is that of the simple command specified by the arguments to command.

SEE ALSO

sh(1)

NAME

compress – Compresses expanded files

SYNOPSIS

compress [-c] [-f] [-v] [-b *bits*] [*filename*]

compress [-f] [-v] [-b *bits*] [*filename* ...]

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4
AT&T

DESCRIPTION

The compress utility reduces the size of the named files using adaptive Lempel-Ziv coding. A .Z extension is added to the compressed file name. The ownership modes, access time, and modification time stay the same. If no files are specified, the standard input is compressed to the standard output.

The amount of compression obtained depends on the size of the input, the number of *bits* per code, and the distribution of common substrings. Usually, text such as source code or English is reduced by 50 to 60%. Compression is generally much better than that achieved by Huffman coding (as used in pack(1)), and takes less time to compute. The *bits* parameter specified during compression is encoded within the compressed file, along with a magic number to ensure that neither decompression of random data nor recompression of compressed data is subsequently allowed.

Compressed files can be restored to their original form using the uncompress utility.

The compress utility accepts the following options:

-b *bits* Sets the upper limit (in bits) for common substring codes. *bits* must be between 9 and 16 (16 is the default). Lowering the number of bits will result in larger, less-compressed files. For a portable application, bits must be between 9 and 14, inclusive.

-c Writes to the standard output; no files are changed. The nondestructive behavior of zcat is identical to that of specifying uncompress -c.

-f Forces compression, even if the file does not actually shrink, or the corresponding .Z file already exists. When running in the background (under /bin/sh), and -f is not specified, prompt to verify whether an existing .Z file should be overwritten.

-v Verbose. Displays the percentage reduction for each file compressed.

filename File to be compressed.

NOTES

Although compressed files are compatible between machines with large memory, `-b 12` should be used for file transfer to architectures with a small process data space (64 Kbytes or less).

The `compress` utility should be more flexible about the existence of the `.Z` suffix.

EXIT STATUS

The following exit values are returned:

0 Successful completion.

1 An error occurred.

2 One or more files were not compressed because they would have increased in size (and the `-f` options was not specified).

>2 An error occurred.

MESSAGES

`Usage: compress` [`-fvc`] [`-b maxbits`] [*filename* ...]
Invalid options were specified on the command line.

`Missing maxbits`
Maxbits must follow `-b`.

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

filename`: compressed with` *xx*`bits, can only handle` *yy*`bits`
filename was compressed by a program that could deal with more *bits* than the compress code on this machine. Recompress the file with smaller *bits*.

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

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

`Compression:` *xx.xx*%
Percentage of the input saved by compression. (Relevant only for `-v`.)

`-- not a regular file: unchanged`
When the input file is not a regular file, (such as a directory), it is left unaltered.

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

`-- file unchanged`
No savings are achieved by compression. The input remains uncompressed.

SEE ALSO

pack(1), sh(1), uncompress(1), zcat(1)

''A Technique for High Performance Data Compression,'' Terry A. Welch, *IEEE Computer*, vol. 17, no. 6 (June 1984), pp. 8–19.

NAME

cp – Copies files

SYNOPSIS

cp [-f] [-i] [-p] [-r] [-R] *file1* [*file2* ...] *target*

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The cp utility copies the contents of a file to another path name, or copies the contents of one or more files into path names in another directory.

The cp utility accepts the following options:

-f If an existing file cannot be overwritten, unlinks the existing file and re-creates it.

-i Prompts for confirmation whenever the copy overwrites an existing *target*. To proceed with the copy, specify y. Any other answer prevents cp from overwriting *target*.

-p Duplicates not only the contents of *file*, but also preserves the modification time and permission modes. The access control list (ACL) is preserved also.

-r, -R If *file* is a directory, cp will copy the directory and all its files, including any subdirectories and their files; *target* must be a directory.

files A path name of a file (any type) to be checked. If no *file* operands are specified, the standard input is used.

target A path name of a target file where the files will be copied.

The cp utility copies *file* to *target*. *file* and *target* may not have the same name. (Care must be taken when using shell metacharacters.) If *target* is not a directory, only one file may be specified before it; if *target* is a directory, more than one file may be specified. If *target* does not exist, cp creates a file named *target*. If *target* exists and is not a directory, its contents are overwritten. If *target* is a directory, the file(s) are copied to that directory.

If *file* is a directory, *target* must be a directory in the same physical file system. *target* and *file* do not have to share the same parent directory.

If *file* is a file and *target* is a link to another file with links, the other links remain and *target* becomes a new file.

If *target* does not exist, `cp` creates a new file named *target*, which has the same mode as *file* except that the sticky bit is not set unless the user is a privileged user; the owner and group of *target* are those of the user.

If *target* is a file, its contents are overwritten, but the mode, owner, and group associated with it are not changed. The last modification time of *target* and the last access time of *file* are set to the time the copy was made.

If *target* is a directory, a new file with the same mode is created in the target directory for each file named; the file's user ID is set to the effective user ID of the process, and the file's group ID is set to the group ID of the directory in which the file is copied.

NOTES

A `--` permits users to mark the end of any command line options explicitly, thus allowing `cp` to recognize file name arguments that begin with a `-`. If a `--` and a `-` both appear on the same command line, the second is interpreted as a file name.

If this utility is installed with a privilege assignment list (PAL), a user with one of the following active categories is allowed to perform the actions shown:

Active Category	**Action**
`system`, `secadm`	Allowed to copy any file. The values of set-user-ID and set-group-ID mode bits are preserved.
`sysadm`	Allowed to copy any file, subject to security label restrictions on the source and destination file paths. The values of set-user-ID and set-group-ID mode bits are preserved. Shell-redirected I/O is subject to security label restrictions.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to copy any file. The values of set-user-ID and set-group-ID mode bits are preserved.

EXIT STATUS

The `cp` utility exits with one of the following values:

0 No error occurred.

>0 An error occurred.

WARNINGS

Beware of a recursive copy that copies the contents of a directory to one of its subdirectories. For example:

```
$ cp -r ~/src ~/src/bkup
```

will keep copying files until it fills the entire file system.

EXAMPLES

Example 1: Use the following command line to copy file `copyone` to `copytwo`, which already exists:

```
$ cp -i copyone copytwo
overwrite copytwo? y
$
```

Example 2: Use the following command line to copy all files in your working directory that begin with the letter `b` to subdirectory `copieshere`:

```
$ cp b* copieshere
```

Example 3: Use the following command line to copy file `file.c` to subdirectory `newdir`. The copy will have the name `copy.c`.

```
$ cp file.c newdir/copy.c
```

Example 4: Use the following command line to make a copy of the directory `dir1`, naming it `dirnew`:

```
$ cp -r dir1 dirnew
```

SEE ALSO

`cat`(1), `chmod`(1), `cpio`(1), `ln`(1), `mv`(1), `rcp`(1), `rm`(1)

`chmod`(2), `creat`(2), `open`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

General UNICOS System Administration, Cray Research publication SG–2301

NAME

cpio – Copies file archives in and out

SYNOPSIS

cpio -o [-a] [-c] [-e] [-v] [-x] [-z [-x] [-M] [-P]] [-B]

cpio -i [-6] [-c] [-d] [-h *hdr-type*] [-f] [-m] [-r] [-t] [-u] [-v] [-x] [-z [-x] [-M] [-P]] [-B] [-E *filename*] [*patterns*]

cpio -p [-a] [-d] [-e] [-l] [-m] [-u] [-v] [-x] [-z [-x] [-M] [-P]] *directory*

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The cpio utility, invoked with the -o option, reads the standard input to obtain a list of path names and copies those files onto the standard output together with path name and status information.

Output is padded to a 512-byte boundary. Using the -z option outputs security information, access control lists (ACLs), and privilege assignment lists (PALs) corresponding to those files.

The -i option 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*, metacharacters ?, *, and [...] match the slash (/) character. Use double quotation marks ("), single quotation marks(’), or backslashes (\) to protect the metacharacters from being expanded by the shell. 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 in the following. 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 appropriately authorized to make cpio retain the owner and group of the files of the previous cpio -o; see the NOTES section. When reading an input created with the -z option, specifying the -z option on input preserves security information, ACLs, and PALs.

The -p option 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 in the following. Using the -z option preserves security information, ACLs, and PALs to copied files.

Only files with security levels and compartments that are dominated by the user’s active security levels and compartments may be copied. Appropriately authorized users can copy any file; see the NOTES section.

Only authorized users can copy special files; see the NOTES section.

On UNICOS systems using multilevel security labels, any user can archive data to a single-level medium. Only appropriately authorized users can archive data to a multilevel medium. Data restored from any medium is assigned the original ACL, no PAL, and the mandatory access control security attributes of the invoking user. An appropriately authorized user can restore the original file attributes and PAL by using the `-M` option. Only appropriately authorized user can restore data from a multilevel medium; see the NOTES section.

The available options are as follows. A dash (`-`) may precede the following options; dashes may be omitted if the options are concatenated without spaces.

`-6` Processes a UNIX System Sixth Edition format file. Useful only with the `-i` option (copy in).

`-a` Resets access times of input files after they have been copied.

`-B` Blocks input/output at 5120 bytes to the record (does not apply to the `-p` option; useful only with data that will be stored on tape).

`-c` Writes header information in ASCII character form for portability. This option is necessary to transfer a `cpio` archive between a Cray Research mainframe and another machine. Only an appropriately authorized user can copy restart files by using this option; see the NOTES section. This option is not valid when using the (`-oz`) option.

`-d` Creates directories as needed.

`-e` Verifies all parameters of the `stat` structure (`stat`(2)) are within certain limits. If any value is invalid, the file is not copied into the archive. This option is not guaranteed to exist in future UNICOS releases. By default, a file is copied regardless of its `stat` structure contents.

`-E` *filename*
Specifies an input file that contains a list of file names to be extracted from the archive (one file name per line).

`-f` Copies in all files except those in *patterns*.

`-h` *hdr-type*
By default, `cpio` attempts to automatically interpret the `cpio` non-ASCII header that precedes each file in the `cpio` archive (see `cpio`(5)). The `-h` option allows control over the interpretation of this non-ASCII header. If *hdr-type* is `o`, `cpio` interprets the headers as having been created in a pre-6.0 UNICOS release. If *hdr-type* is `n`, `cpio` interprets the headers as having been created in a UNICOS 6.0 or later release. If *hdr-type* is incorrectly chosen, a `phase error` will most likely occur and `cpio` will not read the archive. Use this option if `cpio` does not automatically interpret the non-ASCII header correctly. Use only with the `-i` option.

`-l` Whenever possible, link files rather than copying them. Use only with the `-p` option.

`-m` Retains previous file modification time. This is ineffective on directories that are being copied.

`-M` Preserves all security attributes of all files. Useful only with the `-z` option.

`-P` Excludes copy or preservation of PALs. Useful only with the `-z` option.

`-r` Interactively renames files. The user will be prompted with the name of the file. If the user types a null line, the file is skipped. If the line consists of a single period, the file is processed with no modification to its name. Otherwise, it's name will be replaced with the contents of the line. Use only with the `-i` option.

`-t` Prints table of contents of the input. No files are created.

`-u` Copies unconditionally. (Usually, an older file does not replace a newer file with the same name).

`-v` (Verbose) Causes a list of file names to be printed. When used with the `-t` option, the table of contents looks like the output of an `ls -l` command (see `ls`(1)).

`-x` Excludes copy or preservation of ACLs. Useful only with the `-z` option.

`-z` Copies security information and ACLs.

NOTES

The `-x`, `-M`, `-P` and `-z` options are valid only when used with the `-z` option.

Files generated with the `-z` option can be processed only using the `-z` option.

Appropriately authorized users are users that are assigned the following privilege text:

Privilege Text	Action Allowed
`LKSU`	Retains owner/group of a file and copies special files or restart files
`UPIP`	Restores security attributes from unnamed pipe
`ALL`	Restores security attributes from any medium
`ALLLKSU`	Combined behavior of `ALL` and `LKSU` privilege texts
`UPIPLKSU`	Combined behavior of `UPIP` and `LKSU` privilege texts

Note that the `LKSU` privilege text is the only one used in the PAL supplied by default. The `UPIP`, `ALL`, `ALLLKSU`, and `UPIPLKSU` privilege texts are supported if your site wants to use site-defined PALs.

If the `PRIV_SU` configuration option is enabled, root is an appropriately authorized user.

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.

EXAMPLES

Example 1: This example copies the contents of a directory into an archive.

```
ls | cpio -o >/archive/file
```

Example 2: This example duplicates a directory hierarchy.

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

Example 3: The following example uses the online tape subsystem to write and read `cpio` tapes:

```
rsv
tpmnt -l nl -p /tmp/tapedev -v vsn -b 4096
find . -print | cpio -cov > /tmp/tapedev
cd newdirectory
dd if=/tmp/tapedev bs=4096 | cpio -civd
rls -a
```

SEE ALSO

`ar`(1), `cp`(1), `find`(1), `ls`(1), `pax`(1), `privtext`(1), `rls`(1), `rsv`(1), `spset`(1), `tar`(1), `tpmnt`(1)

`chown`(2), `getpal`(2), `getppriv`(2), `setdevs`(2), `setpal`(2), `setppriv`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`acl`(5), `cpio`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

Tape Subsystem User's Guide, Cray Research publication SG–2051

UNICOS Resource Administration, Cray Research publication SG–2302

NAME

crontab – Creates or modifies the user's crontab file

SYNOPSIS

```
crontab [file]
crontab -e
crontab -l
crontab -r
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The crontab utility creates, replaces, or edits a user's crontab entry, which is a list of commands and the times at which they are to be executed. To input the new crontab entry, specify *file* on the command line. If you omit *file*, standard input is used. Use the -e to invoke an editor for the crontab file.

The crontab utility accepts the following options and operands:

-e Edits a copy of the invoking user's crontab entry, or creates an empty entry to edit if the crontab entry does not exist. When editing is complete, the entry is installed as the user's crontab entry.

-l Lists the invoking user's crontab entry.

-r Removes a user's crontab entry from the crontab directory.

file Name of the file that contains the crontab entry.

Users who are permitted to use crontab are those whose names appear in the /usr/lib/cron/cron.allow file. If that file does not exist, the /usr/lib/cron/cron.deny file is checked to determine whether a 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 that no users are allowed to use cron; null file cron.deny would mean that no users are 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 <space>s or <tab>s. 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). The specification of days may be made by two fields (day of the month and day of the week). If you specify both as a list of elements, both are followed. 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 as a `<newline>` character. The shell executes only the first line (up to a % or end-of-line character) of the command field. 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 want to have their `.profile` file executed must so state that explicitly in the `crontab` file. `cron`(8) supplies a default environment for every shell, defining `HOME`, `LOGNAME`, `SHELL(=/bin/sh)`, and `PATH(=:/bin:/usr/bin:/usr/ucb)`. You will probably want to set the `TZ` environment variable.

Blank lines and lines that begin with # are ignored.

At the time of submission, `crontab` files are run at the user's current security label.

NOTES

Redirect the standard output and standard error files of commands; otherwise, all generated output and errors will be mailed to you.

If this utility is installed with a privilege assignment list (PAL), a user who is assigned the following privilege text upon execution of this command is allowed to perform the action shown:

Privilege Text	Action
`showall`	Allowed to manage all jobs.

If this utility is installed with a PAL, a user with one of the following active categories is allowed to perform the action shown:

Active Category	Action
`system`, `secadm`, `sysadm`, `sysops`	Allowed to manage all jobs.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to manage all jobs.

EXIT STATUS

The `crontab` utility exits with one of the following values:

0 Successful completion.

>0 An error occurred.

EXAMPLES

Example 1: The following contents appear in a file in the `/usr/lib/cron/crontabs` directory. The `/u/tom/bin/hskp` file is executed by the shell at 3:00 A.M. every day of the month, every month of the year, Tuesday through Saturday:

```
0  3  *  *  2-6  /u/tom/bin/hskp
```

Example 2: The following contents appear in a file in the `/usr/lib/cron/crontabs` directory. The file `$HOME/diskwatch` is executed by the shell every half-hour, Monday through Friday:

```
0,30  *  *  *  1-5  $HOME/diskwatch
```

FILES

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

SEE ALSO

`sh`(1)

`queuedefs`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`cron`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

crypt – Encodes or decodes a file

SYNOPSIS

```
crypt [-O] [password]
crypt [-k] [-O]
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The crypt utility reads from standard input and writes on standard output. If the *password* argument is not specified, crypt demands a key from the terminal and turns off printing while the key is being typed. If the -k option is used, crypt uses the key assigned to the environment variable CrYpTkEy.

The crypt utility accepts the following options:

-O Decrypts files previously encrypted on a UNICOS system before release 5.0.

password Specifies a key that selects a particular transformation.

-k Causes crypt to use the key assigned to the CrYpTkEy environment variable.

The crypt utility encrypts and decrypts with the same key:

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

Files encrypted by crypt are compatible with those treated by the ed(1), edit(1), ex(1), and vi(1) editors in encryption mode.

The security of encrypted files depends on three factors: 1) the fundamental method must be hard to solve; 2) direct search of the key space must be infeasible; 3) ‘‘sneak paths’’ by which keys or clear text can become visible must be minimized.

The crypt utility implements a one-rotor machine designed along the lines of the German Enigma, but with a 256-element rotor. Methods of attack on such machines are known, but not widely; moreover, the amount of work required is likely to be large.

The transformation of a key into the internal settings of the machine is deliberately designed to be expensive, that is, to take a substantial fraction of a second to compute. However, if keys are restricted to three lowercase letters, encrypted files can be read in only a substantial fraction of 5 minutes of machine time.

If the key is an argument to the crypt utility, it is potentially visible to users executing ps(1) or a derivative. To minimize this possibility, crypt takes care to destroy any record of the key immediately upon entry. The choices of keys and key security are the most vulnerable aspect of crypt.

NOTES

The inclusion of encryption code requires a special license for sites outside the United States. If these encryption functions are not available on your system, check with your system support staff.

BUGS

If output is piped to `nroff` and the encryption key is not specified on the command line, `crypt` can leave terminal modes in a strange state (see `stty`(1)).

EXAMPLES

Example 1: The following example encrypts file `secrets` into file `secure`. A key must be provided (note that printing is turned off while the key is being entered). User input is shown in bold type:

```
$ crypt < secrets > secure
Enter key:
```

Example 2: To print the file to `stdout`, enter the following by using the same key as you did to encrypt the previous file:

```
$ crypt < secure
Enter key:
This is the text of the file secrets.
```

FILES

`/dev/tty`	Controlling input terminal for typed key

SEE ALSO

`ed`(1), `edit`(1), `ex`(1), `makekey`(1), `ps`(1), `stty`(1), `vi`(1)

NAME

csh – Invokes the C shell

SYNOPSIS

csh [-b] [-c] [-e] [-f] [-i] [-n] [-s] [-t] [-v] [-x] [-S] [-V] [-X] [*args*]

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The C shell is a command interpreter that has a syntax similar to that of the C language. The csh utility incorporates a history mechanism (see the History Substitution subsection) and job control facilities.

A session with csh begins with the execution of commands from file .cshrc in your home directory. However, if this is a login shell, csh executes commands from file /etc/cshrc first, then .cshrc and .login in your home directory.

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 user's home directory.

If argument 0 to the shell is a dash (-), this is a login shell.

The csh utility accepts the following options:

-b Forces a *break* from option processing. Subsequent command-line arguments are not interpreted as C shell options. This allows the passing of options to a script without confusion. The shell does not run a set-user-ID or set-group-ID script unless this option is present.

-c Reads a single command or file of commands from *args*. Any remaining arguments are placed in the *argv* variable. If no argument is specified, this option will have no effect.

-e Exits when an invoked command terminates abnormally or yields a nonzero exit status.

-f Suppresses startup processing, which consists of searching for and executing the .cshrc file in your home directory. This makes the transition into the shell faster.

`-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 when their inputs and outputs are terminals.

`-n` Parses but does not execute commands. This helps check for accuracy in the syntax of shell scripts.

`-s` Takes command input from the standard input.

`-t` Reads and executes one line of input. Use a backslash (\) to escape the new-line character at the end of this line and continue to 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.

`-S` Sets the `timestamp` variable. Prefixes commands with a date and time stamp in the form *day month date hh:mm:ss.*

`-V` Sets the `verbose` variable even before `.cshrc` is executed.

`-X` Sets the `echo` variable even before .`cshrc` is executed.

After argument processing, if arguments (*args*) remain but you did not specify the `-c`, `-i`, `-s`, or `-t` option, 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`. Remaining arguments initialize the *argv* variable.

Lexical Structure

The shell splits input lines into words at blanks and tabs with the following exceptions. The `&`, `|`, `;`, `<`, `>`, `(`, and `)` characters form separate words. If the `&`, `|`, `<`, or `>` characters are doubled to `&&`, `||`, `<<`, or `>>`, respectively, 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 new line preceded by \ is equivalent to a blank.)

In addition, strings enclosed in matched pairs of quotation marks ' or "), form parts of a word; metacharacters in these strings, including blanks and tabs, do not form separate words. Quotation mark semantics are described in the Quotations with ' and " subsection. Within pairs of ' or " characters, a newline preceded by a \ produces 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 quotation marks.

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 a semicolon (`;`). The piped commands are then executed sequentially. You can execute a sequence of pipelines without waiting for the sequence to terminate by following it with an ampersand (&).

You can put any of the preceding characters in parentheses 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 utility accepts the following list of built-in commands (execution of nonbuilt-in commands is described later). When a built-in command occurs as any component of a pipeline except the last, it is executed in a subshell.

alias
alias *name*
alias *name wordlist*

The first form prints all aliases. The second form prints the alias for *name*. The final form assigns the specified *wordlist* as the alias of *name; wordlist* is command and file name substituted. *name* cannot be alias or unalias. The C shell restricts the number of nested alias substitutions on a line to 20.

bg
bg %*job*...

Runs the current or specified jobs in the background.

break

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

breaksw

Causes a break from switch, resuming after endsw.

case *label:*

Labels a switch statement, as discussed under the default command.

cd
cd *name*
chdir
chdir *name*

Changes the shell's working directory to directory *name*. If no argument is specified, 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 ../), each component of the variable cdpath is checked to see whether it has a subdirectory *name*. Finally, if all else fails but *name* is a shell variable with a value that begins with /, then this is tried to see whether it is a directory.

continue

Continues the 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.

dmmode

`dmmode` *n* Sets the data migration recall mode to *n*. See `dmmode`(1) for usage and description.

`echo` *wordlist*
`echo -n` *wordlist*
Writes the specified words to the shell's standard output, separated by spaces and terminated with a new-line character unless the `-n` option is specified.

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

`eval`,*arg* ... As in `sh`(1), the arguments are read as input to the shell and the resulting commands 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, because parsing occurs before these substitutions.

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

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

`fg`
`fg` *%job* ... Brings the current or specified jobs into the foreground, continuing them if they were stopped.

`foreach` *name (wordlist)*
.
.
.
`end` 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.)

The `continue` statement is used to continue the loop prematurely and `break` to terminate it prematurely. When this command is read from the terminal, the loop is read once, prompting with `?` before any statements in the loop are executed. If you make a mistake typing in a loop at the terminal, you can delete it.

`glob` *wordlist* Similar to `echo`, but words are delimited by null characters in the output. `glob` is useful for programs that use the shell to file-name-expand a list of words.

`goto` *word* The specified *word* is file-name-expanded 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 specified, only the *n* most recent events are printed. The `-r` option reverses the order of printout so that the most recent is 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, the single *command* with arguments is executed. Variable substitution (see the Variable Substitution subsection) 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. Pipelines and command lists are executed outside the `if` command. I/O redirection occurs when *command* is not executed, even if *expr* is false (this is a bug).

`if` (*expr*) `then`
.
.
.
`else if` (*expr2*) `then`
.
`else`
.
`endif` If the specified *expr* is true, the commands up to the first `else` are executed; if *expr2* is true, 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`; `if`, `then`, `else`, and `endif` must be separate words.)

`jobs`
`jobs -l` Lists the active jobs. The `-l` option lists the process IDs in addition to the normal information.

`kill` %*job*
`kill` -*sig* %*job* ...
`kill` *id*
`kill` -*sig pid* ...
`kill -l` Sends either the `SIGTERM` (terminate) signal or the specified signal to the specified jobs or processes. Signals are given either 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; `kill` alone does not send a signal to the current job.

`logout` Terminates a login shell. Especially useful if the `ignoreeof` variable is set.

`nice`
`nice +`*number*
`nice` *command*
`nice +`*number*`command`

The first form adds 4 to the current `nice` value for this shell. 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. Super users may specify negative niceness by using `nice -`*number* .... The *command* is always executed in a subshell, 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` When used in shell scripts, causes hangups to be ignored for the remainder of the script. All processes detached with an ampersand (&) can effectively use `nohup`.

`notify`
`notify` %*job* ...

Causes the shell to notify the user asynchronously when the status of the current or specified job changes; usually 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 and 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, none of the forms of `onintr` have meaning, and interrupts continue to be ignored by the shell and all invoked commands.

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

`pushd`
`pushd` *name*
`pushd +`*n* The first form, without arguments, exchanges the top two elements of the directory stack. The second form changes to the new directory (as does `cd`) and pushes the old current working directory onto the directory stack. The last form, `pushd` with a numeric argument, rotates the *n*th argument of the directory stack so that it is the top element and changes 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 be necessary only 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 *command* in the one-line `if` statement, *count* times. I/O redirections occur once, even if *count* is 0.

`set`
`set` *name*
`set` *name=word*
`set` *name[index]=word*
`set` *name=(wordlist)*
The first form of the command shows the value of all shell variables. Variables that have a value other than a single word print as a parenthesized word list. The second form sets *name* to the null string. The third form sets *name* to the single *word.* The fourth form sets the *index* component of *name* to *word*; this component must already exist. The final form sets *name* equal to the list of words in *wordlist*. In all cases, the value is command and file-name-expanded.

These arguments may be repeated to set multiple values in a single `set` command. Note however, that variable expansion is done for all arguments before any setting occurs.

The *name* argument can be a maximum of 18 characters long and must begin with a letter. (The underscore character is considered a letter.) When entering the command line, include either one or more spaces on both sides of the `=` sign or no space on either side.

You can repeat these arguments to set multiple values in a single `set` command. Variable expansion is done for all arguments before any setting occurs.

`setenv`
`setenv` *name value*
`setenv` *name* The first form lists all current environment variables. The second form sets the value of environment variable *name* to *value,* a single string. The last form sets *name* to an empty string. The most commonly used environment variables (`USER`, `TERM`, and `PATH`) are automatically imported to and exported from the `csh` variables `user`, `term`, and `path`; there is no need to use `setenv` for these.

The *name* argument can be a maximum of 18 characters long and must begin with a letter. The underscore character is considered a letter.

`setucat` *cat* Sets the active category. See `setucat`(1) for usage and description.

`setucmp` *cmp* Sets active compartments. Available only to the lowest-level login shell. See `setucmp`(1) for usage and description.

`setulvl` *level* Raises the security level. Available only to the lowest-level login shell. See `setulvl`(1) for usage and description.

`setusrv` Sets the user's security attributes. See `setusrv`(1) for usage and description.

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

`source` *name*
`source -h` *name*
Reads commands from *name*. `source` commands can be nested; however, 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 `-h` option causes the commands to be placed in the history list without being executed.

`stop` %*job* ... Stops the specified job that is executing in the background.

`suspend` Causes the shell to stop abruptly, much as if it had been sent a stop signal with `<CONTROL-Z>`. This is most often used to stop shells started by `su`(1).

```
switch (string)
case label:
.
.
.
breaksw
.
.
.
default:
.
.
.
breaksw
endsw
```

Matches each case label successively against *string*, which is the first command and file name expanded. 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, execution would begin after the default label. Each case label and default label must appear at the beginning of a line. The `breaksw` command causes execution to continue after the `endsw;` otherwise, control may fall through case labels and default labels as in C. When no label matches and no `default` exists, execution continues after `endsw.`

`time`

`time` *command* With no argument, the shell prints a summary of time used by this shell and its child processes. When the argument is specified, the shell times the specified simple *command* and prints a time summary as described under the `time` variable. If necessary, an extra shell will be created to print the time statistic when the *command* completes.

`umask`

`umask` *value* The first form of the command displays the file creation mask. The second form sets the file creation mask to the specified value. The mask is specified in octal values. Common values for the mask are 002, which gives all access to the group and read and execute access to others, or 022, which gives all access except write to users in the group and to others.

`unalias` *pattern*

Discards all aliases with names that match *pattern*. All aliases are removed by `unalias *`. The omission of *pattern* is acceptable.

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

`unset` *pattern* Removes all variables with names that match *pattern*. To remove all variables, use `unset *`; this has noticeably negative side-effects. The omission of *pattern* is acceptable.

`unsetenv` *pattern*

Removes from the environment all variables with a name that matches *pattern*. See also the preceding `setenv` command and `printenv`(1B).

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

`which` *name* ... Searches for the file that would be executed had *name* been specified as a command. *name* is expanded if it is aliased, and searched for along your path.

`while` *(expr)*
.
.
.
`end`

Although the specified expression evaluates nonzero, the commands between the `while` and the matching `end` are evaluated. Built-in commands `break` and `continue` may be used to terminate or continue the loop prematurely. (`while` and `end` must appear alone on their input lines.) If the input is a terminal, prompting occurs here the first time through the loop as it does with the `foreach` statement.

`%`[*job*][`&`] Brings the current or indicated job to the foreground. When & is included, the job continues to run in the background.

@
@ *name=expr*
@ *name*[*index*]*=expr*

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

The operators *=, +=, and so on, are available as in C. You must include either one or more spaces on both sides of the = symbol or no space on either side. Spaces are mandatory in separating components of *expr* that 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 by using execv (see exec(2)). Each word in the PATH variable names a directory from which the shell will attempt to execute the command. When it is not given a -c or -t option, the shell hashes the names in these directories into an internal table so that it executes an exec(2) in a directory only 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 option (and for each directory component of PATH that does not begin with /), the shell will concatenate with the given command name to form a path name of a file, which it will then attempt to execute.

Parenthesized commands are always executed in a subshell. For example, the following command prints the home directory; leaving you where you were (printing this after the home directory).

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

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

```
cd ; pwd
```

When a path name that has proper execution permissions is found, the shell forks a new process and passes it, along with its arguments to the kernel (using the execv(2) system call). The kernel then attempts to overlay the new process with the desired program. When the file is an executable binary file (see a.out(5)), the kernel succeeds and begins executing the new process. If the file is a text file, and the first line begins with #!, the next word is taken to be the path name of a shell (or command) to interpret that script. Subsequent word(s) on the first line, usually only 1 if any, are used as a single option to that shell. The kernel invokes (overlays) the original shell with the indicated shell, using the name of the script as an argument, which is preceded by the previously mentioned option, if any, and followed by the original user-supplied arguments, again if any. If the indicated shell is a setuid executable file (see setuid(2)), the kernel will not overlay the original shell.

If neither of the preceding conditions holds, the kernel cannot overlay the file (the `execv` call will fail); the C shell then attempts to execute the file by spawning a new shell, as follows:

- If the first character of the file is #, a C shell will be invoked.
- Otherwise, a standard shell (`/bin/sh`) will be invoked.

When there is an alias for *shell*, the words of `alias` are prepended to the argument list to form the shell command. The first word of `alias` should be the full path name of the shell (for example, `$shell`). This is a special, late-occurring case of `alias` substitution, and it allows only 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 the following line:

```
[1] 1234
```

This line indicates that the job that was started asynchronously, was job number 1, and it had one (top-level) process, with a process ID of 1234.

The shell redirects standard input to `/dev/null` for a job being run in the background. This results in the receipt of an end-of-file if such a job tries to read from the terminal. Background jobs are usually 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 `%1`. Jobs can also be named by prefixes of the string entered in to start them if these prefixes are unambiguous. It is also possible to enter %?*string*, which specifies a job with text that contains *string* if there is only one such job.

The shell maintains the current and previous jobs. In output pertaining to jobs, the current job is marked with a + symbol and the previous job with a - symbol. The abbreviation `%+` refers to the current job, and `%-` refers to the previous job. The characters %% are a synonym for the current job.

Status Reporting

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

Substitutions

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

- History substitution
- Alias substitution

- Variable substitution
- Command substitution
- File-name 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, and correct spelling mistakes in the previous command with little typing and a high degree of confidence. History substitutions begin with `!` and may begin anywhere in the input stream (as long as they do not nest.) The `!` can be preceded by `\` to override its special meaning; for convenience, `!` is passed unchanged when it is followed by a blank, tab, new-line character, `=`, or `(`. Note that `!~` is a `csh` operator and cannot be used for history substitution. (History substitutions also occur when an input line begins with a carat (`^`), described later.) Any input line that contains history substitution is echoed on the terminal before it is executed because it could have been typed without history substitution.

Commands input from the terminal consisting of 1 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.

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 you can make the current event number part of the prompt by placing `!` in the prompt string.

With the current event 13, you can refer to previous events by event number, such as `!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, reintroduce the words of the specified events, each separated by a blank character. As a special case, `!!` refers to the previous command; thus `!!` alone is essentially `redo.`

To select words from an event, add a colon (`:`) and a designator for the desired words to the event specification. The words of an input line are numbered starting at 0. The first (usually command) word is 0, the second (first argument) is 1, and so on. The basic word designators are as follows:

`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*	Abbreviation `0-`*y*
*	Abbreviation `^-$`, or nothing if only 1 word in event
x *	Abbreviation *x*–`$`
x–	Like *x** but omits word `$`

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

`h`	Removes trailing path name component, leaving the head
`r`	Removes trailing *.xxx* component, leaving the root name
`e`	Removes all but the extension *.xxx*
`s/`*l*`/`*r*`/`	Substitutes *r* for *l*
`t`	Removes all leading path name components, leaving the tail
`&`	Repeats the previous substitution
`g`	Applies the change globally, prefixing & (for example, `g&`)
`p`	Prints the new command but does not execute it
`q`	Places quotation marks around the substituted words, preventing further substitutions
`x`	Like `q`, but breaks into words at blanks, tabs, and new-line characters

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

The left-hand side of substitutions are strings rather than regular expressions in the sense of the editors. Any character may be used as the delimiter in place of the `/` symbol; a `\` places quotation marks around the delimiter in the *l* and *r* strings. The character & in the right-hand side is replaced by the text from the left. A `\` quotes & also. A null *l*" uses the previous string, either from a *l*" or from a contextual scan string *s*" in `!?`*s*`?`. The trailing delimiter in the substitution may be omitted if a new line follows immediately, as may the trailing `?` in a contextual scan.

A 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 nonblank character of an input line is a caret (^). This is equivalent to `!:s^`, providing a convenient shorthand for substitutions in 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 braces `{}` if necessary to insulate it from the characters that follow. For example, after `ls -ld ~paul`, if you use `!{l}a` to do `ls -ld ~paula`, `!la` will look for a command starting with `la`.

Quotations with ′ and "

You can put single and double quotation marks around strings (′ and ") to override all or some of the remaining substitutions. Strings enclosed in ′ are prevented from any further interpretation. Strings enclosed in " may be expanded as described in the following.

In both cases, the resulting text becomes (all or part of) a single word; only in one special case (see the Command Substitution subsection) does a string enclosed in " yield parts of more than 1 word. Strings enclosed in ′ never yield parts of more than 1 word.

Alias substitution

The shell maintains a list of aliases that 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 is checked left-to-right to see whether it has an alias. If it does, the text that is the alias for that command will be 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, the argument list will remain unchanged.

For example, if the alias for `ls` were `ls -l`, the command `ls /usr` would map to `ls -l /usr`; the argument list would be undisturbed. Similarly, if the alias for `lookup` were `grep !^ /etc/passwd`, `lookup bill` would map to `grep bill /etc/passwd`.

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

The mechanism allows aliases to introduce parser metasyntax. Thus, you can use `alias print 'pr \!* | exlp'` to make a command that `pr`'s its arguments to the line printer.

Variable substitution

The shell maintains a set of variables, each of which has as its 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, several are toggling; the shell disregards their value, checking only to see whether they are set. For instance, toggling the `verbose` variable 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 by the $ character. You can prevent this expansion by preceding the $ with a \ except within double quotation marks (") where it always occurs, and within single quotation marks (') where it never occurs. Strings enclosed by ` are interpreted later (see the Command Substitution subsection); $ substitution does not occur there until later, if at all. $ is passed unchanged if it is followed by a blank, tab, or end-of-line character.

Input/output redirections are recognized before variable expansion, and they 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 1 word, the first of which becomes the command name, and the rest of which become arguments.

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

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

$*name*
${*name*} Replaced by the words of the value of variable *name*, each separated from the others by a blank. Braces insulate *name* from following characters, which would otherwise be part of it. Shell variables have names consisting of up to 18 letters and digits starting with a letter. The underscore character is considered a letter. If *name* is not a shell variable, but is set in the environment, that value will be returned (but : modifiers and the other forms specified 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 either a single number or two numbers separated by a - symbol. The first word of a variable's value is numbered 1. If the first number of a range is omitted, it defaults to 1. If the last member of a range is omitted, it defaults to $#*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*} Specifies the number of words in the variable. This is useful later in [*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[*]`.

Modifiers `:h`, `:t`, `:r`, `:q`, and `:x` may be applied to the preceding substitutions (except for `$0`) as may `:gh`, `:gt`, and `:gr`. If braces `{ }` appear in the command form, the modifiers must appear within the braces. The current implementation allows only one `:` modifier on each `$` expansion.

The following substitutions may not be modified with `:` modifiers:

$?*name*
${?*name*} Substitutes 1 if name is set, 0 if it is not.

$?0 Substitutes 1 if the current input file name is known, 0 if it is not.

$$ Substitutes the (decimal) process number of the (parent) shell.

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

Command and file-name substitution

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

Command substitution

Command substitution is indicated by a command enclosed in single quotation marks (`). The output is broken into separate words at blanks, tabs, and new lines, and null words are discarded. This text then replaces the original string. Within double quotation marks ("), only new lines force new words; blanks and tabs are preserved.

In any case, the single final new line does not force a new word. Note that this makes it possible for a command substitution to yield only part of a word, even if the command outputs a complete line.

File-name substitution

If a word contains any of the characters `*`, `?`, `[`, or `{`, or begins with the character `~`, that word is a candidate for file-name substitution (also known as *globbing*). This word is then regarded as a pattern and is replaced with an alphabetically sorted list of file names that match the pattern. In a list of words specifying file name substitution, it is an error if no pattern matches an existing file name, but it is not required that each pattern match. Metacharacters `*`, `?`, and `[` are the only ones that imply pattern matching; the characters `~` and `{` are more akin to abbreviations.

In matching file names, the dot character (`.`) at the beginning of a file name or immediately following a `/`, as well as the character `/`, must be matched explicitly. The character `*` matches any string of characters, including the null string. The character `?` matches any single character. The sequence `[...]` matches any one of the characters enclosed. Within `[...]`, a pair of characters separated by `-`, matches any character lexically between the two.

The ~ character at the beginning of a file name is used to refer to home directories. When the ~ is alone, 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 that user's 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.

Metanotation `a{b,c,d}e` is shorthand for `abe ace ade`. Left-to-right order is preserved, with results of matches being sorted separately at a low level to preserve this order. This construction 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 output of a command may be redirected with the following syntax:

`< name` — Opens file *name* (which is first variable, command, and file-name expanded) as the standard input.

`<< word` — Reads the shell input up to a line that is identical to *word*. *word* is not subjected to variable, file-name, or command substitution, and each input line is compared to *word* before any substitutions are done on this input line. Unless a quoting \, ", ´, or ` appears in *word*, variable and command substitution are performed on the intervening lines, allowing \ to add quotation marks to $, \, and `. Commands that are substituted have all blanks, tabs, and new lines preserved, except for the final new line, which is dropped. The resultant text is placed in an anonymous temporary file that 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 will be created; if the file exists, it will be truncated, losing its previous contents.

If the variable `noclobber` is set, the file must not exist or be a character special file (for example, a terminal or `/dev/null`); if it is, an error will result. This helps prevent accidental destruction of files. In this case, the `!` forms can be used to suppress this check.

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

`>> name`
`>>& name`
`>>! name`
`>>&! name` — Uses file *name* as standard output, for example, `>`, but places output at the end of the file. If the variable `noclobber` is set, the file must not exist unless one of the `!` forms is specified; otherwise, it is similar to `>`.

A command receives the environment in which the shell was invoked, as modified by the input-output parameters and the presence of the command in a pipeline. Unlike the case in some previous shells, commands run from a file of shell commands have no access to the text of the commands by default; instead they receive the original standard input of the shell. The `<<` mechanism should be used to present in-line data. This permits shell command scripts to function as components of pipelines and allows the shell to block read its input. 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, 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 `|`.

Expressions

A number of the built-in commands take expressions, in which the operators are similar to those of C, with the same precedence. Within an individual precedence group, however, the expressions are evaluated from right to left, as opposed to the standard left to right. These expressions appear in the `@`, `exit`, `if`, and `while` commands. The following operators are available:

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

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, `*`, `?`, and instances of `[...]`) against which the left-hand operand is matched. This reduces the need for the *switch* statement in shell scripts when all that is really needed is pattern matching.

Strings that begin with 0 are octal numbers; valid octal digits are 0 through 7. Strings that begin with `0x` or `0X` are hexadecimal numbers; valid hexadecimal digits are 0 through 9, a through f, and A through F. Null or missing arguments are considered 0. The results of all expressions are strings, which represent decimal numbers.

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

Also available in expressions as primitive operands are command executions enclosed in braces `{}` and file inquiries of the form `-` *lname*;

l is one of the following:

`r`	Read access
`w`	Write access
`x`	Execute access
`e`	Existence
`l`	Symbolic link
`o`	Ownership
`z`	Zero size
`f`	Plain file

m Migrated file (type IFOFL)
M Migrated file (has a DMF handle)
d Directory

The specified name is command and file name expanded and then tested to see whether it has the specified relationship to the real user. If the file does not exist or is inaccessible, all inquiries 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, the command should be executed outside 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, because of 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.

If the shell's input is to be searched, the shell buffers 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 allowed, backward `goto` statements succeed on nonseekable inputs.)

Predefined and Environment Variables

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

The shell copies the environment variable `LOGNAME` into the variable `user`, `TERM` into `term`, and `HOME` into `home;` these are then copied back into the environment whenever the normal shell variables are reset. The environment variable `PATH` is handled likewise; you do not have to worry about its setting other than in the `.cshrc` file because child `csh` processes reset the path variable when you use the `source` command on the `.cshrc` file.

`argv` Sets the arguments to the shell. Positional parameters are substituted from this variable; for example, `$1` is replaced by `$argv[1]`.

`cdpath` Provides a list of alternative directories that are searched to find subdirectories in `chdir` commands.

`cwd` Provides the full path name of the current directory.

`echo` Set when the `-x` command-line option is specified. 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 file-name substitution because 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 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 number of events is not discarded. If the value of `history` is too large, the shell may run out of memory. The last executed command is saved on the history list.

`home` The user's home directory, initialized from the environment. The file-name expansion of `~` refers to this variable. Setting `home` to a path name that begins with `.` or `..` causes errors.

`ignoreeof` When set, the shell ignores end-of-file from input devices, which are terminals. This prevents shells from accidentally being killed by one or more `<CONTROL-d>`.

`mail` Specifies the files in which 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 displays `You have new mail` if the file exists with an access time not greater than its modification time.

When 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.

When multiple mail files are specified, the shell displays `New mail in` *name* when there is mail in 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` When set, inhibits file-name expansion. This is most useful in shell scripts that are not dealing with file names, or when a list of file names has been obtained and further expansions are not desirable.

`nonomatch` When set, specifies that a file-name expansion does not have to 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 produces an error.

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

`path` Each word of the path variable specifies a directory in which commands are to be sought for execution. A null word specifies the current directory. If there is no `path` variable, only full path names will execute. The usual search path is `/bin`, `/usr/bin`, and `/usr/ucb`, 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 usually hashes the contents of the directories in the `path` variable after reading `.cshrc` and also 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 specify the `rehash` command; otherwise, the commands may not be found.

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

`savehist` Contains a numeric value that controls the number of entries of the history list that are saved in `~/.history` when the user logs out. Any command referenced in this number of events is saved. During startup, the shell sources `~/.history` into the history list, enabling history to be saved across logins. Values of `savehist` that are too large can slow the shell during startup.

`shell` The file in which the shell resides. This is used in forking shells to interpret files that have execute bits set that are not executable by the system. (See the description of nonbuilt-in command execution.) This file is initialized to the (system-dependent) home of the shell.

`status` The status returned by the last command. If it terminated abnormally, 0200 is added to the status. Built-in commands that fail return exit status 1; all other built-in commands set the status to 0.

`time` Controls the automatic timing of commands. If it is set, any command that takes more than this number of CPU seconds prints a line specifying user, system, and real times. It also prints a usage percentage, which is the ratio of user plus system times to real time that is printed when it terminates.

`timestamp` Set by the `-S` command-line option. Causes a date and time stamp in the form *day month date hh:mm:ss* to be `echo`ed before each command is executed.

`verbose` Set by the `-v` command-line option, which prints the words of each command after history substitution.

Nonbuilt-in Command Execution

When a command to be executed is found to not be a built-in command, the shell attempts to execute the command through `execv` (see `exec`(2)). Each word in the variable *path* names a directory from which the shell attempts to execute the command. If neither option `-c` or option `-t` is specified, the shell will hash the names in these directories into an internal table so that it will try an `exec` in a directory only 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 (by `unhash`), or if `-c` or `-t` was specified, for each directory component of *path* that does not begin with a `/`, the shell will concatenate with the specified command name to form a path name of a file, which it will then attempt to execute.

Signal Handling

The shell ignores `quit` signals. Jobs running detached (either by `&` or the `bg` or `%. . . &` commands) are immune to signals generated from the keyboard, including hangups. Other signals have the values that the shell inherited from its parent. `onintr` can control the shell's 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. Interrupts are not allowed when a login shell is reading the `.logout` file.

NOTES

Words can be no longer than 1024 characters. The system limits argument lists to 50,000 characters. The number of arguments to a command involving file-name expansion is limited to one-sixth the number of characters allowed in an argument list. Command substitutions cannot substitute more characters than are allowed in an argument list.

The `csh` utility does not send messages about illegal options.

The `csh` metacharacters will not match those characters that are greater than or equal to 0200.

The C shell will not execute the last command line in a script file if the last line is not terminated with a newline character. Some editors, such as `emacs`(1), are capable of creating such files.

BUGS

When a command is restarted from a stop, the shell prints the directory in which it started if this is different from the current directory; this can be incorrect because the job may have changed directories internally.

Shell built-in functions cannot be stopped and restarted. Command sequences of the form `a ; b ; c` are also not handled gracefully when stopping is attempted. If you suspend `b`, the shell will then immediately execute `c`. This is especially noticeable if this expansion results from an alias. It suffices to place the sequence of commands in parentheses to force it to a subshell, that is, `( a ; b ; c )`.

Control over tty output after processes are started is primitive.

Alias substitution is most often used to simulate shell scripts, but this is not efficient; shell scripts should be provided.

Commands within loops, prompted 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 may be possible to use the : modifiers on the output of command substitutions. All and more than one : modifier may be allowed on $ substitutions.

Although set and setenv allow the definition of variable names that do not begin with a letter or underscore or that are longer than 18 characters, use of such variables results in an error.

If onintr follows onintr - in a shell script, interrupts continue to be ignored.

A job started asynchronously with & and containing command substitution is not protected from interrupts.

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
/etc/passwd	Source of home directories for ~*name*
/etc/cshrc	Read at beginning of execution of shell, before .cshrc
/tmp/sh*	Temporary file for <<

(See the File-name Substitution subsection for a discussion of the ~ character in file names.)

SEE ALSO

setucat(1), setucmp(1), setulvl(1), setusrv(1), sh(1)

access(2), exec(2), fork(2), kill(2), pipe(2), setuid(2), signal(2), umask(2), wait(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

a.out(5), cshrc(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

dmmode(1) Online only

NAME

csort – Sorts and/or merges blocked files

SYNOPSIS

csort [-e] [-k *keyfile*] [-l *statfile*] [-r] *outfile sortfiles*

csort [-e] [-k *keyfile*] [-l *statfile*] -m *mfiles* [-n] [-r] *outfile* [*sortfiles*]

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The csort utility orders records of unsorted *sortfiles* and/or previously sorted -m *mfiles* and writes the result to the file named by the required *outfile* specification. The csort utility reads and writes files, depending on the sort keys you provide. If every key field is either a DEC key or a CHR key, input and output files are considered to be character files. If any key is BIN, FLT, or INT, the input and output files are considered to be Fortran unformatted files.

Two synopses for csort are provided to show that csort can be invoked only with either unsorted *sortfiles*, or with previously sorted *mfiles* and optionally one or more *sortfiles*. Invocation of csort with only the *outfile* operand will be unsuccessful.

The csort utility accepts the following options:

-e Writes the directives from *keyfile* into *statfile*.

-k *keyfile* The *keyfile* argument names the file that contains the directives to be used to determine tuning parameters and/or keys to determine the sorting scheme. For information about these directives, see the DIRECTIVES section.

If the -k option is not specified, csort provides default values for the sorting process. These defaults are as follows:

- Key type is a character string.
- Ascending sort order.
- Key field starts with the first character of the first 64-bit word of the record.
- Key field ends with the last character of the first 64-bit word of the record.
- ASCII collating sequence.

When there are multiple sort keys, later keys are compared only if earlier keys are equal.

-l *statfile* The *statfile* argument names the file to contain the listing of the statistics on the sorting process. The default file name is SRTSTAT.

`-m` *mfiles* The files specified by *mfiles* are assumed to be sorted and ready for merging. They will be merged with each other and with intermediate output from the sorting process of the unsorted *sortfiles*.

The files named in *mfiles* can be designated by using one of the following forms:

- List of files separated from one another by a comma.
- List of files enclosed in quotation marks and separated by commas and/or white space.

`-n` The files specified by *mfiles* are not checked to verify whether they have been sorted previously. The `-m` option must also be specified.

`-r` Retains the original input order of a sequence of records with equivalent keys.

outfile Specifies the file where the results of `csort` are stored.

sortfiles Specifies the files to sort and/or merge.

For suitably large files, `csort` will require temporary files. By default, these files are created in the directory specified by your `TMPDIR` environment variable. In all but the most demanding applications, this will work well.

In some cases, however, you may need a greater degree of control over the allocation of temporary files. You may initialize the environment variable `CSORTDIR` to specify the directories within which temporary files are created.

`CSORTDIR` is a colon-separated list of directory names. As temporary files are needed, they are created round-robin within the provided list of directories. (If `CSORTDIR` is defined but null, temporary files will be created in your current directory, which may or may not be desirable.

DIRECTIVES

The two `csort` directives are `KEY` and `TUNE`. The `KEY` directive lets you define the keys that determine the major and minor fields on which a sort is to be performed. The `TUNE` directive permits you to optimize the sort operation. Only `KEY` is required. The directives must be in a file specified with the `-k` option on the `csort` command line.

KEY Directive

The required `KEY` directive defines the keys of the major and minor field on which a sort is to be performed. The major field is the one on which `csort` first orders the records. If two or more records contain identical major fields, `csort` uses the minor fields to order them.

The `KEY` directive can appear any time and as often as necessary. The number of specifications is unlimited. Keys are given priority by the order in which they are specified.

The format of the `KEY` directive is as follows:

`KEY,TYPE=`*type*`,ORDER=`*order*`,START=`*w*`:`*c*`:`*b*`,END=`*w*`:`*c*`:`*b*`,COLSEQ=`*colseq*

`TYPE=`*type* Specifies the form in which the key appears in the record; *type* can be one of the following:

`BIN` Binary bit stream (unsigned).

`FLT` Floating-pont number (64-bit Cray internal format).

`INT` Integer value (a maximum of 64 bits in twos complement Cray format).

`CHR` Character string. The string's field is the length defined by the `START` and `END` parameters. The default character size is 8 bits; word size is 64 bits. This key causes `csort` to read the file in character mode, decompressing blanks.

`DEC` Decimal number (real number with a decimal point or integer) in ASCII or EBCDIC characters. `csort` does not accept real numbers with exponents. Leading and trailing blanks are insignificant; embedded blanks are interpreted as 0. The + or – sign precedes the number or decimal point. This key causes `csort` to read the file in character mode, decompressing blanks.

If you specify `DEC`, `csort` converts all numbers to floating-point numbers. Thus, `10` and `10.` can be used interchangeably.

`ORDER=`*order* Specifies the order in which `csort` sorts the key; *order* can be either `ASCEND` (sorts the records in ascending order by the key rank; default) or `DESCEND` (sorts the records in descending order by the key rank).

`START=`*w*`:`*c*`:`*b* Required parameter; no defaults. The starting position of the field containing the key. The end position is not required for floating-point or integer keys.

If you specified `FLT` or `INT` for the key *type*, the key defaults to 64 bits; therefore, you do not need to specify an end position unless the key length is other than 64 bits. The starting position specifiers are the only key location identifiers necessary.

Specify integer numbers as follows:

w Starting position of a word

c Starting position of a character

b Starting position of a bit

Count words, characters, and bits from the left beginning with 1, not 0. You can define a field as a character position within a word (*w:c*). The following example defines the seventh character of the third word:

```
START=3:7
```

You can also define a field as a character position without a word position by putting an asterisk in place of the omitted unit. The following example is equivalent to the previous example:

```
START=*:23
```

Because each word is 8 characters, the seventh character of the third word is the twenty-third character.

Similarly, you can specify a bit position directly in a word or in a character position.

END=*w*:*c*:*b* Indicates the ending position of the field containing the key. The END parameter is required only with sequence types BIN, CHR, and DEC. END is optional with FLT and INT. If the type of the key is FLT or INT, the default is the beginning of the field as specified by the START directive plus 64 bits.

Specify integer numbers as follows:

w Ending position of a word

c Ending position of a character

b Ending position of a bit

COLSEQ=*colseq* Specifies the collating sequence for the CHR or DEC key type. The choices are the following:

ASCII Sorts according to the ASCII sequence (default).

ASCIIUP Sorts according to the ASCII sequence except that lowercase letters are treated as if they are uppercase.

EBCDIC Sorts ASCII characters according to the EBCDIC sequence.

EBCDICUP Sorts ASCII characters according to the EBCDIC sequence except that lowercase letters are treated as if they are uppercase.

EBCDIC and EBCDICUP are used when the ASCII characters in the output file are to be translated from ASCII to EBCDIC (for example, during transfer to a front-end machine that used EBCDIC). The resulting translated file then has EBCDIC characters in EBCDIC sequence. If the Cray file contains EBCDIC characters that have not been converted to ASCII, the ASCII or ASCIIUP sequences (which order elements based on the numerical bit value) yield the expected results.

The ASCII, ASCIIUP, EBCDIC, and EBCDICUP collating sequences are the only ones available with the csort control statement. To define any other collating sequence, you must use the csort subroutines and identify the collating sequence with a SAMSEQ call.

TUNE Directive

The optional TUNE directives optimizes csort through the efficient allocation of resources. To judge the effect of the options, you need the statistics found in the dataset indicated by the L parameter on the csort control statement. The TUNE directive can appear as often as necessary. If a parameter is speciefed more than once in different directives, the last value specified takes precedence over the previous ones.

Use the TUNE directive to specify information about the files to be sorted and resources to be allocated. When you make these specifications, accuracy is important; inaccurate estimates can degrade performance.

The format of the TUNE directive is as follows:

```
TUNE,AVRL=n,MXRL=n,NRECEST=n, DISK=name:name...,DSLO=n,
NAMEBM=namebm,NAMESSD=namessd,NDS=n,
NDSSD=n,NBSSD=n,NDBM=n,NBBM=n,NBDSK=n, MNBL=n,MXBL=n
```

AVRL=*n*	Specifies the average record length. The default is the maximum record length.
MXRL=*n*	Specifies the maximum record length. The default is 20 Cray words (160 bytes). If the maximum record length is too short, csort aborts during the input phase.
NRECEST=*n*	An estimate of the number of records in the input files. The default is 1,000,000 records. This value is no longer used.
DISK=*name:name...*	No longer used in this implementation. Instead, you may specify a colon-separated list of directories in the CSORTDIR environment variable.
DSLO=*n*	No longer used in this implementation.
NAMEBM=*namebm*	No longer used in this implementation.
NAMESSD=*namessd*	No longer used in this implementation.
NDS=*n*	Specifies the number of temporary datasets to be used during the merge phase. The default is 10; minimum is 4. Change this parameter only if you must run Sort/Merge in minimum memory. A large value is recommended, but many temporary datasets are assigned smaller buffers and buffers should be large to maximize I/O efficiency. This makes it difficult to assign an efficient number.
NDSSD=*n*	No longer used in this implementation.
NBSSD=*n*	No longer used in this implementation.
NDBM=*n*	No longer used in this implementation.
NBBM=*n*	No longer used in this implementation.
NBDSK=*n*	Specifies the number of sort buffers to be allocated to each temporary dataset. The size of the buffer is specified by the MNBL= and MXBL= parameters. The default value is 2.
MNBL=*n*	Specifies the number of word blocks in each sort buffer. The default is 42 (the track size of a DD-49). If you supply both a minimum and a maximum, the minimum must be the smaller of the two numbers.
MXBL=*n*	Specifies the number of word blocks in each sort buffer. The default is 42. If you supply both a minimum and a maximum, the minimum must be the smaller of the two numbers.

NOTES

When calling any of the sort routines from within an applications program, you must specify that the sort library, `/usr/lib/libsort`, be loaded along with your program and the standard system default libraries. To do this, you can use the following command line (provided that your program requires no other special loading options):

`segldr -lsort` *your_program*

EXAMPLES

Example 1: This example sorts the contents of `infile1` and `infile2`, placing the output in `outfile` and using the default sorting key:

```
csort outfile infile1 infile2
```

Example 2: This example sorts, in reverse order, the records found in `infile1` and `infile2`, placing the output in `outfile` and using only the first character of the second word as the sort key.

```
csort -k keyfile outfile infile1 infile2
```

The key information is obtained from this keyfile:

```
key,type=chr,order=descend,start=2:1,end=2:2.
```

Example 3: This example merges the records of the already sorted files `sorted1` and `sorted2` along with the sorted output from the unsorted file `unsort1`, placing the output in `outfile` and using the integer located in the first possible position of the record as the sort key.

```
$ csort -m sorted1,sorted2 -k keyfile outfile unsort1
```

The key information is obtained from this keyfile:

```
key,type=int,start=1:1.
```

Example 4: This example sorts direct access files:

```
#!/bin/csh
rm -rf mkdata.f srtfile outfile keyfile
cat > mkdata.f << EOF
      program makedata
      implicit integer (A-Z)
      character*8 junk
30    FORMAT(I5)
      write(junk,'(A8)') 'srtfile'L
      open ( unit=8, file=junk , access='direct',
     C       form='unformatted', status='NEW' , recl=24 )
      iseed = IRTC()
      CALL RANSET(iseed)
      DO 10 I=1,10
```

```
      K=INT(RANF()*10000+10000)
      x=i*4.0
      y=x/2.0
      WRITE(8,REC=I) K, X, Y
10    continue
      close(8)
      stop
      end
EOF
#
cat > keyfile << EOF
key,type=int,start=1:1:1.
EOF
#
cf77 -o mkdata mkdata.f
#
./mkdata
#
assign -F ibm.f:24 srtfile
assign -F ibm.f:24 outfile
#
csort -k keyfile outfile srtfile
#
echo "============================================================"
echo "                    UNSORTED INPUT"
echo "============================================================"
od srtfile
#
echo "============================================================"
echo "                    SORTED INPUT"
echo "============================================================"
od outfile
```

FILES

SRTSTAT Default *statfile*

SEE ALSO

sort(1)

SAMKEY(3F), SAMTUNE(3F) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NAME

csplit – Splits files based on context

SYNOPSIS

csplit [-f *prefix*] [-k] [-n *number*] [-s] *file args*

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

The csplit utility reads *file* and separates it into sections, one section for each of the *args* arguments. By default, the pieces are placed in files named xx00, xx01, ..., xx*n*, where *n* is 99, by default. 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*.

The csplit utility accepts the following options and operands:

-f *prefix* Specifies the created files *prefix*00, *prefix*01, ..., *prefixn*. The default is xx00, xx01, ..., xx*n*. If the *prefix* argument would create a file name that exceeds {NAME_MAX} bytes, an error occurs and csplit exits.

-k Leaves previously created files intact. By default, csplit removes created files if an error occurs.

-n *number* Uses *number* decimal digits to form file names for the file pieces. The default is 2.

-s Suppresses the printing of all character counts. The csplit utility usually prints the character counts for each file created.

file The path name of a text file to be split. If *file* is -, the standard input is used.

args The *args* operands can be a combination of the following:

/*rexp*/[*offset*]	Creates a file for the piece from the current line up to (but not including) the line that contains the regular expression *rexp*. The optional *offset* is a positive or negative integer value preceded by a + or -. After the section is created, the current line is set to the line that results from the evaluation of the regular expression with any offset applied.
%*rexp*%[*offset*]	This argument is the same as /*rexp*/, except that no file is created for the section.
line_no	A file is to be created from the current line up to (but not including) line number, *line_no*. The current line becomes *line_no*.
{*num*}	Repeat argument. This argument may follow any of the preceding 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 a shell in the appropriate quotation marks. Regular expressions may not contain embedded new lines. `csplit` does not affect the original file; the user must remove it.

NOTES

If this utility is installed with a privilege assignment list (PAL), a user with one of the following active categories is allowed to perform the actions shown:

Active Category	Action
`system`, `secadm`	Allowed to split any file. In a privileged administrator shell environment, shell-redirected I/O is not subject to file protections.
`sysadm`	Allowed to split any file subject to security label restrictions. Shell-redirected I/O is subject to security label restrictions.

If the `PRIV_SU` configuration option is enabled, the super user is allowed to split any file. Shell-redirected I/O on behalf of the super user is not subject to file protections.

EXIT STATUS

The `csplit` utility exits with one of the following values:

0 Successful completion.

>0 An error occurred.

MESSAGES

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

```
arg - out of range
```

This message means that the given argument did not reference a line between the current position and the end of the file.

EXAMPLES

Example 1: This example creates four files, `fort00` ... `fort03`:

```
csplit -f fort file.f '/subroutine xyz/' '/function abc/' '/blockdata/'
```

After editing the ''split'' files, they can be recombined, as follows:

```
cat fort0[0-3] > file.f
```

This example overwrites the original file.

Example 2: This example splits the file at every 100 lines, up to 10,000 lines. The `-k` option causes the created files to be retained if less than 10,000 lines exist; however, an error message is still printed.

```
csplit -k file '100' '{99}'
```

Example 3: Assuming that `prog.c` follows the typical `C` coding convention of ending routines with a `}` at the beginning of the line, this example creates a file that contains each separate C routine (up to 21) in `prog.c`.

```
csplit -k prog.c '%main(%' '/^}/+1' '{20}'
```

Example 4: This example creates up to 20 chapter files from the file `novel`:

```
csplit -k -f chap. novel '%CHAPTER%' '{20}'
```

SEE ALSO

`ed`(1), `sh`(1), `split`(1)