NAME

intro — introduction to commands and application programs

DESCRIPTION

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

(1) Commands of general utility.
(1C) Commands for communication with other systems.
(1G) Commands used primarily for graphics and computer-aided design.
(1M) Commands used primarily for system maintenance.
(1S) Commands used for Source Code Control.
(1X) Games and useful commands.

SEE ALSO

How to Get Started, at the front of this volume.

DIAGNOSTICS

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

NAME

abort – remove previously queued line printer jobs

SYNOPSIS

abort type item [item]

DESCRIPTION

Abort removes a set of line printer requests from their respective queues. *Type* (either **user**, **printer**, or **job**) indicates the type of *items* which follow. At least one *item* is required.

In the case of *type* **user**, the *items* are login ids of users with queued jobs. For each specified user, *abort* removes all of the user's line printer requests, regardless of the queue in which the request resides. If the *type* is **printer**, then each *item* is a printer name, and *abort* terminates the currently printing job on each of the named printers. For *type* **job**, each of the specified jobs is removed from the queue.

SEE ALSO

hold(1), init(1), lpr(1), release(1), restrain(1), start(1)

NAME

ac − login accounting

SYNOPSIS

ac [**−w** wtmp] [**−p**] [**−d**] people

DESCRIPTION

Ac produces a printout giving connect time for each user who has logged in during the life of the current *wtmp* file. A total is also produced. The **−w** option is used to specify an alternate *wtmp* file. The **−p** option prints individual totals; without this option, only totals are printed. The **−d** option causes a printout for each midnight to midnight period. Any *people* after **−p** will limit the printout to only the specified login names. If no *wtmp* file is given, **/etc/wtmp** is used.

The accounting file **/etc/wtmp** is maintained by *init, date,* and *login*. Neither of these programs creates the file, so if it does not exist no connect-time accounting is done. To start accounting, it should be created with length 0. On the other hand if the file is left undisturbed it will grow without bound, so periodically any information desired should be collected and the file truncated.

FILES

/etc/wtmp

SEE ALSO

date(1), init(1M), login(1), wtmp(5)

NAME

accton − turn accounting on/off

SYNOPSIS

/etc/accton [file]

DESCRIPTION

Accton is used to turn system level accounting on and off. If the *file* argument is given, accounting is turned on and accounting information for each process is placed at the end of *file*. If the *file* argument is not given, accounting is turned off.

This command is effective only if system accounting is enabled at system generation time by specifying the parameter SYSACCT in **/usr/include/sys/param.h**.

FILES

/usr/include/sys/param.h.

SEE ALSO

sa(1), acct(2)

NAME

adb – debugger

SYNOPSIS

adb [–**w**] [objfil [corfil]]

DESCRIPTION

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

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

Requests to *adb* are read from the standard input and responses are written to the standard output. If the –**w** flag is present then both *objfil* and *corfil* are created if necessary and opened for reading and writing so that files can be modified using *adb*. *Adb* ignores QUIT; INTERRUPT causes return to the next *adb* command.

In general requests to *adb* are of the form

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

If *address* is present then *dot* is set to *address.* Initially *dot* is set to 0. For most commands *count* specifies how many times the command will be executed. The default *count* is 1. *Address* and *count* are expressions.

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

EXPRESSIONS

. The value of *dot.*

\+ The value of *dot* incremented by the current increment.

^ The value of *dot* decremented by the current increment.

" The last *address* typed.

integer An octal number if *integer* begins with a 0; a hexadecimal number if preceded by # or **0x** otherwise a decimal number.

integer.fraction
A 32 bit floating point number.

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

< name
The value of *name*, which is either a variable name or a register name. *Adb* maintains a number of variables (see VARIABLES) named by single letters or digits. If *name* is a register name then the value of the register is obtained from the system header in *corfil*. The register names are **r0 ... r5 sp pc ps.**

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

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

routine.name

The address of the variable *name* in the specified C routine. Both *routine* and *name* are *symbols*. If *name* is omitted the value is the address of the most recently activated C stack frame corresponding to *routine*.

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

Monadic operators:

**exp* The contents of the location addressed by *exp* in *corfil*.

@exp The contents of the location addressed by *exp* in *objfil*.

−exp Integer negation.

~exp Bitwise complement.

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

e1+e2 Integer addition.

e1−e2 Integer subtraction.

*e1*e2* Integer multiplication.

e1%e2 Integer division.

e1&e2 Bitwise conjunction.

e1|e2 Bitwise disjunction.

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

COMMANDS

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

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

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

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

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

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

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

s *n* Print the addressed characters until a zero character is reached.

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

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

i n Print as PDP11 instructions. *n* is the number of bytes occupied by the instruction. This style of printing causes variables 1 and 2 to be set to the offset parts of the source and destination respectively.

a 0 Print the value of *dot* in symbolic form. Symbols are checked to ensure that they have an appropriate type as indicated below.

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

p 2 Print the addressed value in symbolic form using the same rules for symbol lookup as **a**.

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

r 0 Print a space.

n 0 Print a new-line.

"..." 0 Print the enclosed string.

^ *Dot* is decremented by the current increment. Nothing is printed.

\+ *Dot* is incremented by 1. Nothing is printed.

\− *Dot* is decremented by 1. Nothing is printed.

new-line

Repeat the previous command with a *count* of 1.

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

Words starting at *dot* are masked with *mask* and compared with *value* until a match is found. If L is used then the match is for 4 bytes at a time instead of 2. If no match is found then *dot* is unchanged; otherwise *dot* is set to the matched location. If *mask* is omitted then −1 is used.

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

Write the 2-byte *value* into the addressed location. If the command is W, write 4 bytes. Odd addresses are not allowed when writing to the subprocess address space.

[?/]**m** *b1 e1 f1*[?/]

New values for (*b1, e1, f1*) are recorded. If less than three expressions are given then the remaining map parameters are left unchanged. If the ? or / is followed by * then the second segment (*b2, e2, f2*) of the mapping is changed. If the list is terminated by ? or / then the file (*objfil* or *corfil* respectively) is used for subsequent requests. (So that, for example, '/m?' will cause / to refer to *objfil*.)

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

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

$*modifier*

Miscellaneous commands. The available *modifiers* are:

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

>*f* Send output to the file *f*, which is created if it does not exist.

r Print the general registers and the instruction addressed by **pc**. *Dot* is set to **pc**.

f Print the floating registers in single or double length. If the floating point

status of **ps** is set to double (0200 bit) then double length is used anyway.

b Print all breakpoints and their associated counts and commands.

a ALGOL 68 stack backtrace. If *address* is given then it is taken to be the address of the current frame (instead of **r4**). If *count* is given then only the first *count* frames are printed.

c C stack backtrace. If *address* is given then it is taken as the address of the current frame (instead of **r5**). If C is used then the names and (16 bit) values of all automatic and static variables are printed for each active function. If *count* is given then only the first *count* frames are printed.

e The names and values of external variables are printed.

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

s Set the limit for symbol matches to *address* (default 255).

o All integers input are regarded as octal.

d Reset integer input as described in EXPRESSIONS.

q Exit from *adb*.

v Print all non zero variables in octal.

m Print the address map.

:*modifier*

Manage a subprocess. Available modifiers are:

b*c* Set breakpoint at *address*. The breakpoint is executed *count*−1 times before causing a stop. Each time the breakpoint is encountered the command *c* is executed. If this command sets *dot* to zero then the breakpoint causes a stop.

d Delete breakpoint at *address*.

r Run *objfil* as a subprocess. If *address* is given explicitly then the program is entered at this point; otherwise the program is entered at its standard entry point. *count* specifies how many breakpoints are to be ignored before stopping. Arguments to the subprocess may be supplied on the same line as the command. An argument starting with < or > causes the standard input or output to be established for the command. All signals are turned on on entry to the subprocess.

c*s* The subprocess is continued with signal *s* (see *signal*(2)). If *address* is given then the subprocess is continued at this address. If no signal is specified then the signal that caused the subprocess to stop is sent. Breakpoint skipping is the same as for **r**.

s*s* As for **c** except that the subprocess is single stepped *count* times. If there is no current subprocess then *objfil* is run as a subprocess as for **r**. In this case no signal can be sent; the remainder of the line is treated as arguments to the subprocess.

k The current subprocess, if any, is terminated.

VARIABLES

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

0 The last value printed.

1 The last offset part of an instruction source.

2 The previous value of variable 1.

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

b The base address of the data segment.

d	The data segment size.
e	The entry point.
m	The 'magic' number (0405, 0407, 0410 or 0411).
s	The stack segment size.
t	The text segment size.

ADDRESSES

The address in a file associated with a written address is determined by a mapping associated with that file. Each mapping is represented by two triples (*b1, e1, f1*) and (*b2, e2, f2*). The *file address* corresponding to a written *address* is calculated as follows.

$$b1 \leqslant address < e1 \Rightarrow file\ address = address + f1 - b1,\ \text{otherwise,}$$

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

otherwise, the requested *address* is not legal. In some cases (e.g. for programs with separated I and D space) the two segments for a file may overlap. If a ? or / is followed by an * then only the second triple is used.

The initial setting of both mappings is suitable for normal **a.out** and **core** files. If either file is not of the kind expected then, for that file, *b1* is set to 0, *e1* is set to the maximum file size and *f1* is set to 0; in this way the whole file can be examined with no address translation.

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

FILES

a.out
core

SEE ALSO

ptrace(2), a.out(5), core(5)

DIAGNOSTICS

"Bad core magic number" when the magic number of the *corfil* does not match that of *objfil*. This message is expected when debugging a unix crash dump tape. "Adb" appears when there is no current command or format.

Comments about inaccessible files, syntax errors, abnormal termination of commands, etc.

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

BUGS

A breakpoint set at the entry point is not effective on initial entry to the program.
When single stepping, system calls do not count as an executed instruction.
Local variables whose names are the same as an external variable may foul up the accessing of the external.

NAME

addsccs – add SCCS keywords to a file

SYNOPSIS

addsccs [**w**] file [file ...]

DESCRIPTION

The *addsccs* command will add SCCS keywords (%W%) to each named file. This is accomplished by the text editor *ed*(1). If one of the named files is "w" then this is taken as a flag and forces *addsccs* to write out the results of the edit commands into each named file. Otherwise, *addsccs* just edits the file, adds the keywords, prints the addition, and exits the editor without writing (this is useful to find out if *addsccs* knows about a particular suffix).

Addsccs knows about the following types of files and the correct syntax for adding the keywords in the form of a comment:

*.[chsyl], *.mk, *.sh

The type of file is determined by its suffix.

Addsccs is used by the *gadd*(1S) command to automatically add the SCCS keyword string when entering new files into the SCCS form.

SEE ALSO

ed(1), gadd(1S), sccsfile(5)

DIAGNOSTICS

All diagnostics are printed on file descriptor 2.

BUGS

There is no way to specify the type of file other than the suffix. Also, the new shell comment syntax is not used; rather the old form of colon and quoted string.

NAME

admin − administer SCCS files

SYNOPSIS

admin [−n] [−i[name] [−rrel]] [−t[name]] [−fadd-flag[flag-val]] ... [−ddelete-flag] ... [−aadd-login] @[−aadd-login] ... [−eerase-login] ... [−h] [−z] **name ...**

DESCRIPTION

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

If a directory is named, *admin* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname 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, but the effects of any keyletter argument other than **i** and **r** apply independently to each named file.

−n This argument indicates that new files are to be created. This argument must be specified when creating new SCCS files. The **i** argument implies an **n** argument.

−i The name of a file from which the text of an initial delta is to be taken. If this argument is supplied, but the file name is omitted, the text is obtained by reading the standard input until an end-of-file is encountered. If this argument is omitted, and the *admin* command creates one or more SCCS files; then their initial deltas must be inserted in the normal manner, using *get*(1S) and*delta*(1S). Only one SCCS fiel may be created by an *admin* command on which the *i* argument is supplied.

−r The release into which the initial delta will be inserted. This argument may only be supplied if the *i* argument is also supplied. If this argument is omitted, the initial delta will be inserted into release 1. The level of the initial delta will always be 1.

−t The name of a file from which descriptive text for the SCCS file is to be taken. If this argument is supplied and *admin* is creating a new SCCS file, the descriptive-text filename must also be supplied. In the case of existing SCCS files, if this argument is supplied but the file name is omitted, the descriptive text (if any) currently in the SCCS file will be removed. If the file name is supplied, the text in the file named will replace the descriptive text (if any) currently in the SCCS file.

−f This argument specifies a flag, and, possibly, a value for the flag, to be added to the SCCS file. Several **f** arguments may be supplied on a single *admin* command. The allowable flags and their values are as follows:

b The presence of this flag indicates that the use of the **b** argument on a *get* command will cause a branch to be taken in the delta tree.

cceil The 'ceiling:' the highest release (less than or equal to 9999) which may be specified by the **r** argument on a *get* with an **e** argument. If this flag is not specified, the ceiling is 9999.

dSID The default SID to be used on a *get* when the **r** argument is not supplied.

ffloor The 'floor:' the lowest release (greater than 0) which may be specified by the **r** argument on a *get* with an **e** argument. If this flag is not specified, the floor is 1.

i The presence of this flag causes the 'No id keywords (ge6)' message issued by *get* or *delta* to be treated as a fatal error. In the absence of this flag, the message is only a warning.

n The presence of this flag indicates that *delta* is to create a 'null' delta (a delta that applies *no* changes) in each of those releases (if any) beitn skipped when a delta is made in a *new* release (e.g., in making delta 5.1 after delta 2.7, releases 3 and 4 are skipped). The null deltas serve as 'place holders' so that branch deltas may later be created in *any* release desired. The absence of this flag causes skipped releases to be completely empty, preventing branch deltas from being created in them.

mmod This flag specifies the module name of the SCCS file. Its value will be used by *get* as the replacement for the %M% keyword.

ttype This flag specifies the type of the module. Its value will be used by *get* as a replacement for the %Y% keyword.

v[pgm] The presence of this flag indicates that *delta* is to promt for MR numbers in addition to comments. If the optional value of this flag is present, it specifies the name of an MR number validity checking program.

−d This argument specifies a flag to be completely removed from an SCCS file. This argument may only be specified when processing existing SCCS files. Several **d** arguments may be supplied on a single *admin* command. See the **f** argument for the allowable flags.

−a A login name to be added to the list of logins which may add deltas. Several **a** arguments may be supplied on a single *admin* command. As many logins as desired may be on the list simultaneously. If the list of logins is empty, then anyone may add deltas.

−e A login name to be erased from the list of logins. Several **e** arguments may be supplied on a single *admin* command.

−h This argument provides a convenient mechanism for checking for corrupted files. With this argument, *admin* will check that the sum of all the characters in the SCCS file (the check-sum) agrees with the sum which is stored in the first line of the file. If the sums are not in agreement a 'corrupted file' message will be produced. This argument inhibits writing on the file, so that it will nullify the effect of any other arguments supplied, and is, therefore, only meaningful when processing existing files.

−z This argument will cause *admin* to ignore any discrepancy in the check-sum of the SCCS file (see **h** argument), and to replace it with the new one. (The same effect may be had by the first editing the SCCS file with *ed*(1) in order to replace the five-character check-sum in the first line of the file with five zeroes. A subsequent invocation of an SCCS command which modifies the file (e.g., *admin*, *delta*), will cause check-sum validation to be by-passed, and a new check-sum to be computed). The purpose of this is to correct the check-sum in those files which may have been edited by the user. Note that use of this argument on a truly corrupted file will prevent future detection of the corruption.

FILES

The last component of all SCCS file names must be of the form '.s.modulename'. New SCCS files are given mode 444. 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 (see *get*(1), created with mode 444 if the *admin* command is creating a new SCCS file, or with the same mode as the SCCS file if it exists. After successful execution of *admin*, The SCCS file will be deleted, if it exists, and the x-file will be renamed with the name of the SCCS file. This ensures that changes will be made to the SCCS file onlyif no errors occurred.

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

If it should be necessary to patch an SCCS file for any reason, the mode may be changed to 644 by the owner, and then the owner may edit the file at will with *ed*(1).

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

SEE ALSO

ed(1), delta(1S), get(1S), help(1S), prt(1S), what(1S) sccsfile(5)
SCCSIPWB User's Manual by L. E. Bonanni and A. L. Glasser.

DIAGNOSTICS

Use *help*(1S) for explanations.

NAME

ar − archive and library maintainer

SYNOPSIS

ar key [posname] afile name ...

DESCRIPTION

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

Key is one character from the set **drqtpmx,** optionally concatenated with one or more of **vuaibcl.** *Afile* is the archive file. The *names* are constituent files in the archive file. The meanings of the *key* characters are:

d Delete the named files from the archive file.

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

q Quickly append the named files to the end of the archive file. Optional positioning characters are invalid. The command does not check whether the added members are already in the archive. Useful only to avoid quadratic behavior when creating a large archive piece-by-piece.

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

p Print the named files in the archive.

m Move the named files to the end of the archive. If a positioning character is present, then the *posname* argument must be present and, as in **r,** specifies where the files are to be moved.

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

v Verbose. Under the verbose option, *ar* gives a file-by-file description of the making of a new archive file from the old archive and the constituent files. When used with **t,** it gives a long listing of all information about the files. When used with **x,** it precedes each file with a name.

c Create. Normally *ar* will create *afile* when it needs to. The create option suppresses the normal message that is produced when *afile* is created.

l Local. Normally *ar* places its temporary files in the directory **/tmp**. This option causes them to be placed in the local directory.

FILES

/tmp/v* temporaries

SEE ALSO

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

BUGS

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

NAME

arcv – convert archives to new format

SYNOPSIS

arcv file ...

DESCRIPTION

Arcv converts archive files (see *ar*(1), *ar*(5)) from 6th edition to 7th edition format. The conversion is done in place, and the command refuses to alter a file not in old archive format.

Old archives are marked with a magic number of 0177555 at the start; new archives have 0177545.

FILES

/tmp/v*, temporary copy

SEE ALSO

ar(1), ar(5)

NAME

as − assembler

SYNOPSIS

as [−] [−**o** objfile] file ...

DESCRIPTION

As assembles the concatenation of the named files. If the optional first argument − is used, all undefined symbols in the assembly are treated as global.

The output of the assembly is left on the file *objfile;* if that is omitted, **a.out** is used. It is executable if no errors occurred during the assembly, and if there were no unresolved external references.

FILES

/lib/as2	pass 2 of the assembler
/tmp/atm[1-3]?	temporary
a.out	object

SEE ALSO

ld(1), nm(1), adb(1), a.out(5)
UNIX Assembler Manual by D. M. Ritchie

DIAGNOSTICS

When an input file cannot be read, its name followed by a question mark is typed and assembly ceases. When syntactic or semantic errors occur, a single-character diagnostic is typed out together with the line number and the file name in which it occurred. Errors in pass 1 cause cancellation of pass 2. The possible errors are:

)	Parentheses error
]	Parentheses error
<	String not terminated properly
*	Indirection used illegally
.	Illegal assignment to .
a	Error in address
b	Branch instruction is odd or too remote
e	Error in expression
f	Error in local (**f** or **b**) type symbol
g	Garbage (unknown) character
i	End of file inside an *.if*
m	Multiply-defined symbol as label
o	Word quantity assembled at odd address
p	. different in pass 1 and 2
r	Relocation error
u	Undefined symbol
x	Syntax error

BUGS

Syntax errors can cause incorrect line numbers in following diagnostics.

NAME

at − execute commands at a later time

SYNOPSIS

at time [day] [file]

DESCRIPTION

At squirrels away a copy of the named *file* (standard input default) to be used as input to *sh*(1) at a specified later time. A *cd*(1) command to the current directory is inserted at the beginning, followed by assignments to all environment variables. When the script is run, it uses the user and group ID of the creator of the copy file.

The *time* is 1 to 4 digits, with an optional following 'A', 'P', 'N' or 'M' for AM, PM, noon or midnight. One and two digit numbers are taken to be hours, three and four digits to be hours and minutes. If no letters follow the digits, a 24 hour clock time is understood.

The optional *day* is either (1) a month name followed by a day number, or (2) a day of the week; if the word 'week' follows invocation is moved seven days further off. Names of months and days may be recognizably truncated. Examples of legitimate commands are

```
at 8am jan 24
at 1530 fr week
```

At programs are executed by periodic execution of the command */usr/lib/atrun* from *cron*(1). The granularity of *at* depends upon how often *atrun* is executed.

Standard output or error output is lost unless redirected.

FILES

/usr/spool/at/yy.ddd.hhhh.uu
activity to be performed at hour *hhhh* of year day *ddd* of year *yy*. *uu* is a unique number.
/usr/spool/at/lasttimedone contains *hhhh* for last hour of activity.
/usr/spool/at/past directory of activities now in progress
/usr/lib/atrun program that executes activities that are due

SEE ALSO

calendar(1), cron(1)

DIAGNOSTICS

Complains about various syntax errors and times out of range.

BUGS

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

NAME

atgex — convert ascii file to GEX format

SYNOPSIS

atgex

DESCRIPTION

Atgex is a filter which converts an ascii file (i.e. its standard input) to a *gex* type file (i.e. its standard output) which can then be edited with *gex*(1G).

SEE ALSO

gex(1G)

AUTHOR

D. J. Jackowski

BUGS

This is a half baked program. It should be reworked to give the user control over output options. Right now it only defaults.

NAME

awk − pattern scanning and processing language

SYNOPSIS

awk [−F*c*] [prog] [-f*file*] ...

DESCRIPTION

Awk scans each input *file* for lines that match any of a set of patterns specified in *prog*. With each pattern in *prog* there can be an associated action that will be performed when a line of a *file* matches the pattern. The set of patterns may appear literally as *prog*, or in a file specified as −f *file*.

Files are read in order; if there are no files, the standard input is read. The file name − means the standard input. Each line is matched against the pattern portion of every pattern-action statement; the associated action is performed for each matched pattern.

An input line is made up of fields separated by white space. (This default can be changed by using FS, *vide infra*.) The fields are denoted **$1**, **$2**, ... ; **$0** refers to the entire line.

A pattern-action statement has the form

pattern { action }

A missing action means print the line; a missing pattern always matches.

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

```
if ( conditional ) statement
while ( conditional ) statement
for ( expression ; conditional ; expression ) statement
{ [ statement ] ... }
variable = expression
variable ++, variable --
print [ expression-list ] [ >expression ]
printf format [ , expression-list ] [ >expression ]
next    # skip remaining patterns on this input line
exit    # skip the rest of the input
```

Statements are terminated by semicolons, newlines or right braces. An empty expression-list stands for the whole line. Expressions take on string or numeric values as appropriate, and are built using the operators +, −, *, /, %, and concatenation (indicated by a blank). The C operators ++ and −− may be used to increment variables, but only in statement contexts, not as expressions. Variables may be scalars, array elements (denoted x[i]) or fields. Variables are initialized to the null string. Array subscripts may be any string, not necessarily numeric; this allows for a form of associative memory. String constants are quoted (").

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

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

Patterns are arbitrary Boolean combinations (, ||, &&, and parentheses) of regular expressions and relational expressions. Regular expressions must be surrounded by slashes and are as in *egrep* (see *grep*(1)). Isolated regular expressions in a pattern apply to the entire line. Regular expressions may also occur in relational expressions.

A pattern may consist of two patterns separated by a comma; in this case, the action is performed for all lines between an occurrence of the first pattern and the next occurrence of the second.

A relational expression is one of the following:

expression matchop regular-expression
expression relop expression

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

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

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

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

or by using the −F*c* option.

Other variable names with special meanings include NF, the number of fields in the current record; NR, the ordinal number of the current record; FILENAME, the name of the current input file; OFS, the output field separator (default blank); ORS, the output record separator (default newline); and OFMT, the output format for numbers (default "%.6g").

EXAMPLES

Print lines longer than 72 characters:

```
length > 72
```

Print first two fields in opposite order:

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

Add up first column, print sum and average:

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

Print fields in reverse order:

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

Print all lines between start/stop pairs:

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

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

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

SEE ALSO

lex(1), sed(1), grep(1)

A. V. Aho, B. W. Kernighan, P. J. Weinberger, *Awk − a pattern scanning and processing language* Bell Laboratories CSTR #68, 1978.

BUGS

Input white space is not preserved on output if fields are involved.

There are no explicit conversions between numbers and strings. To force an expression to be treated as a number add 0 to it; to force it to be treated as a string concatenate the null string ("") to it.

NAME

banner – make headlines

SYNOPSIS

banner string

DESCRIPTION

Banner prints its argument in large letters on the standard output.

NAME

basename, dirname − deliver portions of pathnames

SYNOPSIS

basename string [suffix]
dirname string

DESCRIPTION

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

Dirname delivers all but the last level of the pathname in *string*.

EXAMPLES

This shell procedure invoked with the argument **/usr/src/cmd/cat.c** compiles the named file and moves the output to **cat** in the current directory:

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

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

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

SEE ALSO

sh(1)

NAME

bc − arbitrary-precision arithmetic language

SYNOPSIS

bc [−c] [−l] [file ...]

DESCRIPTION

Bc is an interactive processor for a language which resembles C but provides unlimited precision arithmetic. It takes input from any files given, then reads the standard input. The −l argument stands for the name of an arbitrary precision math library. The syntax for *bc* programs is as follows; L means letter a−z, E means expression, S means statement.

Comments

are enclosed in /* and */.

Names

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

Other operands

arbitrarily long numbers with optional sign and decimal point.
(E)
sqrt (E)
length (E) number of significant decimal digits
scale (E) number of digits right of decimal point
L (E , ... , E)

Operators

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

Statements

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

Function definitions

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

Functions in −l math library

s(x) sine
c(x) cosine
e(x) exponential
l(x) log
a(x) arctangent
j(n,x) Bessel function

All function arguments are passed by value.

The value of a statement that is an expression is printed unless the main operator is an assignment. Either semicolons or new-lines may separate statements. Assignment to *scale* influences the number of digits to be retained on arithmetic operations in the manner of *dc*(1). Assignments to *ibase* or *obase* set the input and output number radix respectively.

The same letter may be used as an array, a function, and a simple variable simultaneously. All variables are global to the program. 'Auto' variables are pushed down during function calls. When using arrays as function arguments or defining them as automatic variables empty square brackets must follow the array name.

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

EXAMPLE

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

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

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

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

FILES

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

SEE ALSO

dc(1)

L. L. Cherry and R. Morris, *BC − An arbitrary precision desk-calculator language*

BUGS

No &&, || yet.

For statement must have all three E's.

Quit is interpreted when read, not when executed.

NAME

bclk, setbclk − reads and sets the battery clock

SYNOPSIS

bclk

setbclk mon dd yy hh:mm

DESCRIPTION

Bclk reads the battery clock (a TCU-100 battery clock) and reports the current time. The time reported should be the current "standard" time. That is, during daylight savings time periods, the battery clock should be one hour behind wall clock time.

Setbclk resets the battery clock. To use **setbclk** the user must have write privledges for /dev/mem. The date is entered as current wall time, a three letter **mon,** a decimal day, year, and hours : minutes. When **setbclk** has successfully reset the battery clock, the time will be reported as it is for **bclk.**

FILES

/dev/mem

SEE ALSO

date(1)

NAME

bd − binary dump of a file

SYNOPSIS

bd [**−rd**] file

DESCRIPTION

Bd dumps *file* to the standard output in four sections with eight decimal bytes per row. The first column is the octal address in the file; the next four columns represent the octal dump of four words; the next eight columns represent the octal dump of eight bytes; the last column consists of the eight bytes printed as *ASCII*. Bytes whose contents are not printable. *ASCII* characters are translated into dot '.'. All numbers are in octal.

Options:

−r Print a restricted ascii character set in the *ASCII* filed. The characters not printed are the following characters ' { | } .

−d Print the data in a double line format. The address is on the first line in the following format <001230>. The next line contains the octal dumps and the *ASCII*. This option is used for TTYs with short carriages.

Bd can be used with a *pipe* as a filter. If multiple files are to the dumped, the format:

cat file1 file2 | **bd** [**−rd**]

will work with or without the option flags.

SEE ALSO

od(1)

DIAGNOSTICS

Can't open input file

AUTHOR

B. C. Hoalst

NAME

bdiff − big diff

SYNOPSIS

bdiff [−s] name1 name2 [numarg]

DESCRIPTION

Bdiff is used in a manner analogous to *diff*(1) to find which lines must be changed in two files to bring them into agreement. Its purpose is to allow processing of files which are too large for *diff*(1). *Bdiff* ignores lines common to the beginning of both files, splits the remainder of each file into *n*-line segments, and invokes *diff* upon corresponding segments. The value of *n* is 3500 by default. If the optional third argument is given, and it is numeric, it is used as the value for *n*. This is useful in those cases in which 3500-line segments are too large for *diff*, causing it to fail. If *name1* (*name2*) is "−", the standard input is read. The optional −s ("silent") argument specifies that no diagnostics are to be printed by *bdiff* (note, however, that this does not suppress possible exclamations by *diff*). If both optional arguments are specified, they must appear in the order indicated above. The output of *bdiff* is exactly that of *diff*, with line numbers adjusted to account for the segmenting of the files (that is, to make it look as if the files had been processed whole). Note that because of the segmenting of the files, *bdiff* does not necessarily find a smallest sufficient set of file differences.

FILES

/tmp/bd?????

SEE ALSO

diff(1)

DIAGNOSTICS

Use *help*(1S) for explanations.

NAME

bdump — read from block device

SYNOPSIS

/etc/bdump dev blockno [filename]

DESCRIPTION

Bdump seeks to the specified block on the specified device and copies one 512-byte block from the device into either the *filename* argument (if it is given), or the default file name **dtmpx** otherwise. *Bdump* prepends '/dev/' onto the front of the *dev* argument and attempts to open the device for reading. The *blockno* argument is assumed to be octal if it contains a leading zero, otherwise decimal.

FILES

dtmpx

SEE ALSO

bload(1M)

DIAGNOSTICS

dev? - The device cannot be opened.
non - numeric character -- the *blockno* argument contains a bad character
destination file? -- "dtmpx" or *filename* cannot be created in the current directory

NAME

bload − write on block device

SYNOPSIS

/etc/bload dev blockno [filename]

DESCRIPTION

Bload seeks to the specified block on the specified device and copies one 512-byte block onto the device *Bload* prepends '/dev/' onto the front of the *dev* argument and attempts to open the device for writing. The *blockno* argument is assumed to be octal if it contains a leading zero, otherwise decimal.

FILES

dtmpx

SEE ALSO

bdump(1M)

DIAGNOSTICS

dev? - The device cannot be opened.
non - numeric character -- the *blockno* argument contains a bad character
destination file? -- "dtmpx" or *filename* cannot be created in the current directory

NAME

bs − a compiler/interpreter for modest-sized programs

SYNOPSIS

bs [file [arg ...]]

DESCRIPTION

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

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

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

```
statement
label statement
```

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

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

Statement Syntax:

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

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

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

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

include expression
: The expression should evaluate to a file name. The file must contain *bs* source statements. *Include* statements may not be nested.

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

dump
: The name and current value of every non-local variable is printed. After an error or interrupt, the number of the last statement and (possibly) the user-function trace are displayed.

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

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

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

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

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

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

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

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

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

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

will cause *bs* to terminate.

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

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

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

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

! shell command
An immediate escape to the Shell.

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

Expression Syntax:

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

name ([expression [, expression] ...])
Functions can be called by a name followed by the arguments in parentheses separated by commas. Except for built-in functions (listed below), the name must be defined with a *fun* statement. Arguments to functions are passed by value.

name [expression [, expression] ...]
Each expression is truncated to an integer and used as a specifier for the name. The resulting array reference is syntactically identical to a name. **a[1,2]** is the same as **a[1][2].** The truncated expressions are restricted to values between 0 and 32767.

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

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

(expression)
Parentheses are used to alter normal order of evaluation.

(expression, expression [, expression ...]) [expression]
The bracketed expression is used as a subscript to select a comma-separated expression from

the parenthesized list. List elements are numbered from the left, starting at zero. The expression

(False, True)[a == b]

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

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

− expression
The result is the negation of the expression.

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

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

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

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

Binary Operators (in increasing precedence):

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

_
_ (underscore) is the concatenation operator.

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

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

+ −
Add and subtract.

* / %
Multiply, divide, and remainder.

^

Exponentiation.

Built-in Functions:

Dealing with arguments

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

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

Mathematical

abs(x)
is the absolute value of x.

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

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

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

exp(x)
is the exponential function of x.

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

log(x)
is the natural logarithm of x.

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

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

sqrt(x)
is the square root of x.

String operations

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

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

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

trans(s, f, t)
Translates characters of the source s from matching characters in f to a character in the

same position in *t*. Source characters that do not appear in *f* are copied to the result. If the string *f* is longer than *t*, source characters that match in the excess portion of *f* do not appear in the result.

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

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

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

File handling

open(name, file, function)
close(name)
The *name* argument must be a *bs* variable name (passed as a string). For the *open*, the *file* argument may be **1)** a 0 (zero), 1, or 2 representing standard input, output, or error output, respectively, **2)** a string representing a file name, or **3)** a string beginning with an ! representing a command to be executed (via *sh* −*c*). The *function* argument must be either **r** (read), **w** (write), **W** (write without new-line), or **a** (append). After a *close*, the *name* reverts to being an ordinary variable. The initial associations are:

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

Examples are given in the following section.

access(s, m)
executes *access*(2).

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

Odds and Ends

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

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

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

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

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

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

plot(request, args)

The *plot* function produces output on devices recognized by *plot*(1G). The *requests* are as follows.

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

Some requests do not apply to all plotters. All requests except zero and twelve are implemented by piping characters to *plot*(1G). See *plot*(5) for more details.

last()

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

PROGRAMMING TIPS

Using *bs* as a calculator:

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

```
exit
```

The outline of a typical *bs* program:

```
# Initialize things:
var1 = 1
open("read", "infile", "r")
...
# Compute:
while ?(str = read)
        ...
next
# Clean up:
close("read")
...
# Last statement executed (exit or stop):
exit
# Last input line:
run
```

Input/Output examples:

1)
```
# Copy "oldfile" to "newfile".
open("read", "oldfile", "r")
open("write", "newfile", "w")
...
while ?(write = read)
...
# Close "read" and "write"
close("read")
close("write")
```

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

SEE ALSO

ed(1), plot(1G), sh(1), access(2), printf(3S), stdio(3S), Section 3 of this volume for further description of the mathematical functions (pow(3M) is used for exponentiation), plot(5). *Bs* uses the Standard Input/Output package.

BUGS

There are built-in design limits. *Bs* source programs are restricted to fewer than 250 lines and fewer than 250 variables (the *name* of an array counts as a variable, as does each dimension and each referenced element).

All names (labels, variables, functions, statement keywords) are internally truncated to six characters.

NAME

cal – print calendar

SYNOPSIS

cal [month] year

DESCRIPTION

Cal prints a calendar for the specified year. If a month is also specified, a calendar just for that month is printed. *Year* can be between 1 and 9999. The *month* is a number between 1 and 12. The calendar produced is that for England and her colonies.

Try September 1752.

BUGS

The year is always considered to start in January even though this is historically naive.
Beware that 'cal 78' refers to the early Christian era, not the 20th century.

NAME

calendar − reminder service

SYNOPSIS

calendar [−]

DESCRIPTION

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

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

FILES

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

SEE ALSO

cron(1), mail(1)

BUGS

Your calendar must be public information for you to get reminder service.
Calendar's extended idea of 'tomorrow' doesn't account for holidays.
Calendar ignores lines that begin with tabs.

NAME

cat − concatenate and print files

SYNOPSIS

cat [**−u**] [**−s**] file ...

DESCRIPTION

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

```
cat file
```

prints the file, and:

```
cat file1 file2 >file3
```

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

If no input file is given, or if the argument **−** is encountered, *cat* reads from the standard input file. Output is buffered in 512-byte blocks unless the **−u** option is specified. The **−s** option makes *cat* silent about non-existent files. No input file may be the same as the output file unless it is a special file.

SEE ALSO

cp(1), pr(1).

NAME

cb − C program beautifier

SYNOPSIS

cb [file]

DESCRIPTION

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

NAME

cc, pcc − C compiler

SYNOPSIS

cc [option] ... file ...
pcc [option] ... file ...

DESCRIPTION

Cc is the standard UNIX C compiler. Other versions may exist with a single letter prefix; in particular, *occ* is supplied as the previous C compiler, and *ncc* may be present as a new, experimental C compiler. *Pcc* is the portable version for a PDP-11 machine. They accept several types of arguments:

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

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

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

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

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

−O Invoke an object-code optimizer.

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

−E Run only the macro preprocessor on the named C programs, and send the result to the standard output.

−P Run only the macro preprocessor on the named C programs, and leave the result on corresponding files suffixed **.i**.

−C Comments are not stripped by the macro preprocessor.

−D*name*=*def*
−D*name*

Define the *name* to the preprocessor, as if by **#define**. If no definition is given, the name is defined as 1.

−U*name*

Remove any initial definition of *name*.

−I*dir* Change the algorithm for searching for **#include** files whose names do not begin with / to look in *dir* before looking in the directories on the standard list. Thus, **#include** files whose names are enclosed in "" will be searched for first in the directory of the *file* argument, then in directories named in **−I** options, and last in directories on a standard list. For **#include** files whose names are enclosed in <>, the directory of the *file* argument is not searched. The current standard list consists of **/usr/include**. If *−I* is used with no *dir* argument, search of the standard directory list is suppressed.

−B Instead of using the standard compiler, use a "backup" compiler (providing that the

system administrator has provided one). This option is identical to using the *occ* command.

-t[p012] Find only the designated compiler passes in the files whose names are **/sys/c/c[012]** or **/sys/c/cpp**. Used for testing compiler changes.

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

FILES

file.c	input file
file.o	object file
a.out	loaded output
/tmp/ctm*	temporary
/lib/cpp	preprocessor
/lib/c[01]	compiler,*cc*
/lib/oc[012]	backup compiler,*cc*
/lib/ocpp	backup preprocessor
/lib/c2	optional optimizer
/usr/lib/comp	compiler,*pcc*
/lib/crt0.o	runtime startoff
/lib/mcrt0.o	startoff for profiling
/lib/libc.a	standard library, see (3)
/usr/include	standard directory for **#include** files

SEE ALSO

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, NY, 1978.
B. W. Kernighan, *Programming in C—A Tutorial.*
D. M. Ritchie, *C Reference Manual.*
adb(1), ld(1), prof(1), monitor(3C).

DIAGNOSTICS

The diagnostics produced by C itself are intended to be self-explanatory. Occasional messages may be produced by the assembler or loader. Of these, the most mystifying are from the assembler, in particular **m**, which means a multiply-defined external symbol (function or data).

NAME

chdir, cd – change working directory

SYNOPSIS

chdir directory
cd directory

DESCRIPTION

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

Because a new process is created to execute each command, *chdir* would be ineffective if it were written as a normal command. It is therefore recognized and executed by the Shell.

SEE ALSO

sh(1), pwd(1)

NAME

check − file system consistency check and repair

SYNOPSIS

check [**−syna**] [filesystem] ...

DESCRIPTION

Check audits UNIX file systems for consistency and corrects any discrepancies. Since these corrections will, in general, result in a loss of data, the program will request operator concurence for each such action. All questions should be answered by typing "**yes**" or "**no**", followed by a line feed. Typing "**yes**" will cause the correction to take place. However, if the program does not have write permission on the file system or the no option, **−n**, is on, then all questions will automatically be answered "**no**". Alternatively the yes option, **−y**, will cause all questions to be answered "**yes**". If the **-a** option is supplied then check will attempt an "automatic" check. In this case it will answer "yes" only to correct nonsevere errors. In all other cases check will loop and print a message requesting manual intervention.

The program consists of six separate phases. Some phases are skipped if they are not needed. In phase one, *check* examines all block pointers in all files, checking for pointers which are outside of the file system (BAD) and for blocks which appear in more than one file (DUP). A table is made of all DUP blocks and all defective files are marked for clearing. Each error is printed, but no correction takes place in this phase.

The second phase is run only if DUP blocks were found in phase one. This phase finds the rest of the DUP blocks; marking each for clearing.

The third phase checks the directory structure of the file system. This is done by descending the directory tree, examining each entry. A count is kept of the number of references to each file. If any entry refers to an unallocated file, a file marked for clearing, or a file number outside the file system; then the entry is printed and, if the operator agrees, it is removed. Refusing to remove an entry to a marked file will clear the mark, preserving the file and its subsequent entries.

In phase four all marked or unreferenced files are listed. With concurence from the operator, each of these files is then cleared. In addition, any file whose link count does not agree with the number of references is listed, and, if agreed, the link count is adjusted.

If the salvage option, **−s**, is on, then phase five is skipped. Otherwise, *check* examines the free list. If any blocks are found which are outside the file system or which have been previously encountered in a file or elsewhere in the free list, then the list is pronounced BAD and a salvage is called for. Agreement will set the salvage option and proceed to the next phase. If there are no defects in the free list and all blocks are accounted for, the check is finished. Otherwise, the number of missing blocks (i.e. in neither a file nor the free list) is printed and a salvage is requested.

The last phase is the salvage operation, where the free list is recreated. It is run whenever the salvage option is on or a problem has been found with the free list. Simply stated, a new free list is constructed containing all blocks not found in some file.

The system responses are in general self explanatory and follow the sequence described above. In the specification that follows, the following notation will be used:

<b> block number

<i> inode number

<fname>
file pathname

<n> positive integer

<c> option character

Check begins with the following output:

<*filesystem*>{(NO WRITE)}

Phase 1 - Check Blocks

The "(NO WRITE)" message indicates that the program does not have write permission on the file system. Therefore, subsequent corrections will be suppressed by automatically answering "**no**" to all questions. Phase one then proceeds to list any BAD or DUP blocks and their inode number, as follows:

<b> BAD I = <i>
<b> DUP I = <i>
<b> EXCESSIVE DUPS I = <i>

If too many DUPs are encountered, the program will list all blocks, but will not mark the excess DUPs for later processing. When Phase 1 is finished, Phase 2 is run if any DUPs were encountered. Otherwise, Phase 2 is skipped. This Phase will list the rest of the DUP blocks as follows:

Phase 2 - Rescan for more DUPS
<b> DUP I = <i>

Check now descends the directory tree, asking to remove any defective entries.

Phase 3 - Check Pathnames
I OUT OF RANGE I = <i> <fname> REMOVE?
UNALLOCATED I = <i> <fname> REMOVE?
BAD/DUP I = <i> <fname> REMOVE?

Unless the no option is on, the program will wait for a response of "**yes**" or "**no**" after each question. Note, a **no** answer to the BAD/DUP entry will unmark that inode for clearing. This will suppress any subsequent correction to that file.

Now *check* will clear or adjust any defective files. Again, it will wait for a "**yes**" or "**no**" response to each question. The program will also indicate whether each entry is a file or a directory.

Phase 4 - Check Reference Counts
UNREFERENCED {FILE/DIRECTORY} I = <i> CLEAR?
BAD/DUP {FILE/DIRECTORY} I = <i> CLEAR?
LINK COUNT {FILE/DIRECTORY} I = <i> ADJUST?

If the salvage option is not on, the program will now validate the free list. Otherwise, this phase is skipped. If there are any errors in the free list, it will specify them and request a salvage.

Phase 5 - Check Free List
BAD FREE LIST SALVAGE?
<n> MISSING SALVAGE?

Phase 6 is the salvage operation. It is only done if one has been requested.

Phase 6 - Salvage Free List

Finally, some totals are printed: the total number of allocated files (including directories and special files); the number of blocks in use; and the number of blocks in the free list.

<n> FILES <n> BLOCKS <n> FREE

If the filesystem has been modified, then the following message is printed and the program goes into a loop. This is only a reminder to the operator since the program can be forced to terminate with a <DEL> character.

*****BOOT UNIX(NO SYNC!)*****

FILES

/dev/rootdev default file system to be checked

SEE ALSO

sync(1M), dcheck(1M), update(1M), clri(1M), crash(6), updfs(1M), fs(5),

DIAGNOSTICS

While running, a number of errors can occur which cause the *check* program to terminate. An illegal option or the inability to open the file system are shown as:

<c> OPTION??
CAN NOT OPEN <filesystem>

An I/O error on the filesystem will also cause an error message. In this case, the operator is given the choice of exiting ("yes") or continuing ("**no**"). This error is generally a hardware error, and continuing is rarely a good idea.

CAN NOT READ <filesystem> BLOCK <b> EXIT?
CAN NOT SEEK <filesystem> BLOCK <b> EXIT?
CAN NOT WRITE <filesystem> BLOCK <b> EXIT?

NAME

chess – the game of chess

SYNOPSIS

/usr/games/chess

DESCRIPTION

Chess is a computer program that plays class D chess. Moves may be given either in standard (descriptive) notation or in algebraic notation. The symbol '**+**' is used to specify check and is not required; '**o-o**' and '**o-o-o**' specify castling. To play black, type '**first**'; to print the board, type an empty line.

Each move is echoed in the appropriate notation followed by the program's reply.

FILES

/usr/lib/book opening 'book'

DIAGNOSTICS

The most cryptic diagnostic is 'eh?' which means that the input was syntactically incorrect.

WARNING

Over-use of this program has been known to cause it to go away.

AUTHOR

K. Thompson

BUGS

Pawns may be promoted only to queens.

NAME

chghist − change the history entry of an SCCS delta

SYNOPSIS

chghist [−r*SID*] name ...

DESCRIPTION

Chghist changes the history information, for the delta specified by the SID, of each named SCCS file.

If a directory is named, *chghist* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname 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 exact permissions necessary to change the history entry of a delta are documented in the *SCCS/PWB User's Manual.* Simply stated, they are either (1) if you made a delta, you can change its history entry; or (2) if you own the file and directory you can change a history entry.

The new history is read from the standard input. If the standard input is a terminal, the program will prompt with "MRs? " (only if the file has a flag, see *admin*(1S)) and with "comments? ". If the standard input is not a terminal, no prompt(s) is (are) printed. A newline preceded by a "\" is read as a blank, and may be used to make the entering of the history more convenient. The first newline not preceded by a "\" terminates the response for the corresponding prompt.

When the history entry of a delta table record (see *prt*(1S)) is changed, all old MR entries (if any) are converted to comments, and both these and the original comments are preceded by a comment line that indicates who made the change and when it was made. The new information is entered preceding the old. No other changes are made to the delta table entry.

FILES

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

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

SEE ALSO

admin(1S), delta(1S), get(1S), help(1S), prs(1S), sccsfile(5)
Source Code Control System User's Guide by L. E. Bonanni and C. A. Salemi.

DIAGNOSTICS

Use *help*(1S) for explanations.

NAME

chkold — file system consistency chkold

SYNOPSIS

chkold [**—lsuib** [numbers]] [filesystem]

DESCRIPTION

Chkold examines a file system, builds a bit map of used blocks, and compares this bit map against the free list maintained on the file system. It also reads directories and compares the link-count in each i-node with the number of directory entries by which it is referenced. If the file system is not specified, a chkold of a default file system is performed. The normal output of *chkold* includes a report of

The number of blocks missing; i.e. not in any file nor in the free list,
The number of special files,
The total number of files,
The number of large files,
The number of directories,
The number of indirect blocks,
The number of blocks used in files,
The highest-numbered block appearing in a file,
The number of free blocks.

The **—l** flag causes *chkold* to produce as part of its output report a list of the all the path names of files on the file system. The list is in i-number order; the first name for each file gives the i-number while subsequent names (i.e. links) have the i-number suppressed. The entries "." and ".." for directories are also suppressed. If the flag as given as **—ll** , the listing will include the accessed and modified times for each file. The **—l** option supersedes **—s** .

The **—s** flag causes *chkold* to ignore the actual free list and reconstruct a new one by rewriting the super-block of the file system. The file system should be dismounted while this is done; if this is not possible (for example if the root file system has to be salvaged) care should be taken that the system is quiescent and that it is rebooted immediately afterwards so that the old, bad in-core copy of the super-block will not continue to be used. Notice also that the words in the super-block which indicate the size of the free list and of the i-list are believed. If the super-block has been curdled these words will have to be patched. The **—s** flag causes the normal output reports to be suppressed.

With the **—u** flag, *chkold* examines the directory structure for connectivity. A list of all i-node numbers that cannot be reached from the root is printed. This is exactly the list of i-nodes that should be cleared (see *clri*(1M)) after a series of incremental restores. (See the **BUGS** section of *mhrestor*(1M)). The **—u** option supersedes **—s** .

The occurrence of **i** *n* times in a flag argument **—ii...i** causes *chkold* to store away the next *n* arguments which are taken to be i-numbers. When any of these i-numbers is encountered in a directory a diagnostic is produced, as described below, which indicates among other things the entry name.

Likewise, *n* appearances of **b** in a flag like **—bb...b** cause the next *n* arguments to be taken as block numbers which are remembered; whenever any of the named blocks turns up in a file, a diagnostic is produced.

FILES

Currently, /dev/rp0 is the default file system.

SEE ALSO

check(1M), clri(1M), mhrestor(1M), sync(1M), update(1M), updfs(1M), fs(5)

DIAGNOSTICS

If a read error is encountered, the block number of the bad block is printed and *chkold* exits. "Bad freeblock" means that a block number outside the available space was encountered in the free list. "*n* dups in free" means that *n* blocks were found in the free list which duplicate blocks either in some file or in the earlier part of the free list.

An important class of diagnostics is produced by a routine which is called for each block which is encountered in an i-node corresponding to an ordinary file or directory. These have the form

b# complaint ; **i=** *i#* (*class*)

Here *b#* is the block number being considered; *complaint* is the diagnostic itself. It may be

blk if the block number was mentioned as an argument after **−b** ;

bad if the block number has a value not inside the allocatable space on the device, as indicated by the devices' super-block;

dup if the block number has already been seen in a file;

din if the block is a member of a directory, and if an entry is found therein whose i-number is outside the range of the i-list on the device, as indicated by the i-list size specified by the super-block. Unfortunately this diagnostic does not indicate the offending entry name, but since the i-number of the directory itself is given (see below) the problem can be tracked down.

The *i#* in the form above is the i-number in which the named block was found. The *class* is an indicator of what type of block was involved in the difficulty:

sdir indicates that the block is a data block in a small file;

ldir indicates that the block is a data block in a large file (the indirect block number is not available);

idir indicates that the block is an indirect block (pointing to data blocks) in a large file;

free indicates that the block was mentioned after **−b** and is free;

urk indicates a malfunction in *chkold.*

When an i-number specified after **−i** is encountered while reading a directory, a report in the form

i# **ino; i=** *d#* (*class*) *name*

where *i#* is the requested i-number. *d#* is the i-number of the directory, *class* is the class of the directory block as discussed above (virtually always "sdir") and *name* is the entry name. This diagnostic gives enough information to find a full path name for an i-number without using the **−i** option: use **−b** *n* to find an entry name and the i-number of the directory containing the reference to *n*, then recursively use **−b** on the i-number of the directory to find its name.

Another important class of file system diseases indicated by *chkold* is files for which the number of directory entries does not agree with the link-count field of the i-node. The diagnostic is hard to interpret. It has the form

i# delta

Here *i#* is the i-number affected. *Delta* is an octal number accumulated in a byte, and thus can have the value 0 through 377(8). The easiest way (short of rewriting the routine) of explaining the significance of *delta* is to describe how it is computed.

If the associated i-node is allocated (that is, has the *allocated* bit on) add 100 to *delta*. If its link-count is non-zero, add another 100 plus the link-count. Each time a directory entry specifying the associated i-number is encountered, subtract 1 from *delta*. At the end, the i-number

and *delta* are printed if *delta* is neither 0 nor 200. The first case indicates that the i-node was unallocated and no entries for it appear; the second that it was allocated and that the link-count and the number of directory entries agree.

Therefore (to explain the symptoms of the most common difficulties) *delta* = 377 (−1 in 8-bit, 2's complement octal) means that there is a directory entry for an unallocated i-node. This is somewhat serious and the entry should be found and removed forthwith. *Delta* = 201 usually means that a normal, allocated i-node has no directory entry. This difficulty is much less serious. Whatever blocks there are in the file are unavailable, but no further damage will occur if nothing is done. A *clri* followed by a *chkold* *−s* will restore the lost space at leisure.

In general, values of *delta* equal to or somewhat above 0, 100, or 200 are relatively innocuous; just below these numbers there is danger of spreading infection.

BUGS

Chkold *−l* or *−u* on large file systems takes a great deal of core.
Since *chkold* is inherently two-pass in nature, extraneous diagnostics may be produced if applied to active file systems.
It believes even preposterous super-blocks and consequently can get core images.

NAME

chmod − change mode of file

SYNOPSIS

chmod mode file ...

DESCRIPTION

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

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

A symbolic *mode* has the form:

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

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

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

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

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

Only the owner of a file (or the super-user) may change its mode.

EXAMPLES

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

```
chmod o−w file

chmod +x file
```

SEE ALSO

ls(1), chmod(2), umask(2)

NAME

chown, chgrp – change owner of group of a file

SYNOPSIS

chown owner file ...
chgrp group file ...

DESCRIPTION

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

Chgrp changes the group-ID of the *files* to *group*. The group may be either a decimal GID or a group name found in the group-ID file. You must be current owner or in proper current group to make change.

FILES

/etc/passwd
/etc/group

SEE ALSO

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

NAME

chroot – change root directory for a command

SYNOPSIS

chroot newroot command

DESCRIPTION

The given *command* is executed relative to the new root. Command may also be of form "**/bin/sh** shellfile". The meaning of any initial slashes (/) in path names is changed for the duration of *command* and any of its children to *newroot*. Furthermore, the initial working directory is *newroot*.

Notice that "chroot newroot command > x" will create the file x relative to the original root, not the new one.

This command is restricted to the super user.

The new root pathname is always absolute: even if a *chroot* is currently in effect, the *newroot* argument is relative to the real root of the file system.

SEE ALSO

chdir(2)

BUGS

One should exercise extreme caution when referencing special files in the new root file system.

NAME

clri − clear inode

SYNOPSIS

/etc/clri i-number filesystem [i-number ...]

DESCRIPTION

Clri writes zeros on the 32 bytes occupied by the i-nodes specified. If the *filesystem* argument is given, the i-node resides on the given device. The *filesystem* must be a special file name referring to a device containing a file system. After *clri* , any blocks in the affected file will show up as "missing" in a *check* of the file system.

Read and write permission is required on the specified file system device. The i-node becomes allocatable.

The primary purpose of this routine is to remove a file which for some reason appears in no directory. If it is used to zap an i-node which does appear in a directory, care should be taken to track down the entry and remove it. Otherwise, when the i-node is reallocated to some new file, the old entry will still point to that file. At that point removing the old entry will destroy the new file. The new entry will again point to an unallocated i-node, so the whole cycle is likely to be repeated again and again.

BUGS

Whatever the default file system is, it is likely to be wrong. Specify the file system explicitly.

If the file is open, *clri* is likely to be ineffective.

NAME

cmp − compare two files

SYNOPSIS

cmp [**−l**] [**−s**] file1 file2

DESCRIPTION

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

Options:

−l Print the byte number (decimal) and the differing bytes (octal) for each difference.

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

SEE ALSO

comm(1), diff(1)

DIAGNOSTICS

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

NAME

cmpfs − compare and archive file systems

SYNOPSIS

/etc/cmpfs [**−Nmtvrseiobpqn**] [tapefile] [name] dir1 [dir2]

DESCRIPTION

Cmpfs saves file systems on mag tape (or a specified file). It writes files on the tape in an archive format; i.e., the tape consists of a sequential list of file headers and files. Whereas the *tp* program collects all the file headers into a fixed-length directory on the tape, *cmpfs* writes a header, then the file, followed by another header and the file, etc. When used in the compare mode, it saves only the differences between the two file systems which are heirarchically under the *dir1* and *dir2* arguments. In this mode, *dir1* is taken to be the "old" file system, and *dir2* is taken to be the "new"; i.e., *cmpfs* will produce all changes necessary in order to take *dir1* to *dir2* . When used in the non-compare mode, it saves the entire file system under the *dir1* argument.

Note: *cmpfs* is coded to run faster if *dir1* is a path name which begins with "/". If *dir2* is present, it must also begin with a '/' in order for the program to run faster.

The flags specify what action *cmpfs* is to take:

N *N* is and optional one or two decimal digits which designate which tape drive the program should use. Drive **0** is default if not drive is specified.

m Write the output to the file *tapefile* instead of the tape drive.

t Produce a list of the differences between the two file systems. If the second file system is null, this will effectively be a list of all the files in the given file system. The produced listing consists of one of the flags **[dcal]** followed by a path name. The flags mean:

d (Delete) The named file is in the *dir1* file system, but not in the *dir2* file system. This flag will never appear when the program is being run in the non-compare mode.

c (Change) The named file is in both file systems and is different. "Different" may be in any of the following senses; mode, owner, group or file contents. The one exception is the case of directories; if the only difference between the two directories is their size, *cmpfs* will produce no output. This flag will never appear in the non-compare mode.

a (Add) In the non-compare mode, this flag means that the file simply exists in *dir1* . In the compare mode, this flag means that the files does not exist in *dir1* but does exist in *dir2* .

l (Link) The named file is a link to a previously found file. The previous file will always be associated with a *c* or *a* flag.

The list of names output by the program is in the order in which they appear in their respective directories; *cmpfs* does no sorting.

v Produce verbose output. For each file encountered which would produce some form of output on the requesting terminal, *cmpfs* normally outputs the entry type **[dcal]** and the relative path name. The verbose option expands this output to include the mode, link count, owner id, group id, and size of the file on the tape.

r Produce a tape representing the differences found. Each entry consists of a header containing the entry type **[dcal]**, mode, owner id, group id, file size and name. The name contains neither the *dir1* or *dir2* name; hence it is relative. The remainder of the entry produced depends on the **[dcal]** flag associated with the file:

d No further information is required for a delete entry.

c,a The contents of the file follows starting at the next tape record − unless the file is a directory or a special file. For a directory, the contents of the directory is dumped if the eflag is specified, but not otherwise. The contents of the directory is used if the eflag is specified when the tape is read by updfs. For a special file the major/minor device specification is recorded in the entry header.

l The "link to" name follows immediately after the header.

s Produce entries for special files. If this flag is missing, *cmpfs* will ignore all special files it comes across. Hence if the tape is being generated for an installation which has a different device configuration, the *s* flag should be left off, or the ignore capability of the program should be used (see below).

e Produce entries according to the current epoch time as define in the date file: **/etc/epoch**. *Dir2* may not be given. when the *e* flag is specified, *cmpfs* will look through the file system specified by the *dir1* argument and output (governed by the *t*, *r* and *s* flags) those files and directories whose modification dates are later than the epoch date. The command *epoch*(1M) may be used to modify the epoch date.

i Ignore all files on the tape which have the same relative pathname as one of the pathnames in an ignore file, or which are heirarchically lower. The name of the ignore file is taken to be the next argument in the argument list, i.e. *name* .

o Look at only files which match a name in an only file or are heirarchically lower. The name of the only file is taken to be the next argument in the argument list.

b Output is blocked 5120 bytes per record instead of 512.

p ignore file *mode* when doing comparsion.

q ignore file *user* ownership when doing comparsion.

n ignore file *group* ownership when doing comparsion.

An ignore/only file should be a list of relative pathnames (both file names and directory names are allowed) separated by newlines. The "relative" requirement is important; for example, it should be clear that no pathname may start with a "/". Although it is logically possible to have a situation where it would be convenient to have both an ignore and an only file, *cmpfs* allows only one or the other to be used.

FILES

/tmp/fslp? link bookkeeping
/tmp/fsln? link bookkeeping
/dev/mt?

SEE ALSO

epoch(1M), updfs(1M)

BUGS

Cmpfs does not know about directories that begin with '.'.

NAME

cmt – insert the delta commentary for an initial SCCS delta

SYNOPSIS

cmt [**-m**[mrlist]] [**-y**[comment]] name ...

DESCRIPTION

Cmt inserts the delta commentary, for the initial delta created by *admin*(1S), of each named SCCS file.

If a directory is named, *cmt* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname 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.

Keyletter arguments apply independently to each named file.

-m[*mrlist*] If the SCCS file has the **v** flag set (see *admin*(1S)) then a Modification Request (MR) number *must* be supplied.

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

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

Note that if the **v** flag has a value (see *admin*(1S)), it is taken to be the name of a program (or shell procedure) which validates the correctness of the MR numbers. If a non-zero exit status is returned from MR number validation program, *cmt* terminates (it is assumed that the MR numbers were not all valid).

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

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

The exact permissions necessary to insert the history entry of an initial delta are documented in the *Source Code Control System User's Guide*. Simply stated, they are either (1) if you created the SCCS file, you can insert its delta commentary; or (2) if you own the file and directory you can insert a delta commentary. No other changes are made to the delta table entry.

FILES

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

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

SEE ALSO

admin(1S), get(1S), delta(1S), prs(1S), help(1S), sccsfile(5)

Source Code Control System User's Guide by L. E. Bonanni and C. A. Salemi.

DIAGNOSTICS

Use *help*(1S) for explanations.

NAME

col − filter reverse line-feeds

SYNOPSIS

col [**−bfx**]

DESCRIPTION

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

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

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

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

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

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

SEE ALSO

nroff(1), tbl(1)

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, and **−Tlp** otherwise.

BUGS

Cannot back up more than 128 lines.
Allows at most 800 characters, including backspaces, on a line.

NAME

comb – combine SCCS deltas

SYNOPSIS

comb [–o] [–s] [–psid] [–clist] name ...

DESCRIPTION

Comb generates a shell procedure (see *sh*(1)) which, when run, will reconstruct the given SCCS files. The reconstructed files will, hopefully, be smaller than the original files. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, *comb* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with s.) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The generated shell procedure is written on the standard output.

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

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

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

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

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

100 * (original – combined) / original

(Sizes are in blocks.) We recommend that before any SCCS files are actually combined, one should use this option to determine exactly how much space is saved by the combining process.

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

FILES

s.COMB The name of the reconstructed SCCS file.

comb????? Temporary.

SEE ALSO

get(1S), delta(1S), admin(1S), prs(1S), help(1S), sccsfile(5)

Source Code Control System User's Guide by L. E. Bonanni and C. A. Salemi.

DIAGNOSTICS

Use *help*(1S) for explanations.

BUGS

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

NAME

comm − select or reject lines common to two sorted files

SYNOPSIS

comm [− [**123**]] file1 file2

DESCRIPTION

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

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

SEE ALSO

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

NAME

cp, ln, mv − copy, link, or move files

SYNOPSIS

cp [−**d**] file1 [file2 ...] target
ln [−**d**] file1 [file2 ...] target
mv [−**d**] file1 [file2 ...] target

DESCRIPTION

File1 is copied (linked, moved) to *target*. Under no circumstance can *file1* and *target* be the same. If *target* is a directory, then one or more files are copied (linked, moved) to that directory.

If *mv* determines that the mode of *target* forbids writing, it will print the mode (see *chmod*(2)) and read the standard input for one line (if the standard input is a terminal). If the line begins with **y**, the move takes place otherwise *mv* exits.

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

The −**d** switch will cause the date of the original file to be retained (or inherited) by the resulting file. This is particularly useful when moving files to new devices, where the original date conveys useful information about the contents of the file. For the *mv* and *ln* command, the original date can only be retained if the user is root or the owner of the file.

SEE ALSO

rm(1), chmod(2), utime(2)

BUGS

If *file1* and *target* lie on different file systems, *mv* must copy the file and delete the original. In this case the owner name becomes that of the copying process and any linking relationship with other files is lost.

Ln will not link across file systems at all.

NAME

cpio - copy file archives in and out

SYNOPSIS

cpio -o [**acBv**]

cpio -i [**Bcdmrtuvfs6**] [patterns]

cpio -p [**adlmruv**] directory

DESCRIPTION

Cpio -o (copy out) reads the standard input to obtain a list of path names and copies those files onto the standard output together with path name and status information.

Cpio -i (copy in) extracts files from the standard input which is assumed to be the product of a previous **cpio -o**. Only files with names that match *patterns* are selected. *Patterns* are given in the name-generating notation of *sh*(1). In *patterns*, meta-characters **?**, *****, and **[...]** match the slash **/** character. Multiple *patterns* may be specified and if no *patterns* are specified, the default for *patterns* is ***** (i.e., select all files). The extracted files are conditionally created and copied into the current directory tree based upon the options described below.

Cpio -p (pass) reads the standard input to obtain a list of path names of files that are conditionally created and copied into the destination *directory* tree based upon the options described below.

The meanings of the available options are:

a Reset access times of input files after they have been copied.

B Input/output is to be blocked 5,120 bytes to the record (does not apply to the *pass* option; meaningful only with data directed to or from **/dev/rmt?**).

d *Directories* are to be created as needed.

c Write *header* information in ASCII character form for portability.

r Interactively *rename* files. If the user types a null line, the file is skipped.

t Print a *table of contents* of the input. No files are created.

u Copy *unconditionally* (normally, an older file will not replace a newer file with the same name).

v *Verbose*: causes a list of file names to be printed. When used with the **t** option, the table of contents looks like the output of an **ls -l** command (see *ls*(1)).

l Whenever possible, link files rather than copying them. Usable only with the **-p** option.

m Retain previous file modification time. This option is ineffective on directories that are being copied.

f Copy in all files except those in patterns.

s Swap bytes. Use only with the **-i** option.

S Swap halfwords. Use only with the **-i** option.

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

6 Process an old (i.e., UNIX Sixth Edition format) file. Only useful with **-i** (copy in).

EXAMPLES

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

```
ls | cpio -c >/dev/mt0

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

The trivial case ``find . -depth -print | cpio -oB >/dev/rmt0'' can be handled more efficiently by:

```
find . -cpio /dev/rmt0
```

SEE ALSO

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

BUGS

Path names are restricted to 128 characters. If there are too many unique linked files, the program runs out of memory to keep track of them and, thereafter, linking information is lost. Only the super-user can copy special files. The **-B** option does not work with certain magnetic tape drives (see un32(7) in the UNIX System Administrator's Manual).