HP-UX Reference

Vol. 2: Sections 1M, 2, 3, 4, 5, and 7

HP 9000 Series 500 Computers
HP-UX Release 5.3

HP Part Number 09000-90010

HEWLETT
PACKARD

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

Printing History

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

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

April 1987...Edition 1

February 1988...Update 1. Replaced incorrect manual entry for *fsck*[SDF](1M) in Volume 2.

April 1989...Update 2. Replaced the following incorrect manual entries: *awk*(1) in Volume 1; *csh*(1) in Volume 1; *grep*(1) in Volume 1; *lifcp*(1) in Volume 1; *sort*(1) in Volume 1; *spell*(1) in Volume 1; *cron*(1M) in Volume 2; *fsck*[SDF](1M) in Volume 2.

NAME

awk – text pattern scanning and processing language

SYNOPSIS

awk [**–F***c*] [*prog* | **–f** *awkfile*] [*parameters*] [*files* ...]

DESCRIPTION

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

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

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

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

A pattern-action statement has the form:

```
pattern { action }
```

A missing action means print the line; a missing pattern always matches. An action is a sequence of statements. A statement can be one of the following:

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

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

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

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

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

A relational expression is one of the following:

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

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

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

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

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

or by using the **-F***c* option.

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

EXAMPLES

Print lines longer than 72 characters:

```
length > 72
```

Print first two fields in opposite order:

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

Add up first column, print sum and average:

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

Print fields in reverse order:

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

Print all lines between start/stop pairs:

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

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

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

Print file, filling in page numbers starting at 5:

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

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

mail This variable contains a list of the files where *csh* checks for your mail. *Csh* periodically (default is 10 minutes) checks this variable after a command completion which results in a prompt. If the variable contains a filename that has been modified since the last check (resulting from mail being put in the file), *csh* prints **You have new mail.**

If the first word of the value of *mail* is numeric, that number specifies a different mail checking interval in seconds.

If multiple mail files are specified, then the shell says *New*mail*in*file_name, where *file_name* is the file containing the mail.

noclobber This variable places restrictions on output redirection to insure that files are not accidentally destroyed, and that commands using append redirection (>>) refer to existing files.

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

nonomatch If set, it is no longer an error for a filename expansion to not match any existing files. If there is no match, the primitive pattern is returned. It is still an error for the primitive pattern to be malformed, i.e. ′**echo** [′ still gives an error.

notify If set, *csh* notifies you immediately (through your standard output device) of background job completions. The default is **unset** (indicate job completions just before printing a prompt).

path Each word of the path variable specifies a directory in which commands are to be sought for execution. A null word specifies your current working directory. If there is no *path* variable then only full path names can be executed. When *path* is not set and when users do not specify full pathnames, *csh* searches for the command through the directories **.** (your current directory), */bin*, */lbin*, */usr/bin*, and */usr/lbin*. A *csh* which is given neither the **−c** nor the **−t** option normally hashes the contents of the directories in the *path* variable after reading *.cshrc*, and each time the *path* variable is reset. If new commands are added to these directories while the shell is active, it is necessary to execute *rehash* for *csh* to access these new commands.

prompt This variable lets you select your own prompt character string. The prompt is printed before each command is read from an interactive terminal input. If a **!** appears in the string it is replaced by the current command history buffer event number unless a preceding \ is given. The default prompt is the percent sign (%) for users and the pound sign (#) for the super-user.

shell This variable contains the name of the file in which the *csh* program resides. This variable is used in forking shells to interpret files which have their execute bits set, but which are not executable by the system. (See the description of **Non-built-In Command Execution**).

status This variable contains the status value returned by the last command. If the command terminated abnormally, 0200 is added to the status variable's value. Built-in commands which terminated abnormally return exit status **1**, and all other built-in commands set status to **0**.

time This variable contains a numeric value which controls the automatic timing of commands. If set, then *csh* prints, for any command which takes more than the specified number of cpu seconds, a line of information to your standard output device giving user, system, and real execution times plus a utilization percentage. The utilization percentage is the ratio of user plus system times to real time. This message is printed after the command finishes execution.

verbose This variable is set by the **−v** command line option. If set, the words of each command are printed on the standard output device after history substitutions have been made.

Command and Filename Substitution

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

Command Substitution

Command substitution is indicated by a command enclosed in grave accents (\` ...\`). The output from such a command is normally broken into separate words at blanks, tabs and newlines, with null words being discarded, this text then replacing the original string. Within double quotes, only newlines force new words; blanks and tabs are preserved.

In any case, the single final newline does not force a new word. Note that it is thus possible for a command substitution to yield only part of a word, even if the command outputs a complete line.

Filename Substitution

If a word contains any of the characters *, ?, [, or {, or begins with the character ~, then that word is a candidate for filename substitution, also known as *globbing*. This word is then regarded as a pattern, and replaced with an alphabetically sorted list of file names which match the pattern. In a list of words specifying filename substitution it is an error for no pattern to match an existing file name, but it is not required for each pattern to match. Only the metacharacters *, ?, and [imply pattern matching, while the characters ~ and { are more like abbreviations.

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

The tilde character (~) at the beginning of a filename is used to refer to home directories. By itself, the tilde expands to your home directory as reflected in the value of the variable *home*. When followed by a name consisting of letters, digits and - characters, the shell searches for a user with that name and substitutes their home directory; thus **~ken** might expand to **/users/ken** and **~ken/chmach** to **/usr/ken/chmach**. If the ~ is followed by a character other than a letter or /, or appears somewhere other than at the beginning of a word, it is left undisturbed.

The metanotation **a{b,c,d}e** is a shorthand for "abe ace ade". Left to right order is preserved, with results of matches being sorted separately at a low level to preserve this order. This construct may be nested. Thus

NAME

grep, egrep, fgrep – search a file for a pattern

SYNOPSIS

Levels B and C

grep [options] expression [files]

Level C Only

egrep [options] [expression] [files]

fgrep [options] [strings] [files]

DESCRIPTION

Commands of the *grep* family search the input *files* (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output. *Grep* patterns are limited regular *expressions* in the style of *ed*(1); it uses a compact non-deterministic algorithm. *Egrep* patterns are full regular *expressions*; it uses a fast deterministic algorithm that sometimes needs exponential space. *Fgrep* patterns are fixed *strings*; it is fast and compact. The following *options* are recognized:

–v All lines but those matching are printed.
–x (Exact) only lines matched in their entirety are printed (*fgrep* only).
–c Only a count of matching lines is printed.
–i Ignore upper/lower case distinction during comparisons.
–l Only the names of files with matching lines are listed (once), separated by new-lines.
–n Each line is preceded by its relative line number in the file.
–b Each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.
–s The error messages produced for nonexistent or unreadable files are suppressed (*grep* only).
–e *expression*
Same as a simple *expression* argument, but useful when the *expression* begins with a – (does not work with *grep*).
–f *file* The regular *expression* (*egrep*) or *strings* list (*fgrep*) is taken from the *file*.

In all cases, the file name is output if there is more than one input file. Care should be taken when using the characters **$**, *, [, ^, |, (,), and \ in *expression*, because they are also meaningful to the shell. It is safest to enclose the entire *expression* argument in single quotes ´...´.

Fgrep searches for lines that contain one of the *strings*, each of which is separated from the next by a new-line.

Egrep accepts regular expressions as in *ed*(1), except for \(, \), \{ and \}, with the addition of:

1. A regular expression followed by + matches one or more occurrences of the regular expression.
2. A regular expression followed by ? matches 0 or 1 occurrences of the regular expression.
3. Two regular expressions separated by | or by a new-line match strings that are matched by either.
4. A regular expression may be enclosed in parentheses ( ) for grouping.

The order of precedence of operators is [ ], then * ? +, then concatenation, then | and new-line.

EXAMPLES

The following example searches two files, finding all lines containing occurrences of any of four strings:

```
fgrep ´if
then
else
fi´ script1 script2
```

Note that the single quotes are necessary to tell *fgrep* when the strings have ended and the file names have begun.

This example searches for a new-line in a file:

```
grep -v '\.' file1
```

The **−v** option causes *grep* to print those lines that do not match the expression. Since a new-line cannot be matched with dot, only lines containing a new-line are printed.

SEE ALSO

ed(1), sed(1), sh(1).

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files (even if matches were found).

BUGS

Ideally there should be only one *grep*, but we do not know a single algorithm that spans a wide enough range of space-time tradeoffs.

Lines are limited to BUFSIZ characters; longer lines are truncated. (BUFSIZ is defined in **/usr/include/stdio.h**.)

Egrep does not recognize ranges, such as **[a−z]**, in character classes.

Grep finds lines in the input file by searching for a new-line. Thus, if there is no new-line at the end of the file, *grep* will ignore the last line of the file.

If there is a line with embedded nulls, *grep* will only match up to the first null; if it matches, it will print the entire line.

INTERNATIONAL SUPPORT

grep: 8- and 16-bit data, 8-bit filenames, messages
egrep: 8-bit data and filenames
fgrep: 8-bit data and filenames, messages.

NAME

lifcp – copy to or from LIF files

SYNOPSIS

lifcp [–Txxx] [–Lxxx] [–vxxx] [–a] [–b] [–ixxx] [–r] [–t] file1 file2
lifcp [–Txxx] [–Lxxx] [–vxxx] [–a] [–b] [–ixxx] [–r] [–t] file1 [file2 ...] directory

DESCRIPTION

Lifcp copies a LIF file to an HP-UX file, an HP-UX file to a LIF file, or a LIF file to another LIF file. It also copies a list of (HP-UX/LIF) files to a (LIF/HP-UX) directory. The last name on the argument list is the destination file or directory.

Options may appear singly or be combined in any order before the file names. The space between option and argument is optional.

-Txxx Used only when copying files to a LIF volume. This option will force the file type of the LIF directory entry to be set to the argument given, which may be decimal, octal or hex in standard "C" notation.

-Lxxx Used only when copying files to a LIF volume. This option will set the "last volume flag" to xxx (0 or 1). The default "last volume flag" is one.

-vxxx Used only when copying files to a LIF volume. This option will set the "volume number" to **xxx.** The default "volume number" is one.

-a causes an ASCII copy. In the case of copying from HP-UX to LIF, this creates a LIF ASCII file.

-b This option will force a BINARY mode of copying regardless of the file type. When copying in BINARY mode from HP-UX to LIF the default file type is BINARY(-2). (For details on available modes of copying refer to *lif*(4)). This option is a no-op when copying from LIF to LIF.

-ixxx Used only when copying files to a LIF volume. This option sets the "implementation" field of the LIF directory entry to the argument given, which may be decimal, octal or hex in standard "C" notation. The "implementation" field can only be set for file types -2001 to -100000 (octal). The "implementation" field is set to zero for all interchange file types and for file types -2 to -200 (octal).

-r This option will force a RAW mode of copying regardless of the file type. When copying in RAW mode from HP-UX to LIF the default file type is BIN(-23951). **-T** option will override the default file type. (various modes of copying are explained in *lif*(4).) This option is a no-op in case of LIF to LIF copying.

-t will cause the HP-UX file names to be translated to a name acceptable by a LIF utility. That is, all the lower-case letters will be up-shifted and all other characters except numeric will be changed to an underscore (_). If the HP-UX file name starts with a non-letter, the file name will be preceded by the capital letter (X). Note that if there are two files named colon (:) and semicolon (;), both of them will be translated to X_. File names will be truncated to a maximum of 10 characters. When copying a LIF file to (HP-UX/LIF) file **-t** is a no-op. Omitting **-t** will cause error to be generated if an improper name is used.

The default copying modes when copying from LIF to HP-UX are summarized in the following table:

file type	default copying mode
ASCII	ASCII
BINARY	BINARY
BIN	RAW
other	RAW

When copying from HP-UX to LIF, the default copying mode is ASCII and an ASCII file is created.

When copying from LIF to LIF, if no options are specified then all the LIF directory fields and content of the file are duplicated from source to destination.

A LIF file name is recognized by the embedded colon (:) delimiter (see *lif*(4) for LIF file naming conventions). A LIF directory is recognized by a trailing colon. If an HP-UX file name containing a colon is used, the colon must be escaped with two backslash characters (\\) (the shell removes one of them).

The file name '–' (dash) will be interpreted to mean standard input or standard output, depending on its position in the argument list. This is particularly useful if the data requires non-standard translation. When copying from standard input, if no other name can be found, the name "STDIN" is used.

The LIF file naming conventions are known only by the LIF utilities. Since file name expansion is done by the shell, this mechanism cannot be used for expansion of LIF file names.

Note that the media should **not** be mounted while using *lifcp*.

HARDWARE DEPENDENCIES

Series 500:

You **must** use a character special file to access the media.

Series 800:

The following option is also supported:

–Knnn forces each file copied in to begin on a nnn * 1024 byte boundary from the beginning of the volume. This is useful when files are used for Series 800 boot media. This option has no effect when copying from a LIF volume.

EXAMPLES

lifcp abc lifvol:CDE

copy HP-UX file abc to LIF file CDE on LIF volume lifvol which is actually an HP-UX file initialized to be a LIF volume.

lifcp -t * ../lifvol:

will copy all the HP-UX files in the current directory to the LIF volume lifvol which is present in the parent directory. File names are translated to appropriate LIF file names.

lifcp -r -T -5555 -t *.o lifvol:

will copy all the HP-UX object files in the current directory to the LIF volume lifvol. Copying mode is RAW and LIF file types are set to -5555.

lifcp – /dev/dsk/1s2:A_FILE
copy standard input to LIF file A_FILE on LIF volume /dev/dsk/1s2.

lifcp lifvol:ABC /dev/dsk/1s2:CDE
copy LIF file ABC in lifvol to LIF file CDE on /dev/dsk/1s2.

pr abc | lifcp – lifvol:ABC
copy the output of pr to the LIF file ABC.

pr abc | lifcp – lifvol:
copy the output of pr to the LIF volume lifvol. LIF file STDIN is crated since no files names are specified.

lifcp lifvol:ABC –
copy LIF file ABC in lifvol to standard out.

lifcp * ../lifvol:
copy all files within current directory to LIF files of the same name on LIF volume lifvol (may cause errors if file names in the current directory do not obey LIF naming conventions!).

AUTHOR
Lifcp was developed by the Hewlett-Packard Company.

SEE ALSO
lifinit(1), lifls(1), lifrename(1), lifrm(1), lif(4).

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

NAME

lifinit - write LIF volume header on file

SYNOPSIS

lifinit [−vnnn] [−**d**nnn] [−**n** string] file

DESCRIPTION

Lifinit writes a LIF volume header on a volume or file. *Options* may appear in any order. Their meanings are:

−v*nnn* Sets the volume size to *nnn* bytes. If *nnn* is not a multiple of 256, it will be rounded down to the next such multiple.

−**d***nnn* Sets the directory size to *nnn* file entries. If *nnn* is not a multiple of 8, it will be rounded up to next such multiple.

−**n** *string* sets the volume name to be *string.* If the −**n** option is not specified, the volume name is set to the last component of the path name specified by *file.* A legal LIF volume name is 6 characters long and is limited to upper case letters (A-Z), digits (0-9) and the underscore character (_). The first character (if any) must be a letter. The utility will automatically perform translation to create legal LIF volume names. Therefore, all lower-case letters are up-shifted and all other characters except numeric and underscore will be replaced with capital letter (X). If the volume name does not start with a letter, the volume name will be preceded by the capital letter (X). The volume name will also be right padded with blanks or truncated as needed to be 6 characters long. If **-n** is used with no *string*, the default volume name is set to 6 blanks.

If *file* does not exist, a regular HP-UX disk file is created and initialized.

The default values for volume size are 256K bytes for regular files, and the actual capacity of the device for device files.

The default directory size is a function of the volume size. A percentage of the volume size is allocated to the volume directory as follows:

VOLUME SIZE	DIRECTORY SIZE
< 2MB	˜1.3%
> 2MB	˜0.5%

Each directory entry occupies 32 bytes of storage. The actual directory space is subject to the rounding rules stated above.

Note that you should **not** mount the special file before using *lifinit.*

HARDWARE DEPENDENCIES

Series 200, Series 300

If your media has never been initialized, it must be initialized using *mediainit*(1) before *lifinit* can be used. (Refer to the System Administrator Manual for details concerning *mediainit.*)

Series 500

You **must** use a character special file to access the media.

If your media has never been initialized, it must be initialized using *sdfinit*(1M) before *lifinit* can be used.

Series 800:

SEE ALSO

ioctl(2), lp(7).

NAME

sort – sort and/or merge files

SYNOPSIS

sort [**–cmu**] [**–o***output*] [**–y***kmem*] [**–z***recsz*] [**–T***dir*] [**–t***x*]
[**–bdfilnrM**] [+*pos1* [–*pos2*]] [*file* ...]

DESCRIPTION

Sort sorts lines of all the named files together and writes the result on the standard output. The standard input is read if – is used as a file name or no input files are named.

Comparisons are based on one or more sort keys extracted from each line of input. By default, there is one sort key, the entire input line, and ordering is lexicographic by bytes in machine collating sequence.

The following options alter the default behavior:

–c	Check that the input file is sorted according to the ordering rules; give no output unless the file is out of sort.
–m	Merge only, the input files are already sorted.
–u	Unique: suppress all but one in each set of lines having equal keys.
–o*output*	The argument given is the name of an output file to use instead of the standard output. This file may be the same as one of the inputs. There may be optional blanks between **–o** and *output*.
–y*kmem*	The amount of main memory used by the sort has a large impact on its performance. Sorting a small file in a large amount of memory is a waste. If this option is omitted, *sort* begins using a system default memory size, and continues to use more space as needed. If this option is presented with a value, *kmem*, *sort* will start using that number of kilobytes of memory, unless the administrative minimum or maximum is violated, in which case the corresponding extremum will be used. Thus, **–y**0 is guaranteed to start with minimum memory. By convention, **–y** (with no argument) starts with maximum memory.
–z*recsz*	The size of the longest line read is recorded in the sort phase so buffers can be allocated during the merge phase. If the sort phase is omitted via the **–c** or **–m** options, a popular system default size will be used. Lines longer than the buffer size will cause *sort* to terminate abnormally. Supplying the actual number of bytes in the longest line to be merged (or some larger value) will prevent abnormal termination.
–T*dir*	Use *dir* as the directory for temporary sort records rather than the default directory, which is **/usr/tmp**.

The following options override the default ordering rules:

–d	Quasi-dictionary order: only letters, digits and blanks (spaces and tabs) are significant in comparisons. The **–d** option recognizes ASCII characters only (see the DIAGNOSTICS section).
–f	Fold lowercase letters into uppercase. The **-f** option is ignored if a language other than **n-computer** is specified.
–i	Ignore characters outside the ASCII range 040-0176 in non-numeric comparisons. The **-i** option will be ignored if a language other than **n-computer** is specified.

−M	Compare as months; however, the **−M** option compares American month names only (see the DIAGNOSTICS section). The first three non-blank characters of the field are folded to uppercase and compared so that "JAN" < "FEB" < ... < "DEC". An invalid field is considered less than "JAN". The **−M** option implies the **−b** option (see below).
−n	An initial numeric string, consisting of optional blanks, optional minus sign, and zero or more digits with optional decimal point, is sorted by arithmetic value. The **−n** option recognizes only the English radix character (decimal point) in numeric comparisons (see the DIAGNOSTICS section). The **−n** option implies the **−b** option (see below). Note that the **−b** option is only effective when restricted sort key specifications are in effect.
−r	Reverse the sense of comparisons.

The following option applies to International Support:

−l	Collate characters using the collation rules associated with the user's LANG variable (see *environ*(5)). If the language is not specified or is set to **n-computer**, the ordering is lexicographic by bytes in machine-collating sequence. If the user's language includes two-byte characters, one-byte characters are machine-collated before two-byte characters.

The notation +*pos1* −*pos2* restricts a sort key to one beginning at *pos1* and ending at *pos2*. The characters at positions *pos1* and *pos2* are included in the sort key (provided that *pos2* does not precede *pos1*). A missing −*pos2* means the end of the line.

When ordering options appear before restricted sort key specifications, the requested ordering rules are applied globally to all sort keys. When attached to a specific sort key (described below), the specified ordering options override all global ordering options for that key.

Specifying *pos1* and *pos2* involves the notion of a field, a minimal sequence of characters followed by a field separator or a new-line. By default, the first blank (space or tab) of a sequence of blanks acts as the field separator. All blanks in a sequence of blanks are considered to be part of the next field; for example, all blanks at the beginning of a line are considered to be part of the first field. The treatment of field separators can be altered using the options:

−t*x*	Use *x* as the field separator character; *x* is not considered to be part of a field (although it may be included in a sort key). Each occurrence of *x* is significant (for example, *xx* delimits an empty field).
−b	Ignore leading blanks when determining the starting and ending positions of a restricted sort key. If the **−b** option is specified before the first +*pos1* argument, it will be applied to all +*pos1* arguments. Otherwise, the **b** flag may be attached independently to each +*pos1* or −*pos2* argument (see below).

Pos1 and *pos2* each have the form *m.n* optionally followed by one or more of the flags **bdfinrM**. A starting position specified by +*m.n* is interpreted to mean the *n*+1st character in the *m*+1st field. A missing .*n* means .0, indicating the first character of the *m*+1st field. If the **b** flag is in effect, *n* is counted from the first non-blank in the *m*+1st field; +*m*.0**b** refers to the first non-blank character in the *m*+1st field.

A last position specified by −*m.n* is interpreted to mean the *n*th character (including separators) after the last character of the *m th* field. A missing .*n* means .0, indicating the last character of the *m*th field. If the **b** flag is in effect, *n* is counted from the last leading blank in the *m*+1st field; −*m*.1**b** refers to the first non-blank in the *m*+1st field.

When there are multiple sort keys, later keys are compared only after all earlier keys compare equal. Lines that otherwise compare equal are ordered with all bytes significant.

DIAGNOSTICS

Sort comments and exits with non-zero status for various trouble conditions (for example, when input lines are too long), and for disorder discovered under the **–c** option.

When the last line of an input file is missing a new-line character, *sort* appends one, prints a warning message, and continues.

If an error occurs when accessing the tables that contain the collation rules for the specified language, *sort* prints a warning message and defaults to **n-computer.**

If a **–d**, **–n** or **–M** option is used with a language other than **n-computer**, *sort* prints a warning message and defaults to **n-computer**.

EXAMPLES

Sort the contents of *infile* with the second field as the sort key:

```
sort +1 –2 infile
```

Sort, in reverse order, the contents of *infile1* and *infile2*, placing the output in *outfile* and using the first character of the second field as the sort key:

```
sort –r –o outfile +1.0 –1.2 infile1 infile2
```

Sort, in reverse order, the contents of *infile1* and *infile2*, using the first non-blank character of the second field as the sort key:

```
sort –r +1.0b –1.1b infile1 infile2
```

Print the password file (*passwd*(4)) sorted by the numeric user ID (the third colon-separated field):

```
sort –t: +2n –3 /etc/passwd
```

Print the lines of the already sorted file *infile*, suppressing all but the first occurrence of lines having the same third field (the options **–um** with just one input file make the choice of a unique representative from a set of equal lines predictable):

```
sort –um +2 –3 infile
```

WARNINGS

When using the specified ordering option(s) with two-byte characters, *pos1* and *pos2* must specify byte position, not character position.

The **–t** option only recognizes a character encoded in one byte as a field separator character.

FILES

/usr/tmp/stm???

SEE ALSO

comm(1), join(1), uniq(1), col_seq_8(4), environ(5), hpnls(5), langid(5).

INTERNATIONAL SUPPORT

8- and 16-bit data, 8-bit file names, messages.

NAME

spell, hashmake, spellin, hashcheck – find spelling errors

SYNOPSIS

spell [**–v**] [**–b**] [**–x**] [**–l**] [**–i**] [+local_file] [files]

/usr/lib/spell/hashmake

/usr/lib/spell/spellin n

/usr/lib/spell/hashcheck spelling_list

DESCRIPTION

Spell collects words from the named *files* and looks them up in a spelling list. Words that neither occur among nor are derivable (by applying certain inflections, prefixes, and/or suffixes) from words in the spelling list are printed on the standard output. If no *files* are named, words are collected from the standard input.

Spell ignores most *troff*, *tbl*(1), and *eqn* constructions.

Options

–v	All words not literally in the spelling list are printed, and plausible derivations from the words in the spelling list are indicated.
–b	British spelling is checked. Besides preferring *centre*, *colour*, *programme*, *speciality*, *travelled*, etc., this option insists upon *-ise* in words like *standardise*.
–x	Every plausible stem is printed with = for each word.

By default, *spell* (like *deroff*(1)) follows chains of included files (**.so** and **.nx** *troff* requests), *unless* the names of such included files begin with **/usr/lib**. Under the –l option, *spell* will follow the chains of *all* included files. Under the –i option, *spell* will ignore all chains of included files.

Under the *+local_file* option, words found in *local_file* are removed from *spell*'s output. *Local_file* is the name of a user-provided file that contains a sorted list of words, one per line. With this option, the user can specify a set of words that are correct spellings (in addition to *spell*'s own spelling list) for each job.

The spelling list is based on many sources, and while more haphazard than an ordinary dictionary, is also more effective with respect to proper names and popular technical words. Coverage of the specialized vocabularies of biology, medicine, and chemistry is light.

Pertinent auxiliary files may be specified by name arguments, indicated below with their default settings (see **FILES**). Copies of all output are accumulated in the history file. The stop list filters out misspellings (e.g., thier=thy–y+ier) that would otherwise pass.

Three routines help maintain and check the hash lists used by *spell*:

hashmake	Reads a list of words from the standard input and writes the corresponding nine-digit hash code on the standard output.
spellin n	Reads *n* hash codes from the standard input and writes a compressed spelling list on the standard output. Information about the hash coding is printed on standard error.
hashcheck	Reads a compressed *spelling_list* and recreates the nine-digit hash codes for all the words in it; it writes these codes on the standard output.

EXAMPLES

To create a personal spelling list, incorporating the words already present in the default American spelling list, **/usr/lib/spell/hlista**:

```
cat /usr/lib/spell/hlista | /usr/lib/spell/hashcheck >tmp1
/usr/lib/spell/hashmake <addwds >>tmp1
sort -u -o tmp1 tmp1
/usr/lib/spell/spellin `wc -l <tmp1` <tmp1 >hlista
```

To modify the default British spelling list, **/usr/lib/spell/hlistb**, replace all occurrences of **hlista** with **hlistb** in the above example.

To add words to the default spelling list, change login to *root*, change the current working directory to **/usr/lib/spell** and enter the commands listed in the above example.

WARNINGS

The spelling list's coverage is uneven. New installations will probably wish to monitor the history file for several months to gather local additions. Typically, these are kept in a separate local file that is added to the hashed *spelling_list* via *spellin*, as shown above.

The British spelling feature was done by an American.

FILES

D_SPELL=/usr/lib/spell/hlist[ab]	hashed spelling lists, American & British
S_SPELL=/usr/lib/spell/hstop	hashed stop list
H_SPELL=/usr/lib/spell/spellhist	history file
/usr/lib/spell/spellprog	program

VARIABLES

D_SPELL	Your hashed spelling list. (Default as above.)
H_SPELL	Spelling history. (Default as above.)
S_SPELL	Your hashed stop list. (Default as above.)

SEE ALSO

deroff(1), sed(1), sort(1), tbl(1), tee(1).

NAME

cron – clock daemon

SYNOPSIS

/etc/cron

DESCRIPTION

Cron executes commands at specified dates and times. Regularly scheduled commands can be specified according to instructions found in crontab files; users can submit their own crontab file via the *crontab* command. Commands which are to be executed only once may be submitted via the *at* command. Since *cron* never exits, it should only be executed once. This is best done by running *cron* from the initialization process through the file **/etc/rc** (see *init*(1M)).

On the days of daylight savings time transition (in time zones where daylight savings time applies), *cron* will schedule commands differently than normal.

In the following description, an ambiguous time refers to an hour and minute that occurs twice in the same day because of a daylight savings time transition (usually on a day during the Autumn season). A non-existent time refers to an hour and minute that does not occur because of a daylight savings time transition (usually on a day during the Spring season). DST-shift refers to the offset that is applied to standard time to result in daylight savings time. This is normally one hour, but may be any combination of hours and minutes up to 23 hours and 59 minutes (See *tztab*(4)).

When a command is specified to run at a time that is ambiguous, the command will be executed only at the *first* time that such a time occurs.

When a command is specified to run a time that is non-existent, the command will be executed after the specified time by an amount of time equal to the DST-shift. When such an adjustment would conflict with another time specified to run the command, the command is run only once rather than running the command twice at the same time.

For commands that are scheduled to run during all hours by specifying a '*' in the hour field of the crontab entry, the command will be scheduled without any adjustment.

Cron only examines crontab files and at(1) command files during process initialization and when a file changes. This reduces the overhead of checking for new or changed files at regularly scheduled intervals.

EXAMPLES

The following examples assume that the time zone is **MST7MDT**. In this time zone the DST transition occurs one second before 2:00 a.m. and the DST-shift is 1 hour.

Consider the following crontab entries:

# Minute	Hour	Month Day	Month	Weekday	Command
0	01	*	*	*	Job_1
0	02	*	*	*	Job_2
0	03	*	*	*	Job_3
0	04	*	*	*	Job_4
0	*	*	*	*	Job_hourly
0	2,3,4	*	*	*	Multiple_1
0	2,4	*	*	*	Multiple_2

For the period of 01:00 a.m. to 04:00 a.m. on the days of DST transition, the results will be:

Job	Times Run in Fall	Times Run in Spring
Job_1	01:00 MDT	01:00 MST
Job_2	02:00 MDT	03:00 MDT
Job_3	03:00 MST	03:00 MDT
Job_4	04:00 MST	04:00 MDT
Job_hourly	01:00 MDT	01:00 MST
	02:00 MDT	
	02:00 MST	
	03:00 MST	03:00 MDT
	04:00 MST	04:00 MDT
Multiple_1	02:00 MDT	
	03:00 MST	03:00 MDT
	04:00 MST	04:00 MDT
Multiple_2	02:00 MDT	03:00 MDT
	04:00 MST	04:00 MDT

BUGS

In the Spring, when there is a non-existent hour because of daylight savings time, a command that is scheduled to run multiple times during the non-existent hour will only run once. For example, a command scheduled to run at 2:00 and 2:30 a.m. in the **MST7MDT** time zone will only run at 3:00 a.m. The command that was scheduled at 2:30 a.m. will not be run at all, instead of running at 3:30 a.m.

The log file does not include the time zone in the time and date stamps, so it can be confusing when reading the log for a day when an hour occurs twice because of a daylight savings time transition.

FILES

/usr/lib/cron	main cron directory
/usr/spool/cron	spool area
/usr/lib/cron/log	accounting information

SEE ALSO

at(1), crontab(1), init(1M), sh(1).

DIAGNOSTICS

A history of all actions taken by cron are recorded in **/usr/lib/cron/log.**

THIS PAGE INTENTIONALLY LEFT BLANK.

NAME

decode – read and decode diagnostic events from the error log

SYNOPSIS

decode [-**L** logfile] [-**d** device file] [-**e** physical path]
[-**m** major number] [-**n** driver name] [-**p** port] [-**t** [number]]
[-**w**]

DESCRIPTION

Decode reads from stdin a raw diagnostic event message. The message is then decoded into a human-readable format and written to stdout. *Decode* continues reading event messages until EOF is detected.

Decode accepts the following list of parameters:

-**L** *logfile*	set the error log path name. (Default is stdin.)
-**d** *device file*	set the path for a special device file.
-**e** *physical path*	set the hardware path. (Uses the **1.2.3** notation.)
-**m** *major number*	set the major number.
-**n** *driver name*	set the driver name.
-**p** *port*	set the port number.
-**t** *number*	set the tail parameter. See below for a description of *number.* (Default is last 10 events.)
-**w**	causes *decode* to wait after printing each event.

When specifying a *logfile* to read with the -**L**, the full file path name must be used. Events are then read from this file until EOF is detected.

The options -**d**, -**e**, -**m**, -**n**, and -**p** are used to alter the information that *decode* prints. The options may be used singularly or in combination.

The -**t** option has a numeric parameter, *number* . When *number* is positive it shows the last *number* events in the log file. When *number* is negative the first *number* events in the log file are skipped over.

After *decode* decodes an event it is written to stdout in the following format:

```
*****************************************************
Diagnostic Event number X,   Date = DATE & TIME
*****************************************************

Port number of originating event = aa

Manager of originating event = bb

Physical path = a.b.c

Diagnostic event status:
        error        - cc
        proc num       - dd
        subsystem      - ee
```

NAME

fsck – file system consistency check and interactive repair

SYNOPSIS

/etc/fsck −p [*file system* ...]
/etc/fsck −P [*file system* ...]
/etc/fsck [**−y**] [**−n**] [**−s**] [**−d**] [*file system* ...]

DESCRIPTION

Fsck checks for and interactively repairs inconsistent conditions in the SDF file systems. If the file system is consistent, *fsck* reports the number of files, the number of blocks used, the number of blocks free, and the percent of volume unused. If the file system is inconsistent, *fsck* provides a mechanism to fix these inconsistencies depending on which form of the *fsck* command is used.

Fsck makes multiple passes over the file system, so care should be taken to ensure that the system is quiescent. You should unmount the file system being checked, if possible. At the least, the system should be single-user, and spurious processes (such as *cron*) should be killed.

The following options are interpreted by *fsck*:

−p Preen the file system. This option implies the **−y** and **−s** options. If no *file system* argument is given, eligible file systems in the **/etc/checklist** file (see discussion below on *file system)* which have the same *pass number* value are checked in parallel, thus permitting a faster boot-up on large systems with multiple mounted disks.

−P The **−P** option operates in the same manner as the **−p** option except that those file systems which were cleanly unmounted (marked FS_CLEAN) will not be checked (see *fsclean*(1M)). This can greatly decrease the amount of time required to reboot a system that was brought down cleanly. The **−P** option is intended to be invoked during the boot process by the script **/etc/bcheckrc**.

−y Assume a **yes** response to all questions asked.

−n Assume a **no** response to all questions asked; do not open the file system for writing.

−s Ignore the actual free list and unconditionally reconstruct a new one. This option is useful in correcting multiply claimed blocks when one of the claimants is the free list. When using this option, the number of unclaimed blocks reported by *fsck* includes all the blocks in the free map. This can produce extensive output if **−d** is also selected.

The **−s** option should only be selected after a previous *fsck* indicates a conflict between a file and the free map. After **fsck −s** has executed, the integrity of the conflicting file(s) should be checked.

If **−s** is used to correct a problem on a virtual memory device, there is a high probability that the final step in *fsck* will fail, and you will be forced to reboot. Should this occur, an appropriate error message will be printed. No damage should occur.

−d Dump additional information. The more **d**'s that are present, the more information that is dumped. You may specify up to five **d**'s. However, using more than two can result in an overwhelming amount of output.

Fsck also recognizes, and ignores, the **−S** and **−t** options found in other versions of *fsck*. An appropriate warning is printed.

File system is a device file name describing the device on which the file system to be checked resides. If no *file system(s)* are specified, *fsck* will read a list of default file systems from the file **/etc/checklist** (see *checklist*(4)). If only the first field is present in the *checklist* file, the file systems are checked in the order of appearance. If the optional fields (through the pass number field) are present, only the file systems with type of "rw" or "ro" are processed. If the **−p** or **−P** options are used, the *pass number* field is interpreted as follows:

pass number	action
-1	Process these entries sequentially after all other entries.
0	Ignore these entries.
1	The root file system. Process first.
2	Additional file systems. Process these in parallel after pass number 1 is complete.
n	Process these file systems in parallel after pass number *n*-1.

If neither **−p** nor **−P** are used, the actions are similar except that sequential rather than parallel checking is performed on entries with the same ***pass number*** value.

Error messages from *fsck* are written to *stderr*. Information generated because of the **−d** option and normal output is written to *stdout*. Both are unbuffered.

Inconsistencies checked include:

1. Blocks claimed by more than one inode, or by the free list;
2. Blocks claimed by an inode or the free list outside the range of the file system;
3. Incorrect link counts;
4. Blocks not accounted for anywhere;
5. Bad inode format;
6. Directory checks:
   Files pointing to unallocated inodes,
   Inode numbers out of range,
   Multiply linked directories,
   Link to the parent directory.

Orphaned files (allocated but unreferenced) with non-zero sizes are, with the operator's concurrence, reconnected by placing them in the *lost+found* directory. The name assigned is the inode number. The only restriction is that *lost+found* must exist in the root of the file system being checked, and must have empty slots in which entries can be made. This is accomplished by creating *lost+found*, copying a number of files to it, and then removing them (before *fsck* is executed).

Orphaned directories and files with zero size, with the operator's concurrence, are returned directly to the free list. This will also happen if the *lost+found* directory does not exist.

After *fsck* has checked and fixed the file system, it will store the correct fs_clean flag in the super block if it is not already there (see *fs*[SDF](4)).

You should run a backup prior to running *fsck* for repairs.

DIAGNOSTICS

The diagnostics are intended to be self-explanatory.

WARNINGS

All SDF file systems being checked must be described by a character special device file.

Do not redirect *stdout* or *stderr* to a file on the device being checked. This includes pipes when checking the root volume.

Fsck cannot check devices with a logical block size greater than 4096.

FILES

/etc/checklist	contains the default list of file systems to check

SEE ALSO

fsclean(1M), checklist(4), fs[SDF](4), reboot(1M), stopsys(1M), shutdown(1M).

Series 500 HP-UX System Administrator Manual.

HP-UX Reference

Vol. 2: Sections 1M, 2, 3, 4, 5, and 7

HP 9000 Series 500 Computers
HP-UX Release 5.2

HP Part Number 09000-90010

HEWLETT
PACKARD

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

Printing History

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

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

April 1987...Edition 1

TABLE OF CONTENTS

VOLUME 1

1. Commands

NAME

intro – introduction to system maintenance commands and application programs

DESCRIPTION

This section describes commands that are used chiefly for system maintenance and administration purposes. The commands in this section should be used in conjunction with other sections of this manual, as well as the HP-UX System Administrator Manual for your system.

Command Syntax

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

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

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

DIAGNOSTICS

Upon termination, each command returns two bytes of status, one supplied by the system and giving the cause for termination, and (in the case of "normal" termination) one supplied by the program (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.

SEE ALSO

getopt(1), getopt(3C), hier(5).

HP-UX System Administrator Manual.

The introduction to this manual.

NAME

accept, reject – allow/prevent LP requests

SYNOPSIS

/usr/lib/accept destinations
/usr/lib/reject [–r[reason]] destinations

DESCRIPTION

Accept allows *lp*(1) to accept requests for the named *destinations*. A *destination* can be either a printer or a class of printers. Use *lpstat*(1) to find the status of *destinations*.

Reject prevents *lp*(1) from accepting requests for the named *destinations*. A *destination* can be either a printer or a class of printers. Use *lpstat*(1) to find the status of *destinations*. The following option is useful with *reject:*

–r[*reason*]	Associates a *reason* with preventing *lp*(1) from accepting requests. This *reason* applies to all printers mentioned up to the next **–r** option. *Reason* is reported by *lp*(1) when users direct requests to the named *destinations* and by *lpstat*(1). If the **–r** option is not present or the **–r** option is given without a *reason*, then a default *reason* will be used.

FILES

/usr/spool/lp/*

SEE ALSO

enable(1), lp(1), lpadmin(1M), lpsched(1M), lpstat(1).

INTERNATIONAL SUPPORT

8- and 16-bit data, messages.

NAME

acctdisk, acctdusg, accton, acctwtmp – overview of accounting and miscellaneous accounting commands

SYNOPSIS

/usr/lib/acct/acctdisk

/usr/lib/acct/acctdusg [**−u** file] [**−p** file]

/usr/lib/acct/accton [file]

/usr/lib/acct/acctwtmp "reason"

DESCRIPTION

Accounting software is structured as a set of tools (consisting of both C programs and shell procedures) that can be used to build accounting systems. *Acctsh*(1M) describes the set of shell procedures built on top of the C programs.

Connect time accounting is handled by various programs that write records into **/etc/utmp**, as described in *utmp*(4). The programs described in *acctcon*(1M) convert this file into session and charging records, which are then summarized by *acctmerg*(1M).

Process accounting is performed by the HP-UX system kernel. Upon termination of a process, one record per process is written to a file (normally **/usr/adm/pacct**). The programs in *acctprc*(1M) summarize this data for charging purposes; *acctcms*(1M) is used to summarize command usage. Current process data may be examined using *acctcom*(1M).

Process accounting and connect time accounting (or any accounting records in the format described in *acct*(4)) can be merged and summarized into total accounting records by *acctmerg* (see **tacct** format in *acct*(4)). *Prtacct* (see *acctsh*(1M)) is used to format any or all accounting records.

Acctdisk reads lines that contain user ID, login name, and number of disk blocks and converts them to total accounting records that can be merged with other accounting records.

Acctdusg reads its standard input (usually from **find / −print**) and computes disk resource consumption (including indirect blocks) by login. If **−u** is given, records consisting of those file names for which *acctdusg* charges no one are placed in *file* (a potential source for finding users trying to avoid disk charges). If **−p** is given, *file* is the name of the password file. This option is not needed if the password file is **/etc/passwd**. (See *diskusg*(1M) for more details.)

Accton turns process accounting off if the optional *file* argument is omitted. If *file* is given, it must be the name of an existing file, to which the kernel appends process accounting records (see *acct*(2) and *acct*(4)).

Acctwtmp writes a *utmp*(4) record to its standard output. The record contains the current time and a string of characters that describe the *reason* for writing the record. A record type of ACCOUNTING is assigned (see *utmp*(4)). *Reason* must be a string of 11 or fewer characters, numbers, **$**, or spaces. For example, the following are suggestions for use in reboot and shutdown procedures, respectively:

```
acctwtmp `uname` >> /etc/wtmp
acctwtmp "file save" >> /etc/wtmp
```

FILES

/usr/lib/acct	holds all accounting commands listed in section (1M) of this manual
/usr/adm/pacct	current process accounting file
/etc/passwd	used for login name to user ID conversions
/etc/wtmp	login/logoff history file

SEE ALSO

acctcms(1M), acctcom(1), acctcon(1M), acctmerg(1M), acctprc(1M), acctsh(1M), diskusg(1M), fwtmp(1M), runacct(1M), acct(2), acct(4), utmp(4),

System Accounting chapter in *HP-UX System Administrator Manual.*

NAME

acctcms – command summary from per-process accounting records

SYNOPSIS

/usr/lib/acct/acctcms [options] files

DESCRIPTION

Acctcms reads one or more *files*, normally in the form described in *acct*(4). It adds all records for processes that executed identically-named commands, sorts them, and writes them to the standard output, normally using an internal summary format. The *options* are:

–a	Print output in ASCII rather than in the internal summary format. The output includes command name, number of times executed, total kcore-minutes, total CPU minutes, total real minutes, mean size (in K), mean CPU minutes per invocation, "hog factor", characters transferred, and blocks read and written, as in *acctcom*(1). Output is normally sorted by total kcore-minutes.
–c	Sort by total CPU time, rather than total kcore-minutes.
–j	Combine all commands invoked only once under "***other".
–n	Sort by number of command invocations.
–s	Any file names encountered hereafter are already in internal summary format.
–t	Process all records as total accounting records. The default internal summary format splits each field into prime and non-prime time parts. This option combines the prime and non-prime time parts into a single field that is the total of both, and provides upward compatibility with old (i.e., UNIX System V) style *acctcms* internal summary format records.

The following options may be used only with the **-a** option.

–p	Output a prime-time-only command summary.
–o	Output a non-prime (offshift) time only command summary.

When **–p** and **–o** are used together, a combination prime and non-prime time report is produced. All the output summaries will be total usage except number of times executed, CPU minutes, and real minutes which will be split into prime and non-prime.

A typical sequence for performing daily command accounting and for maintaining a running total is:

```
acctcms file ... >today
cp total previoustotal
acctcms -s today previoustotal >total
acctcms -a -s today
```

SEE ALSO

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

BUGS

Unpredictable output results if **–t** is used on new style internal summary format files, or if it is not used with old style internal summary format files.

NAME

acctcon1, acctcon2 – connect-time accounting

SYNOPSIS

/usr/lib/acct/acctcon1 [options]

/usr/lib/acct/acctcon2

DESCRIPTION

Acctcon1 converts a sequence of login/logoff records read from its standard input to a sequence of records, one per login session. Its input should normally be redirected from **/etc/wtmp**. Its output is ASCII, giving device, user ID, login name, prime connect time (seconds), non-prime connect time (seconds), session starting time (numeric), and starting date and time. The *options* are:

–p Print input only, showing line name, login name, and time (in both numeric and date/time formats).

–t *Acctcon1* maintains a list of lines on which users are logged in. When it reaches the end of its input, it emits a session record for each line that still appears to be active. It normally assumes that its input is a current file, so that it uses the current time as the ending time for each session still in progress. The **–t** flag causes it to use, instead, the last time found in its input, thus assuring reasonable and repeatable numbers for non-current files.

–l *file* *File* is created to contain a summary of line usage showing line name, number of minutes used, percentage of total elapsed time used, number of sessions charged, number of logins, and number of logoffs. This file helps track line usage, identify bad lines, and find software and hardware oddities. Hang-up, termination of *login*(1) and termination of the login shell each generate logoff records, so that the number of logoffs is often three to four times the number of sessions. See *init*(1M) and *utmp*(4).

–o *file* *File* is filled with an overall record for the accounting period, giving starting time, ending time, number of reboots, and number of date changes.

Acctcon2 expects as input a sequence of login session records and converts them into total accounting records (see **tacct** format in *acct*(4)).

EXAMPLES

These commands are typically used as shown below. The file **ctmp** is created only for the use of *acctprc*(1M) commands:

```
acctcon1 -t -l lineuse -o reboots <wtmp | sort +1n +2 >ctmp
acctcon2 <ctmp | acctmerg >ctacct
```

FILES

/etc/wtmp

SEE ALSO

acct(1M), acctcms(1M), acctcom(1), acctmerg(1M), acctprc(1M), acctsh(1M), fwtmp(1M), init(1M), login(1), runacct(1M), acct(2), acct(4), utmp(4).

BUGS

The line usage report is confused by date changes. Use *wtmpfix* (see *fwtmp*(1M)) to correct this situation.

NAME

acctmerg – merge or add total accounting files

SYNOPSIS

/usr/lib/acct/acctmerg [options] [file] . . .

DESCRIPTION

Acctmerg reads its standard input and up to nine additional files, all in the **tacct** format (see *acct*(4)) or an ASCII version thereof. It merges these inputs by adding records whose keys (normally user ID and name) are identical, and expects the inputs to be sorted on those keys. *Options* are:

–a	Produce output in ASCII version of **tacct**.
–i	Input files are in ASCII version of **tacct**.
–p	Print input with no processing.
–t	Produce a single record that totals all input.
–u	Summarize by user ID, rather than user ID and name.
–v	Produce output in verbose ASCII format, with more precise notation for floating point numbers.

EXAMPLES

The following sequence is useful for making "repairs" to any file kept in this format:

```
acctmerg –v <file1 >file2
        edit file2 as desired . . .
acctmerg –i <file2 >file1
```

SEE ALSO

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

NAME

acctprc1, acctprc2 – process accounting

SYNOPSIS

/usr/lib/acct/acctprc1 [**ctmp**]

/usr/lib/acct/acctprc2

DESCRIPTION

Acctprc1 reads input in the form described by *acct*(4), adds login names corresponding to user IDs, then writes for each process an ASCII line giving user ID, login name, prime CPU time (tics), non-prime CPU time (tics), and mean memory size (in memory segment units). If **ctmp** is given, it is expected to contain a list of login sessions, in the form described in *acctcon*(1M), sorted by user ID and login name. If this file is not supplied, it obtains login names from the password file. The information in **ctmp** helps it distinguish among different login names that share the same user ID.

Acctprc2 reads records in the form written by *acctprc1*, summarizes them by user ID and name, then writes the sorted summaries to the standard output as total accounting records.

These commands are typically used as shown below:

```
acctprc1 ctmp </usr/adm/pacct | acctprc2 >ptacct
```

FILES

/etc/passwd

SEE ALSO

acct(1M), acctcms(1M), acctcom(1), acctcon(1M), acctmerg(1M), acctsh(1M), cron(1M), fwtmp(1M), runacct(1M), acct(2), acct(4), utmp(4).

BUGS

Although it is possible to distinguish among login names that share user IDs for commands run normally, it is difficult to do this for those commands run from *cron*(1M), for example. More precise conversion can be done by faking login sessions on the console via the *acctwtmp* program in *acct*(1M).

CAVEAT

A memory segment of the mean memory size is a unit of measure for the number of bytes in a logical memory segment on a particular processor.

HARDWARE DEPENDENCIES

Series 500

Each memory segment unit contains 512-bytes. Therefore, memory usage statistics are rounded up to 512-byte units.

NAME

chargefee, ckpacct, dodisk, lastlogin, monacct, nulladm, prctmp, prdaily, prtacct, runacct, shutacct, startup, turnacct – shell procedures for accounting

SYNOPSIS

/usr/lib/acct/chargefee login-name number

/usr/lib/acct/ckpacct [blocks]

/usr/lib/acct/dodisk [-o] [files ...]

/usr/lib/acct/lastlogin

/usr/lib/acct/monacct number

/usr/lib/acct/nulladm file

/usr/lib/acct/prctmp

/usr/lib/acct/prdaily [-l] [-c] [mmdd]

/usr/lib/acct/prtacct file ["heading"]

/usr/lib/acct/runacct [mmdd] [mmdd state]

/usr/lib/acct/shutacct ["reason"]

/usr/lib/acct/startup

/usr/lib/acct/turnacct on | **off** | **switch**

DESCRIPTION

Chargefee can be invoked to charge a *number* of units to *login-name*. A record is written to **/usr/adm/fee**, to be merged with other accounting records during the night.

Ckpacct should be initiated via *cron*(1M). It periodically checks the size of **/usr/adm/pacct**. If the size exceeds *blocks*, 1000 by default, *turnacct* will be invoked with argument *switch*. If the number of free disk blocks in the **/usr** file system falls below 500, *ckpacct* will automatically turn off the collection of process accounting records via the **off** argument to *turnacct*. When at least this number of blocks is restored, the accounting will be activated again. This feature is sensitive to the frequency at which *ckpacct* is executed, usually by *cron*.

Dodisk should be invoked by *cron* to perform the disk accounting functions. By default, it will do disk accounting on the special files in **/etc/checklist.** If the **–o** flag is used, it will do a slower version of disk accounting by login directory. *Files* specify the one or more filesystem names where disk accounting will be done. If *files* are used, disk accounting will be done on these filesystems only. If the **–o** flag is used, *files* should be mount points of mounted filesystem. If omitted, they should be the special file names of mountable filesystems.

Lastlogin is invoked by *runacct* to update **/usr/adm/acct/sum/loginlog**, which shows the last date on which each person logged in.

Monacct should be invoked once each month or each accounting period. *Number* indicates which month or period it is. If *number* is not given, it defaults to the current month (01–12). This default is useful if *monacct* is to executed via *cron*(1M) on the first day of each month. *Monacct* creates summary files in **/usr/adm/acct/fiscal** and restarts summary files in **/usr/adm/acct/sum**.

Nulladm creates *file* with mode 664 and insures that owner and group are **adm**. It is called by various accounting shell procedures.

Prctmp can be used to print the session record file (normally **/usr/adm/acct/nite/ctmp** created by *acctcon1* (see *acctcon*(1M)).

Prdaily is invoked by *runacct* to format a report of the previous day's accounting data. The report resides in **/usr/adm/acct/sum/rprt***mmdd* where *mmdd* is the month and day of the report. The current daily accounting reports may be printed by typing *prdaily*. Previous days' accounting reports can be printed by using the *mmdd* option and specifying the exact report date desired. The **–l** flag prints a report of exceptional usage by login id for the specifed date. Previous daily reports are cleaned up and therefore inaccessible after each invocation of *monacct*. The **–c** flag prints a report of exceptional resource usage by command, and may be used on current day's accounting data only.

Prtacct can be used to format and print any total accounting (**tacct**) file.

Runacct performs the accumulation of connect, process, fee, and disk accounting on a daily basis. It also creates summaries of command usage. For more information, see *runacct*(1M).

Shutacct should be invoked during a system shutdown (usually in **/etc/shutdown**) to turn process accounting off and append a "reason" record to **/etc/wtmp**.

Startup should be called by **/etc/rc** to turn the accounting on whenever the system is brought up.

Turnacct is an interface to *accton* (see *acct*(1M)) to turn process accounting **on** or **off**. The **switch** argument turns accounting off, moves the current **/usr/adm/pacct** to the next free name in **/usr/adm/pacct***incr* (where *incr* is a number starting with **1** and incrementing by one for each additional **pacct** file), then turns accounting back on again. This procedure is called by *ckpacct* and thus can be taken care of by the *cron* and used to keep **pacct** to a reasonable size.

FILES

/usr/lib/acct	holds all accounting commands listed in section (1M) of this manual
/usr/adm/fee	accumulator for fees
/usr/adm/acct/nite	working directory
/usr/adm/pacct	current file for per-process accounting
/usr/adm/pacct*	used if pacct gets large and during execution of daily accounting procedure
/usr/lib/acct/ptecms.awk	contains the limits for exceptional usage by command name
/usr/lib/acct/ptelus.awk	contains the limits for exceptional usage by login id
/usr/adm/acct/sum	summary directory, should be saved
/etc/wtmp	login/logoff summary

SEE ALSO

acct(1M), acctcms(1M), acctcom(1), acctcon(1M), acctmerg(1M), acctprc(1M), cron(1M), diskusg(1M), fwtmp(1M), runacct(1M), acct(2), acct(4), utmp(4).

NAME

autobkup – backup or archive file system

SYNOPSIS

/etc/autobkup [–**archive**] [–**fsck**]

Remarks:

This manual entry describes *autobkup* as implemented on the Series 500 computers.

Autobkup is only supported on the HP 90000 Series 500.

DESCRIPTION

Autobkup uses *find*(1) and *cpio*(1) to save on the default tape drive (**/dev/rct**, which must be a tape autochanger) a *cpio* archive of all files which have been modified since the modification time of **/etc/archivedate**. *Autobkup* should be periodically invoked by *cron*(1M) at night, or when the system is otherwise idle.

The –**archive** option causes *autobkup* to save all files, regardless of their modification date, then update **/etc/archivedate** using *touch*(1).

The –**fsck** option causes *autobkup* to start a file system consistency check (without correction) after the backup is complete. This is the normal mode of nightly operation. For correct results, it is important that the system be effectively single-user while *fsck* is running, especially if it is allowed to automatically fix whatever inconsistencies it finds. *Autobkup* does not ensure that the system is single-user.

Autobkup is an enhanced version of *backup*(1M) and supports tape autochangers such as the HP 35401. *Autobkup* executes a background process */etc/bkserver* which intercepts and responds to *tcio*'s prompts for a new special file name and allows the next tape in the magazine to be loaded automatically by the tape autochanger.

You should edit */etc/autobkup* to customize it for your system.

The following parameters are supported and can be customized:

backupdirs	specifies which directories to recursively back up (usually ., meaning all directories);
backuplog	file name where start and finish times, block counts, and error messages are logged;
archive	file name whose date is the date of the last archive;
remind	file name that is checked by **/etc/profile** to remind the next person who logs in to change the backup tape;
rootdev	character special file of root device for *fsck*;
outdev	specifies the output device for the backed-up files.
masterpty	filename of the master side of the pseudo-terminal.
slavepty	filename of the slave side of the pseudo-terminal.
fscklog	file name where start and finish times and *fsck* output is logged.
mytty	the terminal from which attributes are taken for the pseudo-terminal.

You may want to make other changes, such as whether or not *fsck* does automatic correction (according to its arguments), where *cpio* output is directed, other information logging, etc.

In all cases, the output from *autobkup* is a normal *cpio* archive file (or volume) which can be read using *tcio* (if used to generate the backup) and *cpio* with the –**c** option.

To run *autobkup* from *cron*, use a crontab entry similar to this:

```
* 2 * * 1-6 ( cd / ; /etc/autobkup ) >/dev/null 2>&1
```

FILES

/etc/bkserver
/etc/archivedate
parameterized file names

SEE ALSO

backup(1M), cpio(1), find(1), touch(1), cron(1M), fsck(1M).

See the *HP-UX System Administrator Manual* provided with your system for recommended ways to backup and restore your file system.

BUGS

Refer to **BUGS** in *cpio*(1).

Autobkup cannot archive file systems that are larger than the capacity of a single magazine of tapes. For larger file systems, duplicate the *autobkup* script and customize each copy to separately archive the file systems mounted on separate mass storage devices.

If *autobkup* is left running overnight and runs out of tapes, *autobkup* terminates, leaving the *find* process still waiting. You need to kill this process when you return.

Under some error conditions */etc/bkserver* terminates, leaving the *find*, *cpio* and *tcio* processes still waiting. You need to kill these processes when you return.

NAME

backup – backup or archive file system

SYNOPSIS

/etc/backup [**–archive**] [**–fsck**]

Remarks:

This manual page describes *backup* as it is implemented on Series 200 and 300 computers. Refer to other *backup*(1M) manual pages for information valid for other implementations.

DESCRIPTION

Backup uses *find*(1) and *cpio*(1) to save a *cpio* archive of all files which have been modified since the modification time of **/etc/archivedate** on the default tape drive (**/dev/rct**). *Backup* should be periodically invoked to ensure adequate file backup.

The **–archive** option causes *backup* to save all files, regardless of their modification date, and then update **/etc/archivedate** using *touch*(1).

Backup prompts you to mount a new tape and continue if there is no more room on the current tape. Note that this prompting does not occur if you are running *backup* from *cron*(1M).

The **–fsck** option causes *backup* to start a file system consistency check (without correction) after the backup is complete. For correct results, it is important that the system be effectively single-user while *fsck* is running, especially if **–fsck** is allowed to automatically fix whatever inconsistencies it finds. *Backup* does not ensure that the system is single-user.

You may edit **/etc/backup** to "customize" it for your system. For example, *backup* uses *tcio*(1) with *cpio* to backup your files on an HP Command Set 80 disc's streaming tape. You will need to modify *backup* to use *cpio*(1) if you want to access a standard HP Tape Drive.

Several local values are used which can be customized:

backupdirs	specifies which directories to recursively back up (usually /, meaning all directories);
backuplog	file name where start and finish times, block counts, and error messages are logged;
archive	file name whose date is the date of the last archive;
remind	file name that is checked by **/etc/profile** to remind the next person who logs in to change the backup tape;
outdev	specifies the output device for the backed-up files;
fscklog	file name where start and finish times and *fsck* output is logged.

You may want to make other changes, such as whether or not *fsck* does automatic correction (according to its arguments), where *cpio* output is directed, other information logging, etc.

In all cases, the output from *backup* is a normal *cpio* archive file (or volume) which can be read using *tcio* and *cpio* with the **c** option.

FILES

/etc/archivedate
parameterized file names

SEE ALSO

cpio(1), find(1), touch(1), cron(1M), fsck(1M).

BUGS

Refer to **BUGS** in *cpio*(1).

When *cpio* runs out of tape, it sends an error to *stderr* and demands a new special file name from /dev/tty.

To continue, rewind the tape, mount the new tape, type the name of the new special file at the system console, and press **RETURN**.

If *backup* is left running overnight and the tape runs out, *backup* terminates, leaving the *find* process still waiting. You need to kill this process when you return.

NAME

backup – backup or archive file system

SYNOPSIS

/etc/backup [**–archive**] [**–fsck**]

Remarks: This manual page describes *backup* as it is implemented on the Series 500 computers. Refer to other *backup*(1M) manual pages for information valid for other implementations.

DESCRIPTION

Backup uses *find*(1) and *cpio*(1) to save on the default tape drive (**/dev/rmt79*xx***) a *cpio* archive of all files which have been modified since the modification time of **/etc/archivedate**. *Backup* should be periodically invoked by *cron*(1M) at night, or when the system is otherwise idle.

The **–archive** option causes *backup* to save all files, regardless of their modification date, and then update **/etc/archivedate** using *touch*(1).

Backup prompts you to mount a new tape and continue if there is no more room on the current tape. Note that this prompting does not occur if you are running *backup* from *cron*(1M).

The **–fsck** option causes *backup* to start a file system consistency check (without correction) after the backup is complete. This is the normal mode of nightly operation. For correct results, it is important that the system be effectively single-user while *fsck* is running, especially if it is allowed to automatically fix whatever inconsistencies it finds. *Backup* does not ensure that the system is single-user.

You should edit **/etc/backup** to "customize" it for your system. For example, *backup* uses *tcio*(1) by default. You will need to modify *backup* to use *cpio*(1) if you want to access a raw device.

Several parameters are used which can be customized:

backupdirs	specifies which directories to recursively back up (usually /, meaning all directories);
backuplog	file name where start and finish times, block counts, and error messages are logged;
archive	file name whose date is the date of the last archive;
remind	file name that is checked by **/etc/profile** to remind the next person who logs in to change the backup tape;
rootdev	list of places for *fsck* (usually a character special file that points to the root device);
fscklog	file name where start and finish times and *fsck* output is logged.

You may want to make other changes, such as whether or not *fsck* does automatic correction (according to its arguments), where *cpio* output is directed, other information logging, etc.

In all cases, the output from *backup* is a normal *cpio* archive file (or volume) which can be read using *tcio* (if used to generate the backup) and *cpio* with the **–c** option.

FILES

/etc/archivedate
parameterized file names

SEE ALSO

cpio(1), find(1), touch(1), cron(1M), fsck(1M).

BUGS

Refer to **BUGS** in *cpio*(1).

When *cpio* runs out of tape, it sends an error to *stderr* (which is logged, so it does not appear on your CRT), and demands a new special file name from /dev/tty. To continue, rewind the tape, mount the new tape, type the name of the new special file at the system console, and press **RETURN**.

If *backup* is left running overnight and the tape runs out, *backup* terminates, leaving the *find* process still waiting. You need to kill this process when you return.

NAME
bifdf – report number of free disk blocks

SYNOPSIS
bifdf [**–t**] [**–f**] [file-systems]

DESCRIPTION
Bifdf prints out the number of free blocks and free inodes available for online Bell file systems by examining the counts kept in the super-blocks. *File-systems* can be specified by device name, e.g., **/dev/dsk/1s0**.

The **–t** flag causes the total allocated block figures to be reported as well.

If the **–f** flag is given, only an actual count of the blocks in the free list is made (free inodes are not reported). With this option, *bifdf* will report on raw devices.

AUTHOR
Bifdf was developed by HP.

SEE ALSO
biffsck(1M), df(1M), bif(4).

NAME

biffsck – Bell file system consistency check and interactive repair

SYNOPSIS

biffsck [**–y**] [**–n**] [**–s**X] [–SX] [**–t**filename] [file-system] ...

DESCRIPTION

Biffsck audits and interactively repairs inconsistent conditions in a Bell file system. If the file system is consistent, the number of files, number of blocks used, and number of blocks free are reported. If the file system is inconsistent, the operator is prompted for concurrence before each correction is attempted. It should be noted that most corrective actions will result in some loss of data. The amount and severity of data lost can be determined from the diagnostic output. The default action for each consistency correction is to wait for the operator to respond **yes** or **no**. If the operator does not have write permission, *biffsck* will default to the **–n** option described below.

The following flags are interpreted by *biffsck*.

–y Assume a **yes** response to all questions asked by *biffsck*.

–n Assume a **no** response to all questions asked by *biffsck*, and do not open the file system for writing.

–s*X* Ignore the actual free list and unconditionally reconstruct a new one by rewriting the super-block of the file system. The file system should be unmounted while this is done.

The **–s***X* option allows for creating an optimal free-list organization. The following forms of *X* are supported for the following devices:

–sBlocks-per-cylinder:Blocks-to-skip

If *X* is not given, the values used when the file system was created are used. If these values were not specified, the default values shown below are used:

An HP 7908A uses 35:2;
An HP 7933A uses 23:15;
An HP 7911A uses 16:12;
An HP 7912A uses 16:12;
An HP 7914A uses 16:12;
The default for *biffsck*(1M) is 400:9;
The default for *bifmkfs*(1M) is 500:3.

–S*X* Conditionally reconstruct the free list. This option is like **–s***X* above except that the free list is rebuilt only if there were no discrepancies discovered in the file system. Using **–S** will force a **no** response to all questions asked by *biffsck*. This option is useful for forcing free list reorganization on uncontaminated Bell file systems.

–t If *biffsck* cannot obtain enough memory to keep its tables, it uses a scratch file. If the **–t** option is specified, the file named in the next argument is used as the scratch file, if needed. Without the **–t** flag, *biffsck* will prompt the operator for the name of the scratch file. The file chosen should not be on the file system being checked. If the file does not exist, *biffsck* will create it. If the scratch file is not a special file, it is removed when *biffsck* completes.

File-system is a device file name on which the file system to be checked resides, i.e., **/dev/dsk/1s0**.

Inconsistencies checked are as follows:

1. Blocks claimed by more than one inode or the free list.
2. Blocks claimed by an inode or the free list outside the range of the file system.

3. Incorrect link counts.

4. Size checks:
Incorrect number of blocks.
Directory size not 16-byte aligned.

5.
Bad inode format.

6.
Blocks not accounted for anywhere.

7.
Directory checks:
File pointing to unallocated inode.
Inode number out of range.

8.
Super Block checks:
More than 65536 inodes.
More blocks for inodes than there are in the file system.

9.
Bad free block list format.

10.
Total free block and/or free inode count incorrect.

Orphaned files and directories (allocated but unreferenced) are, with the operator's concurrence, reconnected by placing them in the **/lost+found** directory on the BIF volume. The name assigned is the inode number. The only restriction is that the directory **lost+found** must preexist in the root of the file system being checked and must have empty slots in which entries can be made. This is accomplished by making **lost+found**, copying a number of files to the directory (optimally in multiples of 64), and then removing them before *biffsck* is executed.

Biffsck can check file systems on both raw and blocked devices. Checking raw devices is almost always faster, but should not be used on a mounted file system.

RETURNS

The diagnostics produced by *biffsck* are intended to be self-explanatory.

WARNINGS

Inode numbers for **.** and **..** in each directory should be checked for validity.

AUTHOR

Bif was developed by HP.

SEE ALSO

bif(4).

NAME

biffsdb – Bell file system debugger

SYNOPSIS

biffsdb special [–]

DESCRIPTION

Biffsdb can be used to patch up a damaged Bell file system after a crash/failure. It has conversions to translate block and i-numbers into their corresponding disk addresses. Also included are mnemonic offsets to access different parts of an inode. These greatly simplify the process of correcting control block entries or descending the Bell file system tree.

Biffsdb contains several error checking routines to verify inode and block addresses. These can be disabled if necessary by invoking *biffsdb* with the optional – argument or by the use of the **O** symbol. (*Biffsdb* reads the i-size and f-size entries from the superblock of the file system as the basis for these checks.)

Numbers are considered decimal by default. Octal numbers must be prefixed with a zero. During any assignment operation, numbers are checked for a possible truncation error due to a size mismatch between source and destination.

Biffsdb reads a block at a time and will therefore work with raw as well as block I/O. A buffer management routine is used to retain commonly used blocks of data in order to reduce the number of read system calls. All assignment operations result in an immediate write-through of the corresponding block.

The symbols recognized by *biffsdb* are:

#	absolute address
i	convert from i-number to inode address
b	convert to block address
d	directory slot offset
+,–	address arithmetic
q	quit
>,<	save, restore an address
=	numerical assignment
=+	incremental assignment
=–	decremental assignment
="	character string assignment
O	error checking flip flop
p	general print facilities
f	file print facility
B	byte mode
W	word mode
D	double word mode
!	escape to shell

The print facilities generate a formatted output in various styles. The current address is normalized to an appropriate boundary before printing begins. It advances with the printing and is left at the address of the last item printed. The output can be terminated at any time by typing the delete character. If a number follows the **p** symbol, that many entries are printed. A check is made to detect block boundary overflows since logically sequential blocks are generally not physically sequential. If a count of zero is used, all entries to the end of the current block are printed. The print options available are:

i	print as inodes
d	print as directories

o	print as octal words
e	print as decimal words
c	print as characters
b	print as octal bytes

The **f** symbol is used to print data blocks associated with the current inode. If followed by a number, that block of the file is printed. (Blocks are numbered from zero.) The desired print option letter follows the block number, if present, or the **f** symbol. This print facility works for small as well as large files. It checks for special devices and that the block pointers used to find the data are not zero.

Dots, tabs and spaces can be used as function delimiters but are not necessary. A line with just a new-line character will increment the current address by the size of the data type last printed. That is, the address is set to the next byte, word, double word, directory entry or inode, allowing the user to step through a region of a file system. Information is printed in a format appropriate to the data type. Bytes, words and double words are displayed with the octal address followed by the value in octal and decimal. A **.B** or **.D** is appended to the address for byte and double word values, respectively. Directories are printed as a directory slot offset followed by the decimal i-number and the character representation of the entry name. Inodes are printed with labeled fields describing each element.

The following mnemonics are used for inode examination and refer to the current working inode:

md	mode
ln	link count
uid	user ID number
gid	group ID number
s0	high byte of file size
s1	low word of file size
a#	data block numbers (0 – 12)
at	access time
mt	modification time
maj	major device number
min	minor device number

EXAMPLES

386i	prints i-number 386 in an inode format. This now becomes the current working inode.
ln=4	changes the link count for the working inode to 4.
ln=+1	increments the link count by 1.
fc	prints, in ASCII, block zero of the file associated with the working inode.
2i.fd	prints the first 32 directory entries for the root inode of this file system.
d5i.fc	changes the current inode to that associated with the 5th directory entry (numbered from zero) found from the above command. The first 512 bytes of the file are then printed in ASCII.
1b.p0o	prints the superblock of this file system in octal.
2i.a0b.d7=3	changes the i-number for the seventh directory slot in the root directory to 3. This example also shows how several operations can be combined on one command line.
d7.nm="*name*"	changes the *name* field in the directory slot to the given string. Quotes are optional when used with **nm** if the first character is alphabetic.

AUTHOR

Biffsdb was developed by HP.

SEE ALSO

bif(4), biffsck(1M).

NAME

bifmkfs – construct a Bell file system

SYNOPSIS

bifmkfs special blocks[:inodes] [gap blocks]
bifmkfs special proto [gap blocks]

DESCRIPTION

Bifmkfs constructs a Bell file system by writing on the special file according to the directions found in the remainder of the command line. If the second argument is given as a string of digits, *bifmkfs* builds a file system with a single empty directory on it. The size of the file system is the value of *blocks* interpreted as a decimal number. The boot program is left uninitialized. If the optional number of inodes is not given, the default is the number of blocks divided by 4.

If the second argument is a file name that can be opened, *bifmkfs* assumes it to be a prototype file *proto*, and will take its directions from that file. The prototype file contains tokens separated by spaces or new-lines. The first token is the name of a file to be copied onto block zero as the bootstrap program. The second token is a number specifying the size of the created file system. Typically it will be the number of blocks on the device, perhaps diminished by space for swapping. The next token is the i-list size in blocks The next set of tokens comprise the specification for the root file. File specifications consist of tokens giving the mode, the user ID, the group ID, and the initial contents of the file. The syntax of the contents field depends on the mode.

The mode token for a file is a 6 character string. The first character specifies the type of the file. (The characters **–bcd** specify regular, block special, character special and directory files respectively.) The second character of the type is either **u** or – to specify set-user-id mode or not. The third is **g** or – for the set-group-id mode. The rest of the mode is a three digit octal number giving the owner, group, and other read, write, execute permissions, see *bifchmod*(1).

Two decimal number tokens come after the mode; they specify the user and group ID's of the owner of the file.

If the file is a regular file, the next token is a path name that designates the file from which the contents and size are copied. If the file is a block or character special file, two decimal number tokens follow which give the major and minor device numbers. If the file is a directory, *bifmkfs* makes the entries **.** and **..** and then reads a list of names and (recursively) file specifications for the entries in the directory. The scan is terminated with the token **$**.

A sample prototype specification follows:

```
/stand/diskboot
4872 110
d--777 3 1
usr     d--777 3 1
        sh      ---755 3 1 /bin/sh
        ken     d--755 6 1
                $
        b0      b--644 3 1 0 0
        c0      c--644 3 1 0 0
        $
$
```

In both command syntaxes, the rotational *gap* and the number of *blocks* can be specified. (For RP04 type drives, these numbers should be 7 and 418.)

EXAMPLES

To put a Bell file system on a disk with 770 1K blocks of capacity:

```
          bifmkfs /dev/rdsk/1s0 770
```

where **/dev/rdsk/1s0** is the device special file for the micro floppy.

WARNINGS

If a prototype is used, it is not possible to initialize a file with second- or third-level indirects.

AUTHOR

Bif was developed by HP.

SEE ALSO

bif(4).

NAME

boot – bootstrap process

DESCRIPTION

The bootstrap process on the Series 800 Model 840 involves the execution of three software components: *pdc(1M), isl(1M),* and *hpuxboot(1M).* After the processor is **RESET,** *pdc,* the processor dependent code or firmware, performs a self-test and initializes the processor. It then loads and transfers control to *isl,* the operating system independent initial system loader. *Isl* in turn loads and transfers control to the *hpuxboot* utility, the HP-UX specific bootstrap loader. Lastly, *hpuxboot* downloads the HP-UX kernel object file from an HP-UX file system and then transfers control to the loaded kernel image.

SEE ALSO

hpuxboot(1M), isl(1M), pdc(1M).

NAME

brc, bcheckrc, rc, powerfail – system initialization shell scripts

SYNOPSIS

/etc/brc

/etc/bcheckrc

/etc/rc

/etc/powerfail

DESCRIPTION

These shell procedures are executed via entries in */etc/inittab* by *init*(1M). *Bcheckrc* and *brc* are executed when the system is changed out of *SINGLE USER* mode. *Rc* is executed upon entering a new, numbered, run-level. *Powerfail* is executed whenever a system power failure is detected.

The *brc* procedure clears the mounted file system table, **/etc/mnttab** (see *mnttab*(4)), and loads any programmable micro-processors with their appropriate scripts.

The *bcheckrc* procedure performs all the necessary consistency checks to prepare the system to change into multi-user mode. It will prompt to set the system date and to check the file systems with *fsck*(1M).

The *rc* procedure starts all system daemons before the terminal lines are enabled for multi-user mode. In addition, file systems are mounted and accounting, error logging, system activity logging and the Remote Job Entry (RJE) system are activated in this procedure.

The *powerfail* procedure is invoked when the system detects a power failure condition. Its chief duty is to reload any programmable micro-processors with their appropriate scripts, if suitable. It also logs the fact that a power failure occurred.

SEE ALSO

fsck(1M), init(1M), shutdown(1M), inittab(4), mnttab(4).

NAME

captoinfo – convert a termcap description into a terminfo description

SYNOPSIS

captoinfo [**–1v**] [**–w***n*] [*filenames*]

DESCRIPTION

Captoinfo looks in *filenames* for *termcap*(3X) descriptions. For each one found, an equivalent *terminfo*(4) description is written to standard output along with any comments found. The short two letter name at the beginning of the list of names in a *termcap* entry, a hold-over from Version 6 UNIX, is removed. Any description that is expressed relative to another description (as specified in the termcap *tc=* field) is reduced to the minimum superset before output.

If no *filename* is given, the environment variable TERMCAP is used for the filename or entry. If TERMCAP is a full pathname to a file, only the terminal whose name is specified in the environment variable TERM is extracted from that file. If the environment variable TERMCAP is not set, the file **/etc/termcap** is read.

Options

–1 Print one field per line. If this option is not selected multiple fields are printed on each line up to a maximum width of 60 characters.

–v Print (verbose) tracing information as the program runs. Additional **–v** options print more information (for example **-v -v -v** or **-vvv**).

–w*n* Change the output width to *n* characters.

WARNINGS

Certain *termcap* defaults are assumed to be true. For example, the bell character (terminfo *bel*) is assumed to be *^G*. The linefeed capability (termcap *nl*) is assumed to be the same for both *cursor_down* and *scroll_forward* (terminfo *cud1* and *ind*, respectively.) Padding information is assumed to belong at the end of the string.

The algorithm used to expand parameterized information for *termcap* fields such as *cursor_position* (termcap *cm*, terminfo *cup*) sometimes produces a string which, though technically correct, may not be optimal. In particular, the rarely used termcap operation **%n** produces strings that are especially long. Most occurrences of these non-optimal strings are flagged with a warning message, and may need to be recoded by hand.

HP only supports terminals listed on the current list of supported devices. However, non-supported and supported terminals can be in the terminfo database. If you use such non-supported terminals, they may not work correctly.

DIAGNOSTICS

tgetent failed with return code n (reason).
The termcap entry is not valid. In particular, check for an invalid 'tc=' entry.

unknown type given for the termcap code 'cc'.
The termcap description had an entry for 'cc' whose type was not boolean, numeric or string.

wrong type given for the boolean (numeric, string) termcap code 'cc'.
The boolean termcap entry 'cc' was entered as a numeric or string capability.

the boolean (numeric, string) termcap code 'cc' is not a valid name.
An unknown termcap code was specified.

tgetent failed on TERM=term.
The terminal type specified could not be found in the termcap file.

TERM=term: cap cc (info ii) is NULL: REMOVED
The termcap code was specified as a null string. The correct way to cancel an entry is

with an '@', as in ':bs@:'. Giving a null string could cause incorrect assumptions to be made by any software that uses termcap or terminfo.

a function key for 'cc' was specified, but it already has the value 'vv'.
: When parsing the 'ko' capability, the key 'cc' was specified as having the same value as the capability 'cc', but the key 'cc' already had a value assigned to it.

the unknown termcap name 'cc' was specified in the 'ko' termcap capability.
: A key that could not be handled was specified in the 'ko' capability.

the vi character 'v' (info 'ii') has the value 'xx', but 'ma' gives 'n'.
: The 'ma' capability specified a function key with a value different from that specified in another setting of the same key.

the unknown vi key 'v' was specified in the 'ma' termcap capability.
: A vi key unknown to *captoinfo* was specified in the 'ma' capability.

Warning: termcap sg (nn) and termcap ug (nn) had different values.
: Terminfo assumes that the sg (now xmc) and ug values were the same.

Warning: the string produced for 'ii' may be inefficient.
: The parameterized string being created should be rewritten by hand.

Null termname given.
: The terminal type was null. This occurs when $TERM is null or not set.

cannot open "%s" for reading.
: The specified file could not be opened.

Warning: cannot translate <capability> *(unsupported in terminfo).*
: This termcap capability is no longer supported in terminfo, and therefore cannot be translated.

AUTHOR

Captoinfo was developed by AT&T.

SEE ALSO

curses (3X), termcap (3X), terminfo (4), tic (1M), untic (1M).

NAME

catman – create the cat files for the manual

SYNOPSIS

/etc/catman [**–p**] [**–n**] [**–w**] [sections]

DESCRIPTION

Catman creates the preformatted versions of the online manual from the *nroff* input files. Each manual page is examined and those whose preformatted versions are missing or out of date are recreated. If any changes are made, *catman* will recreate the **/usr/lib/whatis** database.

If there is one parameter not starting with a '–', it is taken to be a list of manual sections to look in. For example

catman 123

will cause the updating to only happen to manual sections 1, 2, and 3.

Options:

–n	prevents creation of **/usr/lib/whatis .**
–p	prints what would be done instead of doing it.
–w	causes only the **/usr/lib/whatis** database to be created. No manual reformatting is done.

FILES

/usr/man/man*/*	raw (*nroff* input) manual pages
/usr/man/cat*/*	formatted manual pages
/usr/local/man/man*/*	
/usr/local/man/cat*/*	
/usr/contrib/man/man*/*	
/usr/contrib/man/cat*/*	
/usr/lib/mkwhatis	commands to make whatis database

AUTHOR

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

SEE ALSO

man(1).

NAME

catman – create the cat files for the manual

SYNOPSIS

/etc/catman [**−p**] [**−n**] [**−w**] [sections]

DESCRIPTION

Catman creates the formatted versions of the online manual from the *nroff* source files. Each manual page is examined and those whose formatted versions are missing or out of date are recreated. If the cat*.Z directory exists, *Catman* compresses the formatted version and puts it there. Otherwise, *Catman* puts the formatted version in the cat* directory. If any changes are made, *catman* will recreate the **/usr/lib/whatis** database.

If there is one parameter not starting with a '–', it is taken to be a list of manual sections to look in. For example

catman 123

will cause the updating to only happen to manual sections 1, 2, and 3.

Options:

−n	prevents creation of **/usr/lib/whatis .**
−p	prints what would be done instead of doing it.
−w	causes only the **/usr/lib/whatis** database to be created. No manual reformatting is done.

FILES

/usr/man/man*[.Z]/*	raw (*nroff* source) manual pages [compressed]
/usr/man/cat*[.Z]/*	formatted manual pages [compressed]
/usr/local/man/man*[.Z]/*	
/usr/local/man/cat*[.Z]/*	
/usr/contrib/man/man*[.Z]/*	
/usr/contrib/man/cat*[.Z]/*	
/usr/lib/mkwhatis	commands to make whatis database

AUTHOR

Catman was developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science and by the Hewlett-Packard Company.

SEE ALSO

man(1), compress(1).

NAME

chroot – change root directory for a command

SYNOPSIS

/etc/chroot newroot command

DESCRIPTION

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

Notice that:

```
chroot newroot command >x
```

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

Command includes both the command name and any arguments.

This command is restricted to the super-user.

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

WARNINGS

Command cannot be in a shell script.
One should exercise extreme caution when referencing special files in the new root file system.
Chroot does not search **PATH** for the location of *command*, so the absolute path name of *command* must be given.

SEE ALSO

chdir(2), chroot(2).

INTERNATIONAL SUPPORT

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

NAME

chsys – change to different operating system or version

SYNOPSIS

/etc/chsys sysname

Remarks:

Chsys is implemented on the Series 500 only.

DESCRIPTION

Chsys is a shell script that enables you to boot a different operating system, or a different version of the same operating system, using only one boot area on one disc. *Sysname* is one of a number of operating system names defined within *chsys*. *Chsys* uses *oscp*(1M) to rebuild the boot area on */dev/rhd* with the selected system, reading from ordinary files containing operating system code. *Chsys* then invokes *osck*(1M) to confirm that the new system is "healthy". (Note that *osck* performs a redundant check, so its invocation in *chsys* may be removed if you want to save time.)

Chsys invokes *oscp* as quietly as possible. *Chsys* causes *oscp* to read the new system ID string from a file selected by the *sysname* given, and redirects the output from *oscp* to */dev/null*. If *oscp* and *osck* are successful, *chsys* calls *stopsys*(1M) to switch to the new operating system. Note that *oscp* and *osck* together can take longer than a minute to run. During this time, *chsys* keeps you informed as to what actions are being taken.

If you simply want to re-boot the operating system already in the boot area, do **not** use *chsys*. Instead, invoke *stopsys*(1M) directly.

If you want to allocate and use several boot areas on several discs, see *osmgr*(1M).

You should modify *chsys* to localize it for your system. You may want to add or delete available *sysnames*, change the names or meanings of *sysnames*, change the name of the character special file (*/dev/rhd*) which points to the boot volume, etc. *Chsys* recognizes four default *sysnames*. They stand for:

HP-UX Model 520 single-user minimal system;
HP-UX Model 520 single-user complete system;
BASIC minimal system;
BASIC complete system.

These *sysnames* serve as examples for any others you may want to add. They may or may not be useful to you.

Chsys should only be invoked by the effective super-user unless both of the following are true:

the special file which points to the boot device must be readable and writable by whoever invokes *chsys*;

the *stopsys* command must be owned by root and have the set-user-ID bit set.

If either of the above are not true, either the *oscp* or the *stopsys* command will fail.

Chsys must be invoked with a **$PATH** that includes the directories containing the *oscp*, *osck*, *stopsys*, and *echo* commands.

RETURN VALUES

If any of the invoked commands fails, *chsys* writes a message to standard error and exits with the same return value as that returned by the unsuccessful command. *Chsys* returns 1 if invoked improperly.

SEE ALSO

sh(1), osmgr(1M), shutdown(1M), stopsys(1M), sync(1M).

WARNINGS

Chsys does not check that the system is idle, and it does not notify all users that the system is going down. You should usually execute *shutdown*(1M) before executing *chsys.*

Chsys does not ask you to confirm that the intended operating system or version has been selected before the system is re-booted. However, *osck* ensures that the system is rebootable, and *stopsys* performs a *sync*(1M). Note that new operating systems built in the boot area by *oscp* are always marked as loadable (see *osmark*(1M)).

NAME

clri – clear inode

SYNOPSIS

/etc/clri file-system i-number ...

DESCRIPTION

Clri writes zeros on the inode numbered *i-number*. *File-system* must be a special file name referring to a device containing a file system. After *clri* is executed, any blocks in the affected file will show up as "missing" in an *fsck*(1M) of the *file-system*. This command should only be used in emergencies and extreme care should be exercised.

Read and write permission is required on the specified *file-system* device. The inode 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 zero out an inode which does appear in a directory, care should be taken to track down the entry and remove it. Otherwise, when the inode 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 inode, so the whole cycle is likely to be repeated again and again.

SEE ALSO

fsck[BFS](1M), fsck[HFS](1M), fsdb[BFS](1M), fsdb[HFS](1M), ncheck[non-SDF](1M), fs[BFS](4), fs[HFS](4).

BUGS

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

NAME

clrsvc – clear x25 switched virtual circuit

SYNOPSIS

clrsvc line pad-type

DESCRIPTION

Clrsvc clears any virtual circuit that might be established on the *line* specified. The *pad-type* indicates to *clrsvc* what *opx25* script to run from **/usr/lib/uucp/X25**.

HARDWARE DEPENDENCIES

HP2334A is the only PAD supported at this time, and results in an *opx25* execution of **HP2334A.clr**.

EXAMPLES

A typical invocation is:

```
/usr/lib/uucp/X25/clrsvc /dev/x25.1 HP2334A
```

AUTHOR

Clrsvc was developed by HP.

SEE ALSO

getx25(1M), opx25(1M), getty(1M), login(1), uucp(1).

NAME

config - configure an HP-UX system

SYNOPSIS

/etc/config [-t] [-m master] [-c file] [- l file] [-a file] dfile

DESCRIPTION

Config enables the user to configure the following parts of the operating system:

1. device switch drivers and I/O cards
2. root and swap devices
3. selected system parameters
4. kernel code that handles messages, semaphores, and shared memory

It takes as input, a user-provided description of an HP-UX system (*dfile*) and always generates two files, with an optional third file. The first file is a C program that defines the configuration tables for the various devices on the system. The second file is a *makefile* script that will compile the C program produced and relink the newly configured system. The third file (if specified) contains a *mknod* command for each device specified in *dfile*.

The options available:

-t gives a short table of major device numbers for the character and block devices named in *dfile*. This can facilitate the creation of special files.

-m *master* specifies that the file *master* contains all the information regarding supported devices. The default file name is */etc/master*. This file is supplied with the HP-UX system and **should not be modified unless the user fully understands its construction**.

-c *file* specifies the name of the configuration table file produced by running the user-data file, *dfile*, through *config*(1M). The default file name is *conf.c*

-l *file* specifies the name of the *makefile* script that will compile the configuration program and relink the newly configured system. The default file name is *config.mk*

-a *file* serves two functions:

1. When specified without *dfile*, a *mkdev* script of templates is produced.
2. If *dfile* is given, this indicates that the user will supply addresses for devices so that *config* can produce a script that contains both the *mkdev* templates and a list of *mknod* commands for each device specified in *dfile*. If this option is chosen, all devices must have addresses. Zero (0) as a dummy address, is valid and necessary for many of the devices, e.g., the card drivers. The default file name is *mkdev*.

The only required argument is either *dfile* or **-a**. If *dfile* is given it must contain device information for the user's system. This file is divided into two parts. The first part contains physical device and driver specifications; the second part contains system-dependent information. Any line with an asterisk (*) in column 1 is a comment.

The following devices are **not** configurable and should not be specified in the system descrition file, *dfile*:

swap	cons
tty	sy
mm	ite200
iomap	graphics
r8042	hil
nimitz	

Part 1 of dfile:

This part of *dfile* allows you to configure:

1. device switch drivers
2. I/O cards
3. pseudo-drivers, e.g., *ieee802*, *pty*

Each line contains 1, 2, or 3 fields, delimited by blanks and/or tabs in the following format:

devname [*address*] [*specialfilename*]

where:

devname is the driver name for the device (e.g., cs80 for the HP7912 64MB disc drive) or card (98629 for the SRM card) or the name of the pseudo-driver (e.g., iee802 for the ieee802 protocol) you wish to configure.

address is the minor number for that device as given to mknod or the select code of the card if addressing checking is desired. For pseudo-drivers, i.e., iee802, pty, ethernet, the address field is 0. (in hexadecimal, without the preceding 0x).

specialfilename
is what you want the device's special file to be called in the afile.

For example, to specify a 7914 disc at select code 14, bus address 0 with mknod name */dev/hd*:

```
cs80 0E0000 hd
```

The complete list of configurable devices, cards, and pseudo-drivers is given in the **EXAMPLE** section.

It is not necessary to specify the *address* field, but if you do specify this field and use the **-a** option, *config* will produce a file containing a *mknod* command for each device you specify. It will also check for the unique use of addresses. The **-a** option allows you to name this file.

Part 2 of dfile:

The second part contains four different types of lines; none of these specifications are required.

1. Root device specification lines which have the following form:

root *devname address*

where *devname* is the product number (without the suffix) of the device you wish to configure, e.g., cs80 for the HP7912 64MB disc drive, and *address* is the minor device number (in hexadecimal, without the preceding 0x).

2. Swap device specification lines:/fR

If you want the system to auto-configure the swap device but you want to specify the swap size, then use:

swapsize <*#blocks*>

If you want to specify both the swap device location and its size then the specification line has the following form:

swap *devname address swplo* [*nswap*]

where:

devname	is the product number (with the suffix) of the device you wish to configure, e.g., cs80 for the HP7912 64MB disc drive (in hexadecimal).
address	is the minor device number (in hexadecimal)
swplo	is the location (decimal) of the swap area
nswap	is the number of disc blocks (decimal) in the swap area. Only the *nswap* parameter is optional. Zero is the default for auto-configuration.

swplo:

A negative value (typically -1) for *swplo* specifies that a file system is expected on the device. At boot-up, the super block will be read to determine the exact size of the file system, and this value will be put in *swplo*. If the swap device is auto-configured, this is the mechanism used. If the super block doesn't appear valid, the entry will be skipped, so that the case of a corrupted super block won't later cause the entire file system to be corrupted by configuring the swap area on top of it.

nswap:

If *nswap* is zero, the entire remainder of the device is automatically configured in as swap area.

If *nswap* is non-zero, it's absolute value is treated as an upper bound for the size of the swap area. Then, for the case that the swap area size has actually been cut back, the sign of *nswap* determines whether *swplo* remains as is, resulting in the swap area being adjacent to the reserved area, or whether *swplo* is bumped by the size of the unused area, resulting in the swap area being adjacent to the tail of the device.

3. Parameter specification

These parameters should not be modified unless the user fully understands the ramifications of doing so. See the *System Administrator's Manual* for more detail on each parameter.

The format: lines of two fields each (number is decimal). Each line is independent and optional.

System Parameters:

maxusers	number or formula
timezone	number or formula
dst	number or formula
procs	number or formula
inodes	number or formula
files	number or formula
nbuf	number or formula
ncallout	number or formula
texts	number or formula
unlockable_mem	number or formula
nflocks	number or formula
npty	number or formula
maxuprc	number or formula
dmmin	number or formula
dmmax	number or formula
dmtext	number or formula
dmshm	number or formula
maxdsiz	number or formula
maxssiz	number or formula
maxtsiz	number or formula
shmmaxaddr	number or formula

System V code: messages (*mesg*), semaphores (*sema*) and shared memory (*shmem*) capability

If *mesg*, *sema*, *shmem*= 1, the kernel code for these features will be included (default); if they = 0, the kernel code will not be included: they are independent. If they are included any of the parameters listed below may be modified.

mesg	1
msgmap	number or formula
msgmax	number or formula
msgmnb	number or formula
msgmni	number or formula
msgssz	number or formula
msgtql	number or formula
msgseg	number or formula
sema	1
semmap	number or formula
semmni	number or formula
semmns	number or formula
semmnu	number or formula
semmsl	number or formula
semvmx	number or formula
semaem	number or formula
shmem	1
shmmax	number or formula
shmmin	number or formula

shmmni	number or formula
shmseg	number or formula
shmbrk	number or formula

EXAMPLE

The *dfile* below will configure an HP-UX system with all the drivers that are currently supported on the Series 200 Release 4.0. The tunable parameters given are the system defaults.

```
* drivers
cs80
flex
amigo
tape
printer
stape
srm
ptymas
ptyslv
ieee802
ethernet
hpib
gpio
ciper
* cards
98624
98625
98626
98628
98642
* reconfigure the swap area to occupy an entire CS/80 drive at
* select code 14 bus address 01
swap                  cs80    0E0100   0      0
* tunable parameters
maxusers              8
timezone              420
dst                   1
procs                 (20+8*MAXUSERS)
inodes                ((NPROC+16+MAXUSERS)+32)
files                 (16*(NPROC+16+MAXUSERS)/10+32+2*NETSLOP)
nbuf                  0       /* configure based on memory */
ncallout              (16+NPROC)
texts                 (24+MAXUSERS+NETSLOP)
unlockable_mem        50
nflocks               200
npty                  96
maxuprc               25
dmmin                 16
dmmax                 2048
dmtext                1365
dmshm                 512
maxdsiz               0x01000000
maxssiz               0x01000000
maxtsiz               0x01000000
shmmaxaddr            0x00ffffff
```

```
* configure in messages, semas, and shared memory
mesg                    1
msgmap                  (msgtql + 3)
msgmax                  8192
msgmnb                  16384
msgmni                  50
msgssz                  1
msgtql                  40
msgseg                  16384
sema                    1
semmap                  10
semmni                  64
semmns                  128
semmnu                  30
semmsl                  25
semvmx                  32767
semaem                  16384
shmem                   1
shmmax                  (2048*1024)
shmmin                  1
shmmni                  100
shmseg                  10
shmbrk                  16
```

FILES

/etc/master	default input master device table
conf.c	default output configuration table
config.mk	default makefile script
mkdev	default mknod script

SEE ALSO

master(4)

NAME

cpset – install object files in binary directories

SYNOPSIS

cpset [**-o**] object directory [mode [owner [group]]]

DESCRIPTION

Cpset is used to install the specified *object* file in the given *directory*. The *mode*, *owner*, and *group*, of the destination file may be specified on the command line. If this data is omitted, two results are possible:

If the user of *cpset* has administrative permissions (that is, the user's numerical ID is less than 100), the following defaults are provided:

mode – 0755
owner – bin
group – bin

If the user is not an super-user, the default mode, owner, and group of the destination file will be that of the invoker.

An optional argument of **–o** will force *cpset* to move *object* to **OLD***object* in the destination directory before installing the new object.

For example:

```
cpset echo /bin 0755 bin bin

cpset echo /bin

cpset echo /bin/echo
```

All the examples above have the same effect (assuming the user is an administrator). The file **echo** will be copied into **/bin** and will be given **0755, bin, bin** as the mode, owner, and group, respectively.

Cpset utilizes the file **/usr/src/destinations** to determine the final destination of a file. The locations file contains pairs of pathnames separated by spaces or tabs. The first name is the "official" destination (for example: **/bin/echo**). The second name is the new destination. For example, if *echo* is moved from **/bin** to **/usr/bin**, the entry in **/usr/src/destinations** would be:

```
/bin/echo        /usr/bin/echo
```

When the actual installation happens, *cpset* verifies that the "old" pathname does not exist. If a file exists at that location, *cpset* issues a warning and continues. This file does not exist on a distribution tape; it is used by sites to track local command movement. The procedures used to build the source will be responsible for defining the "official" locations of the source.

Cross Generation

The environment variable **ROOT** will be used to locate the destination file (in the form **$ROOT/usr/src/destinations**). This is necessary in the cases where cross generation is being done on a production system.

SEE ALSO

install(1M), make(1).

NAME

cron – clock daemon

SYNOPSIS

/etc/cron

DESCRIPTION

Cron executes commands at specified dates and times. Regularly scheduled commands can be specified according to instructions found in crontab files; users can submit their own crontab file via the *crontab* command. Commands which are to be executed only once may be submitted via the *at* command. Since *cron* never exits, it should only be executed once. This is best done by running *cron* from the initialization process through the file **/etc/rc** (see *init*(1M)).

Cron only examines crontab files and at(1) command files during process initialization and when a file changes. This reduces the overhead of checking for new or changed files at regularly scheduled intervals.

FILES

/usr/lib/cron	main cron directory
/usr/spool/cron	spool area
/usr/lib/cron/log	accounting information

SEE ALSO

at(1), crontab(1), init(1M), sh(1).

DIAGNOSTICS

A history of all actions taken by cron are recorded in **/usr/lib/cron/log.**

NAME

decode – read and decode diagnostic events from the error log

SYNOPSIS

decode [-**L** logfile] [-**d** device file] [-**e** physical path]
[-**m** major number] [-**n** driver name] [-**p** port] [-**t** [number]]
[-**w**]

DESCRIPTION

Decode reads from stdin a raw diagnostic event message. The message is then decoded into a human-readable format and written to stdout. *Decode* continues reading event messages until EOF is detected.

Decode accepts the following list of parameters:

-L *logfile*	set the error log path name. (Default is stdin.)
-**d** *device file*	set the path for a special device file.
-**e** *physical path*	set the hardware path. (Uses the **1.2.3** notation.)
-**m** *major number*	set the major number.
-**n** *driver name*	set the driver name.
-**p** *port*	set the port number.
-**t** *number*	set the tail parameter. See below for a description of *number.* (Default is last 10 events.)
-**w**	causes *decode* to wait after printing each event.

When specifying a *logfile* to read with the **-L**, the full file path name must be used. Events are then read from this file until EOF is detected.

The options **-d, -e, -m, -n**, and **-p** are used to alter the information that *decode* prints. The options may be used singularly or in combination.

The **-t** option has a numeric parameter, *number* . When *number* is positive it shows the last *number* events in the log file. When *number* is negative the first *number* events in the log file are skipped over.

After *decode* decodes an event it is written to stdout in the following format:

```
*****************************************************
Diagnostic Event number X,   Date = DATE & TIME
*****************************************************

Port number of originating event = aa

Manager of originating event = bb

Physical path = a.b.c

Diagnostic event status:
          error       - cc
          proc num    - dd
          subsystem   - ee
```

```
Diagnostic class = ff

Number of retries = gg

Hardware status = (status bytes in HEX formatted
                   string)

Mgr info = (Manager specific information in HEX
            formatted string)
```

Where:

X	the diagnostic event number relative to the start of the log file.
DATE & TIME	the date and time that *diag0*(7) processed the event.
aa	port number of the I/O subsystem entity that reported the event.
bb	manager name for reporting port number, derived from the I/O tree.
a.b.c	physical path as derived from the I/O tree and the manager specific information.
cc	is an ASCII string corresponding to the LLIO status.
dd and **ee**	are unique numbers used to identify the I/O subsystem entity.
ff	is replaced with either DIAG_HW_EVENT or DIAG_SW_EVENT.
gg	the number of retries made by the reporting entity before sending the event.

RETURNS

If the log file cannot be opened, *decode* will exit with a non-zero status.

WARNINGS

Decode does not decode the manager specific data that is passed back in each diagnostic event message. It can only output the information in a hex format.

AUTHOR

decode was developed by HP.

FILES

/etc/rc
/dev/diag0
/usr/adm/diaglog

SEE ALSO

delog(1M), diag0(7).

NAME

delog – diagnostic event logger for I/O subsystem.

SYNOPSIS

delog [[**-e** [number]] | [**-w**]] [**-b**] [**-d** dev file]
[**-f** outfile] [**-L** error log] [**-U** vmunix] [**-V** version]

DESCRIPTION

In general, *delog* only reads in an event message from the diagnostic driver and then writes it to a log file, with no information being sent to the system console. The event message can then by decoded into human-readable format at some later time by *decode.* By using various options *delog* can write a synopsis of the messages to the system console and/or to a designated file. These options, as described below, allow the user to tune the output of *delog* to the needs of the system.

Delog accepts the following list of parameters:

-e *number*	print out a brief synopsis for only fatal diagnostic events. See below for a description of *number. Number* can be **1** or **2**.
-w	print out a brief synopsis of the diagnostic event.
-b	send a synopsis to both the system console and a user defined file.
-d *dev file*	set the device file path name (defaults to */dev/diag0*).
-f *outfile*	set the output file path name (defaults to */dev/null*).
-L *error log*	set the error log path name (defaults to */usr/adm/diaglog*).
-U *vmunix*	set the *vmunix* path name (defaults to */hp-ux*).
-V *version*	set the version number of the *vmunix* file (default is to be determined).

Messages sent either to the system console or a user defined file take on the following format:

```
Diagnostic class type from XX (at a.b.c) status = x
Hw stat = (hardware status printout as a HEX string)
```

Where:

class	is replaced with either hardware or software.
type	is replaced with either error or warning.
XX	is the name of the driver that reported the event.
a.b.c	is the hardware path derived from the I/O tree.
x	is an ASCII string corresponding to the LLIO status.

The **-w** option prints out, on the system console, warning and error information. Warnings are informational and are not usually significant. Errors may be informational or they may be reports of fatal errors. Therefore, a large amount of data, mostly informational, will go to the system console.

The **-e** option causes *delog* to print messages that are only I/O subsystem errors. However, because of the nature of I/O subsystem messages, not all superfluous informational messages can be suppressed. To further reduce the amount of data written to the system console, the **-e** option has a numeric parameter, *number.* The default value of *number* is **1** if nothing is specified. When *number* is **2**, the hardware status is suppressed. The format of the output for the **-e** option is the same as that of the **-w** option. (Note: the **-e** and **-w** options are mutually exclusive.)

The synopsis message can be redirected to a file using the **-f** option. In combination with the **-b** option the messages can be sent to the system console and the designated file. If the **-b** option is used by itself, the result is the same as if the **-w** option were specified.

Because the read function of the diagnostic driver is exclusive, only one *delog* logging process can be run at a time.

RETURNS

If the diagnostic driver cannot be opened, *delog* will exit with a non-zero status. If the log file cannot be opened, *delog* will not complain, instead *delog* prints a synopsis of the event on the system console (and the user defined file, if specified) and continues.

WARNINGS

Delog does not do any error log manipulation. If the file used for error logging becomes too large, it is up to a system administrator to clear it out.

AUTHOR

Delog was developed by HP.

FILES

/etc/rc
/dev/diag0
/usr/adm/diaglog
/hp-ux

SEE ALSO

decode(1M), diag0(7).

NAME

devnm – device name

SYNOPSIS

/etc/devnm [names]

DESCRIPTION

Devnm identifies the special file associated with the mounted file system where the argument *name* resides. (As a special case, both the block device name and the swap device name are printed for the argument name / if swapping is done on the same disk section as the **root** file system.) Argument names must be full path names.

This command is most commonly used by **/etc/rc** (see *brc*(1M)) to construct a mount table entry for the **root** device.

EXAMPLE

The command:

```
/etc/devnm /usr
```

produces

```
dsk/0s1 /usr
```

if **/usr** is mounted on **/dev/dsk/0s1**.

FILES

/dev/dsk/*
/etc/mnttab

SEE ALSO

brc(1M), setmnt(1M).

NAME

df – report number of free disk blocks

SYNOPSIS

df [**–t**] [**–f**] [file-systems]

DESCRIPTION

Df prints out the number of free 512-byte blocks and free inodes available for online file systems by examining the counts kept in the super-block(s). *File-systems* may be specified either by device name (e.g., **/dev/dsk/0s1**) or by mounted directory name (e.g., **/usr**). If the *file-systems* argument is unspecified, the free space on all of the mounted file systems is printed.

The **–t** flag causes the total allocated block figures to be reported as well.

If the **–f** flag is given, only an actual count of the blocks in the free list is made (free inodes are not reported). With this option, *df* will report on raw devices.

When *df* is used on an HFS file system, the file space reported is the space available to the ordinary user, and does not include the reserved file space specified by *fs_minfree.* Unreported reserved blocks are available only to super-user. See *fs*[HFS](4) for information about *fs_minfree.*

FILES

/dev/dsk/*
/etc/mnttab

SEE ALSO

du(1), fsck(1M), fs(4), mnttab(4).

NAME

disksecn – calculate default disc section sizes

SYNOPSIS

disksecn [–**p** | –**d**] [–**b** block size] [–**n** disc name]

DESCRIPTION

Disksecn is used to calculate the disc section sizes based on the Berkeley disc partitioning method.

–**p**	Tables suitable for inclusion in the device driver are output.
–**d**	Tables suitable for generating the disc description file **/etc/disktab** are output.
–**b** *block size*	Defines a block size in bytes to be used as the sector size in generating the above tables. Legal values for *blocksize* are **256**, **512**, **1024** and **2048**. Defaults to DEV_BSIZE if not specified.
–**n** *disc name*	Specifies the disc name to be used in calculating sector sizes. For example, hp7912 or hp7945. If an unknown disc name is specified, *disksecn* will prompt the user for the necessary disc information.

If neither –**p** nor –**d** table selection switches are specified a default table of the section sizes and range of cylinders used is output.

The disc section sizes are based on the total amount of space on the disc as given in the table below (all values are supplied in units of 256-byte sectors). If the disc is smaller than approximately 44 MB, *disksecn* aborts and returns the message "**disk too small, calculate by hand**".

Section	44-56MB	57-106MB	107-332MB	333+MB
0	97120	97120	97120	97120
1	39064	39064	143808	194240
3	39064	39064	78128	117192
4	unused	48560	110096	429704
6	7992	7992	7992	7992
10	unused	unused	unused	516096

NOTE:
It is important to note the difference between the block size passed into *disksecn* via the –**b** switch argument and the sector size the user is asked to input when an unknown disc name is passed to *disksecn* via the –**n** switch argument.

The block size is the sector size that *disksecn* assumes the disc to have when it prints out the requested tables. All information printed out in the tables is adjusted to reflect this assumed sector size (block size) passed in by the user. The sector size requested by *disksecn* when an unknown disc name is passed does not necessarily have to be the same as the assumed sector size (block size) passed in by the **-b** switch argument.

For example, a user wishes to see the device driver tables output for the hp7945 with an assumed sector size (block size) of 256 bytes. The user has the following information about the hp7945 disc:

Disc type = winchester

Sector size = 512

Number of sectors per track (512 byte sectors) = 16

Number of tracks = 7

Number of cylinders = 968

Revolutions per minute = 3600

The user invokes *disksecn* by typing the following command:

```
disksecn -p -b 256 -n hp7945
```

Assuming that the hp7945 is an unknown disc name, *disksecn* will prompt the user for the necessary disc information. The user should input the information as shown above, reflecting a sector size of 512 bytes. All the information will be adjusted within *disksecn* to reflect the assumed sector size (block size) of 256 bytes, passed as the argument of the **-b** switch, before the requested device driver table is output.

This adjustment also takes place when the disc name is known and an assumed sector size (block size) is passed in as the argument of the **-b** switch which is not DEV_BSIZE bytes, the assumed sector size (block size) used to create the **etc/disktab** file.

RETURNS

Disksecn returns **0** for a successful completion, **1** for a usage error, **2** for a user not wanted to input parameters for an unknown disc and **3** for a disc that is too small or an invalid block size.

Disksecn will abort and print out an error message under the following conditions:

Disksecn is invoked without specifying a disc name.

Both a **-p** and a **-d** switch were requested.

A block size other than those specified as legal is requested.

An unknown disc name is specified and user does not wish to supply disc information.

A disc whose maximum storage space is less than approximately 44 MB.

WARNINGS

When using the **-d** switch alternate names are not included in the output

There must be spaces typed between each of the switches in the command line when invoking *disksecn*.

There must be a space between the **-n** switch and the disc name argument to that switch. For example:

```
disksecn -p -b 1024 -n hp9712
```

At this point in time the program has no way to save the block size that was used to generate the **/etc/disktab** disc description file. The system assumes that the block size used was DEV_BSIZE when it reads the information stored in the **etc/disktab** file.

AUTHOR

Disksecn was developed by the University of California, Berkeley.

FILES

/etc/disktab

SEE ALSO

disktab(4).

NAME

diskusg - generate disk accounting data by user ID

SYNOPSIS

diskusg [options] [files]

DESCRIPTION

Diskusg generates intermediate disk accounting information from data in *files,* or the standard input if omitted. *Diskusg* output lines on the standard output, one per user, in the following format:

uid login #blocks

where

uid -	the numerical user ID of the user;
login -	the login name of the user; and
#blocks -	the total number of disk blocks allocated to this user.

Diskusg normally reads only the inodes of file systems for disk accounting. In this case, *files* are the special filenames of these devices.

Diskusg recognizes the following options:

-s the input data is already in *diskusg* output format. *Diskusg* combines all lines for a single user into a single line.

-v verbose. Print a list on standard error of all files that are charged to no one.

-i *fnmlist*
ignore the data on those file systems whose file system name is in *fnmlist. Fnmlist* is a list of file system names separated by commas or enclosed within quotes. *Diskusg* compares each name in this list with the file system name stored in the volume ID (see *labelit* on *volcopy*(1)).

-p *file*
use *file* as the name of the password file to generate login names. **/etc/passwd** is used by default.

-u *file*
write records to *file* of files that are charged to no one. Records consist of the special file name, the inode number, and the user ID.

The output of *diskusg* is normally the input to *acctdisk* (see *acct*(1M)) which generates total accounting records that can be merged with other accounting records. *Diskusg* is normally run in *dodisk* (see *acctsh*(1M)).

EXAMPLES

The following will generate daily disk accounting information:

```
for i in /dev/rp00 /dev/rp01 /dev/rp10 /dev/rp11; do
        diskusg $i > dtmp.`basename $i` &
done
wait
diskusg -s dtmp.* | sort +0n +1 | acctdisk > disktacct
```

FILES

/etc/passwd used for user ID to login name conversions

SEE ALSO

acct(1M), acctsh(1M), volcopy(1M), acct(4).

NAME

dmesg – collect system diagnostic messages to form error log

SYNOPSIS

/etc/dmesg [–]

DESCRIPTION

Dmesg looks in a system buffer for recently printed diagnostic messages and prints them on the standard output. The messages are those printed by the system when unusual events occur (such as when system tables overflow or the system crashes). If the – flag is given, then *dmesg* computes (incrementally) the new messages since the last time it was run and places these on the standard output. This is typically used with *cron*(1) to produce the error log */usr/adm/messages* by running the command:

```
/etc/dmesg – >> /usr/adm/messages every 10 minutes.
```

WARNINGS

The system error message buffer is of small finite size. Because *dmesg* is run only every few minutes, not all error messages are guaranteed to be logged.

AUTHOR

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

FILES

/usr/adm/messages	error log (conventional location)
/usr/adm/msgbuf	scratch file for memory of – option

NAME

fsck – file system consistency check and interactive repair

SYNOPSIS

/etc/fsck **–p** [file system ...]
/etc/fsck **–P** [file system ...]
/etc/fsck [**–b** block#][**–y**][**–n**][**–q**][file system ...]

MARKETING MODEL

Level C

TECHNICAL MODEL

Large Machine
HP+ (HFS Subset)

DESCRIPTION

Fsck audits and interactively repairs inconsistent conditions for HP-UX file systems. If the file system is consistent, the number of files on that file system and the number of used and free blocks are reported. If the file system is inconsistent, *fsck* provides a mechanism to fix these inconsistencies depending on which form of the *fsck* command used.

Fsck checks a default set of file systems or the file systems specified in the command line. If *file system* is not specified, *fsck* reads the table in **/etc/checklist**, using the first field (special file name) to determine which file systems to check.

If the **-p** option is used without specifying a *file system, fsck* reads the specified pass numbers in **/etc/checklist** to inspect groups of disks in parallel, taking maximum advantage of I/O overlap to preen the file systems as quickly as possible. The **-p** option is normally used in the script **/etc/bcheckrc** during automatic reboot. Normally, the root file system will be checked on pass 1, and other "root" ("0" section) file systems on pass 2. Other small file systems are checked on separate passes (e.g. the "section 4" file systems on pass 3 and the "section 7" file systems on pass 4), and finally the large user file systems are checked on the last pass (e.g. pass 5). A pass number of zero or a type which is neither "rw" nor "ro" in **/etc/checklist** causes a file system not to be checked. If the optional fields are not present on a line in **/etc/checklist**, or the pass number is -1, *fsck* will preen the file system on such lines sequentially after all eligible file systems with positive pass numbers have been preened.

Below are the inconsistencies that *fsck* with the **–p** option will correct; if it encounters other inconsistencies it exits with an abnormal return status. For each corrected inconsistency, one or more lines will be printed identifying the file system on which the correction will take place, and the nature of the correction. The inconsistencies that are corrected are limited to the following:

Unreferenced inodes

Unreferenced pipes and fifos

Link counts in inodes too large

Missing blocks in the free list

Blocks in the free list also in files

Counts in the super-block wrong.

The **–P** option operates in the same manner as the **–p** option except those file systems which were cleanly unmounted will not be checked (see *fsclean*(1M)). This can greatly decrease the amount of time required to reboot a system which was brought down cleanly.

Without the **-p** or **-P** option, *fsck* prompts for concurrence before each correction is attempted when the file system is inconsistent. It should be noted that some corrective actions will result in a loss of data. The amount and severity of data lost may be determined from the diagnostic output. The default action for each consistency correction is to wait for the operator to respond **yes** or **no.** If the operator does not have write permission, *fsck* will default to a **−n** action. The following options are interpreted by *fsck*.

−b Use the block specified immediately after the flag as the super block for the file system. An alternate super block will always be found at block ((SBSIZE + BBSIZE)/DEV_BSIZE), typically block 16.

−y Assume a **yes** response to all questions asked by *fsck;* this should be used with great caution as this is a free license to continue after essentially unlimited trouble has been encountered.

−n Assume a **no** response to all questions asked by *fsck;* do not open the file system for writing.

−q Quiet *fsck*. Do not print size-check messages in Phase 1. Unreferenced fifos will silently be removed. If *fsck* requires it, counts in the superblock and cylinder groups will be automatically fixed.

In all cases the inconsistencies checked by *fsck* are as follows:

1. Blocks claimed by more than one inode or the free list.
2. Blocks claimed by an inode or the free list outside the range of the file system.
3. Incorrect link counts.
4. Size checks:
   Directory size not of proper format.
5. Bad inode format.
6. Blocks not accounted for anywhere.
7. Directory checks:
   File pointing to unallocated inode.
   Inode number out of range.
8. Super Block checks:
   More blocks for inodes than there are in the file system.
9. Bad free block list format.
10. Total free block and/or free inode count incorrect.

Orphaned files and directories (allocated but unreferenced) are, with the operator's concurrence, reconnected by placing them in the **lost+found** directory. The name assigned is the inode number. The only restriction is that the directory **lost+found** must preexist in the root of the file system being checked and must have empty slots in which entries can be made. This is accomplished by making **lost+found**, copying a number of files to the directory, and then removing them (before *fsck* is executed).

After *fsck* has checked and fixed the file system, it will store the correct fs_clean flag in the super block if it is not already there. For a non-root file system, FS_CLEAN will be stored there. For the root file system, which is mounted at the time of the *fsck*, no changes are required to the super block if there were no problems found and if FS_OK was already set.

Checking the raw device is almost always faster.

WARNINGS

Fsck should not be run on mounted file systems or on the raw root device.

HARDWARE DEPENDENCIES

Series 200, 300

The 5.0 release supports only one section per volume.

AUTHOR

Fsck[HFS] was developed by HP, AT&T, and the University of California, Berkeley.

FILES

/etc/checklist	contains default list of file systems to check.

SEE ALSO

fsclean(1M), mkfs[HFS](1M), newfs[HFS](1M), checklist(4), fs[HFS](4).

NAME

fsck – file system consistency check and interactive repair

SYNOPSIS

/etc/fsck [–y] [–**n**] [–**s**] [–**d**] [*file system*]

DESCRIPTION

Fsck checks for and interactively repairs inconsistent conditions in SDF file systems. If the file system is consistent, *fsck* reports the number of files, the number of blocks used, the number of blocks free, and the percent of volume unused. If the file system is inconsistent, *fsck* prompts the operator for concurrence before each operation is attempted. Note that certain types of corrective actions result in some data loss. The amount and severity of the loss can be determined from the diagnostic output. The default action for each inconsistency correction is to wait for the operator to respond **yes** or **no**. If the operator does not have write permission, *fsck* will default to a –**n** action.

Fsck makes multiple passes over the file system, so care should be taken to ensure that the system is quiescent. You should unmount the file system being checked, if possible. At the least, the system should be single-user, and spurious processes (such as *cron*) should be killed.

The following flags are interpreted by *fsck*:

–y Assume a **yes** response to all questions asked.

–n Assume a **no** response to all questions asked; do not open the file system for writing.

–s Ignore the actual free list and unconditionally reconstruct a new one. This option is useful in correcting multiply claimed blocks when one of the claimants is the free list. When using this option, the number of unclaimed blocks reported by *fsck* includes all the blocks in the free map. This can produce extensive output if –**d** is also selected.

–**s** should only be selected after a previous *fsck* indicates a conflict between a file and the free map. After *fsck –s* has executed, the integrity of the conflicting file(s) should be checked.

If –**s** is used to correct a problem on a virtual memory device, there is a high probability that the final step in *fsck* will fail, and you will be forced to reboot. Should this occur, an appropriate error message will be printed. No damage should occur.

–d Dump additional information. The more **d**'s that are present, the more information that is dumped. You may specify up to five **d**'s. However, using more than two can result in an overwhelming amount of output.

Fsck also recognizes, and ignores, the –**S** and –**t** options found in other versions of *fsck*. An appropriate warning is printed.

File system is a device file name describing the device on which the file system to be checked resides. If no *file system*(s) are specified, *fsck* will read a list of default file systems from the file **/etc/checklist**.

Error messages from *fsck* are written to *stderr*. Information generated because of the –**d** option and normal output is written to *stdout*. Both are unbuffered.

Inconsistencies checked include:

1. Blocks claimed by more than one inode, or by the free list;
2. Blocks claimed by an inode or the free list outside the range of the file system;
3. Incorrect link counts;
4. Blocks not accounted for anywhere;
5. Bad inode format;
6. Directory checks:
   Files pointing to unallocated inodes,
   Inode numbers out of range,
   Multiply linked directories,
   Link to the parent directory.

Orphaned files (allocated but unreferenced) with non-zero sizes are, with the operator's concurrence, reconnected by placing them in the *lost+found* directory. The name assigned is the inode number. The only restriction is that *lost+found* must exist in the root of the file system being checked, and must have empty slots in which entries can be made. This is accomplished by creating *lost+found*, copying a number of files to it, and then removing them (before *fsck* is executed).

Orphaned directories and files with zero size, with the operator's concurrence, are returned directly to the free list. This will also happen if the *lost+found* directory does not exist.

You should run a backup prior to running *fsck* for repairs.

DIAGNOSTICS

The diagnostics are intended to be self-explanatory.

WARNINGS

All SDF file systems being checked must be described by a character special device file.

Do not redirect *stdout* or *stderr* to a file on the device being checked. This includes pipes when checking the root volume.

Fsck cannot check devices with a logical block size greater than 4096.

HARDWARE DEPENDENCIES

SDF file systems are implemented only on Series 500 computers.

FILES

/etc/checklist	contains the default list of file systems to check

SEE ALSO

checklist(4), fs(4).

Series 500 HP-UX System Administrator Manual.

NAME

fsclean – determine shutdown status of specified file system

SYNOPSIS

/etc/fsclean [-v] [special ...]

DESCRIPTION

Fsclean determines the shutdown status of the the file system specified by *special* or, in the absence of *special*, the file systems listed in **/etc/checklist** of type "rw" or "ro". All optional fields of *checklist* must be present for *fsclean* to be able to check each file system.

Fsclean reads the super block to determine if the file system's last shutdown was done correctly. If all of the checked file systems were shutdown correctly, *fsclean* returns **0**. If any were not, *fsclean* returns **1**. Any other errors, such as "cannot open the specified device file," return **2**. *Fsclean* is normally silent.

The only option is:

−v Be verbose. Prints the fs_clean value of each file system checked.

AUTHOR

Fsclean was developed by HP.

FILES

/etc/checklist

SEE ALSO

checklist(4), brc(1M), reboot(1M).

NAME

fsdb – file system debugger

SYNOPSIS

/etc/fsdb special [**–b block#**] [–]

REMARKS

Always execute fsck(1M) after done with fsdb.

DESCRIPTION

Fsdb can be used to patch up a damaged file system after a crash. It normally uses the first super block for the file system located at the beginning of the disk section as the effective super block. If the *-b* flag is used, the block specified immediately after the flag will be used as the super block for the file system. An alternate super block will always be found at block ((SBSIZE + BBSIZE)/DEV_BSIZE), typically block 16.

Fsdb deals with the file system in terms of block fragments, which are the unit of addressing in the file system and the minimum unit of space allocation. To avoid possible confusion, *fragment* is used to mean that, and *block* is reserved for the larger true block. *Fsdb* has conversions to translate fragment numbers and i-numbers into their corresponding disk addresses. Also included are mnemonic offsets to access different parts of an inode. These greatly simplify the process of correcting control block entries or descending the file system tree.

Fsdb contains several error-checking routines to verify inode and fragment addresses. These can be disabled if necessary by invoking *fsdb* with the optional – argument or by the use of the **O** symbol.

Numbers are considered decimal by default. Octal numbers must be prefixed with a zero. Hexadecimal numbers must be prefixed with **0x**. During any assignment operation, numbers are checked for a possible truncation error due to a size mismatch between source and destination.

Fsdb reads a fragment at a time. A buffer management routine is used to retain commonly used fragment of data in order to reduce the number of read system calls. All assignment operations result in an immediate write-through of the corresponding fragment.

The symbols recognized by *fsdb* are:

#	absolute address
i	convert from i-number to inode address
b	convert from fragment number to disk address (historically "block")
d	directory slot offset
+,–	address arithmetic
q	quit
>,<	save, restore an address
=	numerical assignment
=+	incremental assignment
=–	decremental assignment
="	character string assignment
X	hexadecimal flip flop
O	error checking flip flop
p	general print facilities
f	file print facility
B	byte mode
W	word mode
D	double word mode
!	escape to shell

The print facilities generate a formatted output in various styles. Octal numbers are prefixed with a zero. Hexadecimal numbers are prefixed with 0x. The current address is normalized to an appropriate boundary before printing begins. It advances with the printing and is left at the address of the last item printed. The output can be terminated at any time by typing the interrupt character. If a number follows the **p** symbol, that many entries are printed. A check is made to detect fragment boundary overflows since logically sequential blocks are generally not physically sequential. If a count of zero is used, all entries to the end of the current fragment are printed. The print options available are:

i	print as inodes
d	print as directories
o	print as octal words
x	print as hexadecimal words
e	print as decimal words
c	print as characters
b	print as octal bytes

The **f** symbol is used to print data fragments associated with the current inode. If followed by a number, that fragment of the file is printed. (Fragments are numbered from zero). The desired print option letter follows the fragment number, if present, or the **f** symbol. This print facility works for small as well as large files except for special files such as fifos, and device special files.

Dots, tabs, and spaces may be used as function delimiters but are not necessary. A line with just a new-line character will increment the current address by the size of the data type last printed. That is, the address is set to the next byte, word, double word, directory entry or inode, allowing the user to step through a region of a file system. Information is printed in a format appropriate to the data type. Bytes, words and double words are displayed with the octal (hexadecimal if X toggle is used) address followed by the value in octal (hexadecimal if X toggle is used) and decimal. A **.B** or **.D** is appended to the address for byte and double word values, respectively. Directories are printed as a directory slot offset followed by the decimal i-number and the character representation of the entry name. Inodes are printed with labeled fields describing each element.

The following mnemonics are used for inode examination and refer to the current working inode:

md	mode
ln	link count
uid	user ID number
gid	group ID number
sz	file size in byte unit
a#	data block numbers (0 – 14)
at	time last accessed
mt	time last modified
ct	last time inode changed
maj	major device number
min	minor device number

The following mnemonics are used for directory examination:

di	i-number of the associated directory entry
nm	name of the associated directory entry

HARDWARE DEPENDENCIES

The HFS file system is implemented on Series 200 beginning with HP-UX Release 5.0, and on Series 300 and Series 800.

EXAMPLES

386i	prints i-number 386 in an inode format. This now becomes the current working inode.
ln=4	changes the link count for the working inode to 4.

ln=+1	increments the link count by 1.
fc	prints, in ASCII, fragment zero of the file associated with the working inode.
2i.fd	prints the first fragment-size piece of directory entries for the root inode of this file system.
d5i.fc	changes the current inode to that associated with the 5th directory entry (numbered from zero) found from the above command. The first fragment's worth of bytes of the file are then printed in ASCII.
1b.px	prints the first fragment of the superblock of this file system in hexadecimal.
2i.a0b.d7=3	changes the i-number for the seventh directory slot in the root directory to 3. This example also shows how several operations can be combined on one command line.
d7.nm="name"	changes the name field in the directory slot to the given string. Quotes are optional if the first character of the name field is alphabetic.
a2b.p0d	prints the third fragment of the current inode as directory entries.

AUTHOR

Fsdb[HFS] was developed by the Hewlett-Packard Company and AT&T Bell Laboratories.

SEE ALSO

fsck[HFS](1M), dir(4), fs[HFS](4).

WARNING

The use of *fsdb* should be limited to experienced *fsdb* users.

NAME
fsdb – examine/modify file system

SYNOPSIS
fsdb file-system

DESCRIPTION
Fsdb provides you with the ability to perform the following functions for each specified SDF *file-system*:

1. Find the inode number of a file, given its full path name. The *file-system* must be the root file system, or must be mountable to use this feature.
2. Examine and modify the contents of the superblock (volume header).
3. Examine and modify the contents of any inode or other file attribute file record.

Integer input to *fsdb* may be entered in decimal (default), octal (with a preceding "0"), or hexadecimal (with a preceding "0x").

File-system is a raw or block special file describing the device on which the file system is located.

Fsdb may be executed only by the super-user.

Fsdb execution is largely self-explanatory. Prompts consist of questions requesting the needed information. When execution begins, *fsdb* displays the following menu:

```
1 – find inode numbers.
2 – examine superblock.
3 – examine inodes.
q – quit.
```

after which you are requested to enter one of the options shown. Typing **1** causes *fsdb* to accept full pathnames of files, in return for which it prints the corresponding inode number. Typing **q** returns you to the main menu.

Typing **2** displays the contents of each record in the superblock. Each record is numbered. If a right parenthesis ")" follows the number, then the record can be modified. If a right curly bracket "}" follows the number, then the record cannot be modified. You are then asked whether or not you want to modify the superblock. An answer beginning with **n** sends you back to the menu; an initial **y** causes *fsdb* to ask for the record number to be modified. If the record number specified cannot be modified, you are told about it, and prompted for another record number. If you specify a record number which can be changed, you are prompted for the new data. Typing **q** returns you to the main menu.

Typing **3** causes *fsdb* to prompt you for a file attribute record number. Upon receipt of a valid number, the contents of that record are displayed, and you are prompted for the information you want to change. Parentheses and curly brackets have the same meanings as described above. Typing **q** returns you to the main menu.

Typing **q** at the main menu level terminates the command.

A word of caution: *fsdb* is deceptively easy to use, and therefore should be used with extreme care. Be sure you know what you are doing before you enter too deeply into options **2** or **3**. You are given the opportunity to abort (by typing **q**) any operation before you have changed anything, so consider carefully what you are about to do before you do it. *Fsdb* does not provide an "undo" function – the changes you make are immediate.

SEE ALSO
fsck(1M).

BUGS

If *fsdb* changes a field that is duplicated in an in-memory OS data structure, the change may be undone by the OS. Forcing a reboot while still in *fsdb* sometimes circumnavigates this problem. Changes to inodes 0 and 1 always fall into this category.

NAME

fwtmp, wtmpfix – manipulate connect accounting records

SYNOPSIS

/usr/lib/acct/fwtmp [–**ic**]
/usr/lib/acct/wtmpfix [files]

DESCRIPTION

Fwtmp

Fwtmp reads from the standard input and writes to the standard output, converting binary records of the type found in **wtmp** to formatted ASCII records. The ASCII version is useful to enable editing, via *ed*(1), bad records or general purpose maintenance of the file.

The argument –**ic** is used to denote that input is in ASCII form, and output is to be written in binary form. (The arguments **i** and **c** are independent, respectively specifying ASCII input and binary output, thus –**i** is an ASCII to ASCII copy and –**c** is a binary to binary copy).

Wtmpfix

Wtmpfix examines the standard input or named files in **wtmp** format, corrects the time/date stamps to make the entries consistent, and writes to the standard output. A – can be used in place of *files* to indicate the standard input. If time/date corrections are not performed, *acctcon1* will fault when it encounters certain date-change records.

Each time the date is set, a pair of date change records are written to **/usr/adm/wtmp**. The first record is the old date denoted by the string **old time** placed in the line field and the flag **OLD_TIME** placed in the type field of the **<utmp.h>** structure. The second record specifies the new date and is denoted by the string **new time** placed in the line field and the flag **NEW_TIME** placed in the type field. *Wtmpfix* uses these records to synchronize all time stamps in the file. *Wtmpfix* nullifies date change records when writing to the standard output by setting the time field of the **<utmp.h>** structure in the old date change record equal to the time field in the new date change record. In this way, *wtmpfix* and *acctcon1* will not factor in a date change record pair more than once.

In addition to correcting time/date stamps, *wtmpfix* will check the validity of the name field to ensure that it consists solely of alphanumeric characters or spaces. If it encounters a name that is considered invalid, it will change the login name to **INVALID** and write a diagnostic to the standard error. In this way, *wtmpfix* reduces the chance that *acctcon1* will fail when processing connect accounting records.

FILES

/usr/include/utmp.h
/etc/wtmp

SEE ALSO

acct(1M), acctcms(1M), acctcom(1), acctcon(1M), acctmerg(1M), acctprc(1M), acctsh(1M), ed(1), runacct(1M), acct(2), acct(4), utmp(4).

DIAGNOSTICS

Wtmpfix generates these diagnostics:

Cannot make temporary: xxx failed to make temp file
Input truncated at offset: xxx missing half of date pair
New date expected at offset: xxx missing half of date pair
Cannot read from temp: xxx some error reading
Bad file at offset: xxx ut_line entry not digit, alpha, nor "|" or "{" (first character only checked)
Out of core: *malloc* fails. (Saves table of date changes)
No dtab: software error (not seen yet)

BUGS

Fwtmp generates no errors, even on garbage input.

NAME

getty – set terminal type, modes, speed, and line discipline

SYNOPSIS

/etc/getty [**–h**] [**–t** timeout] line [speed [type [linedisc]]]
/etc/getty –c file

DESCRIPTION

Getty is a program that is invoked by *init*(1M). It is the second process in the series, (*init-getty-login-shell*) that ultimately connects a user with the HP-UX system. Initially, if **/etc/issue** exists, *getty* prints its contents to the user's terminal, followed by the login message field for the entry it is using from **/etc/gettydefs**. *Getty* reads the user's login name and invokes the *login*(1) command with the user's name as argument. While reading the name, *getty* attempts to adapt the system to the speed and type of terminal being used.

Line is the name of a tty line in **/dev** to which *getty* is to attach itself. *Getty* uses this string as the name of a file in the **/dev** directory to open for reading and writing. Unless *getty* is invoked with the **–h** flag, *getty* will force a hangup on the line by setting the speed to zero before setting the speed to the default or specified speed. The **–t** flag plus *timeout* in seconds, specifies that *getty* should exit if the open on the line succeeds and no one types anything in the specified number of seconds. The optional second argument, *speed*, is a label to a speed and tty definition in the file **/etc/gettydefs**. This definition tells *getty* at what speed to initially run, what the login message should look like, what the initial tty settings are, and what speed to try next should the user indicate that the speed is inappropriate (by typing a <*break*> character). The default *speed* is 300 baud. The optional third argument, *type*, is a character string describing to *getty* what type of terminal is connected to the line in question. *Getty* understands the following types:

none	default
vt61	DEC vt61
vt100	DEC vt100
hp45	Hewlett-Packard HP2645
c100	Concept 100

The default terminal is **none**; i.e., any crt or normal terminal unknown to the system. Also, for terminal type to have any meaning, the virtual terminal handlers must be compiled into the operating system. They are available, but not compiled in the default condition. The optional fourth argument, *linedisc*, is a character string describing which line discipline to use in communicating with the terminal. Again the hooks for line disciplines are available in the operating system but there is only one presently available, the default line discipline, **LDISC0**.

When given no optional arguments, *getty* sets the *speed* of the interface to 300 baud, specifies that raw mode is to be used (awaken on every character), that echo is to be suppressed, either parity allowed, new-line characters will be converted to carriage return-line feed, and tab expansion performed on the standard output. It types the login message before reading the user's name a character at a time. If a null character (or framing error) is received, it is assumed to be the result of the user pushing the "break" key. This will cause *getty* to attempt the next *speed* in the series. The series that *getty* tries is determined by what it finds in **/etc/gettydefs**.

The user's name is terminated by a new-line or carriage-return character. The latter results in the system being set to treat carriage returns appropriately (see *ioctl*(2)).

The user's name is scanned to see if it contains any lower-case alphabetic characters; if not, and if the name is non-empty, the system is told to map any future upper-case characters into the corresponding lower-case characters.

In addition to the typical UNIX operating system's erase and kill characters (# and @), *getty* also understands **\b** and **^U**. If the user uses a **\b** as an erase, or **^U** as a kill character, *getty* sets the standard erase character and/or kill character to match.

Getty also understands the "standard" ESS2 protocols for erasing, killing and aborting a line, and terminating a line. If *getty* sees the ESS erase character, _, or kill character, **$**, or abort character, **&**, or the ESS line terminators, **/** or **!**, it arranges for this set of characters to be used for these functions.

Finally, *login* is called with the user's name as an argument. Additional arguments may be typed after the login name. These are passed to *login*, which will place them in the environment (see *login*(1)).

A check option is provided. When *getty* is invoked with the **−c** option and *file*, it scans the file as if it were scanning **/etc/gettydefs** and prints out the results to the standard output. If there are any unrecognized modes or improperly constructed entries, it reports these. If the entries are correct, it prints out the values of the various flags. See *ioctl*(2) to interpret the values. Note that some values are added to the flags automatically.

FILES

/etc/gettydefs
/etc/issue

SEE ALSO

ct(1), init(1M), login(1), ioctl(2), gettydefs(4), inittab(4), termio(7).

BUGS

While *getty* does understand simple single character quoting conventions, it is not possible to quote the special control characters that *getty* uses to determine when the end of the line has been reached, which protocol is being used, and what the erase character is. Therefore it is not possible to login via *getty* and type a #, @, /, !, _, backspace, ^**U**, ^**D**, or **&** as part of your login name or arguments. They will always be interpreted as having their special meaning as described above.

NAME

getx25 – get x25 line

SYNOPSIS

/etc/getx25 line speed pad-type

DESCRIPTION

Getx25 is very similar to getty in function, but is used only for incoming lines that are connected to an X.25 PAD. It performs special functions such as setting up an initial PAD configuration. It also logs the number of the caller in /usr/spool/uucp/X25LOG. The third parameter is the name of the PAD being used. HP2334A is the only one supported at this time. A typical invokation would be:

```
/etc/getx25 x25.1 2 HP2334A
```

SEE ALSO

getty(1M), login(1), uucp(1)

AUTHOR

Getx25 was developed by HP.

NAME

hpux – HP-UX bootstrap and installation utility

SYNOPSIS

hpux [**boot**] [**–f**number] [**–i**string] [**–r**] [devicefile]
hpux copy devicefile devicefile
hpux –v

DESCRIPTION

Hpux is the HP-UX specific initial system loader (*isl*(1M)) utility for bootstrap and first-time installation. Currently it supports two operations: **boot** and **copy**. The **boot** operation loads an object file from an HP-UX file system or raw device and transfers control to the loaded image. The **copy** operation copies data between HP-UX files and/or raw devices and is typically used during first-time installation.

The command sequences for all operations are presented under **SYNOPSIS** above. They may be either entered interactively to *isl* or present in an *isl autoexecute* file. The first sequence performs the **boot** operation. The second performs the **copy** operation. The third returns the release and version numbers of *hpux*.

NUMBERS

Hpux accepts numbers (i.e. numeric constants) in many of its options. Numbers follow the *C* language notation for decimal, octal, and hexadecimal constants. A leading 0 (zero) implies octal and a leading 0x or 0X implies hexadecimal. For example, 037, 0x1F, 0X1f, and 31 all represent the same number, decimal 31.

DEVICEFILES

Hpux **boot** and **copy** operations accept *devicefile* specifications, which have the following format:

manager(x.y.z;n,s)filename

They are called *devicefiles* because they are composed of a device name part and a file name part. In the device part, *manager(x.y.z;n,s), manager* is the name of an HP 9000 Series 800 I/O System manager (i.e. device driver), for example **disc0**. *X.y.z* is the physical hardware path to the device, identifying bus converters, slot numbers, and hardware addresses. (Bus converters are not currently supported.) *N* is the minor number which controls manager dependent functionality. *S* is the file skip count. For **tape0** and **tape1** devices, this parameter describes how many files must be skipped (from the beginning of the tape) before the desired file can be accessed. It has a default value of **0**, and is completely ignored for other devices. The file name part, *filename*, is a standard HP-UX path name. In a *devicefile* specification, the only required component is the hardware path. All other components are optional. Some *hpux* operations have defaults for particular components. A *devicefile* specification containing a device part only specifies a raw device. A *devicefile* specification containing a file name implies that the associated device part names a device containing an HP-UX file system. The named file resides in that file system. For example, the typical boot *devicefile* specification is disc0(8.0.0;0)hp-ux.

Managers

Currently, *hpux* supports the **disc0**, **tape0**, and **tape1** managers. **Disc0** manages all CS-80 discs including cartridge tape devices. **Tape0** manages the HP 7970 tape drives and **tape1** manages the HP 7974 and HP 7978 tape drives.

Hardware Paths

The hardware path in a *devicefile* specification is an arbitrary length string of numbers separated by periods. (Commas are also acceptable.) Each number identifies a hardware component. A single number is the shortest path specification. Currently, path specifications typically contain three components. In *x.y.z* above, *x* would be the MID-BUS slot number, *y* would be the CIO slot number, and *z* would be the HP IB address.

Minor Numbers

The minor number, *n*, in a *devicefile* specification controls driver dependent functionality. The manual pages for the individual drivers describe their specific minor number encodings. Currently, minor numbers for all device drivers adhere to a standard encoding scheme that places optional hardware addresses in bits 24 through 31. The logical unit number resides in bits 16 through 23. Driver specific options are found in bits 9 through 15 and the transparent mode (diagnostic bit) flag is bit 8. *Hpux* manages its own logical units and consequently ignores any logical unit specification that maybe specified in a *devicefile*. The minor number formats for **disc0**, **tape0**, and **tape1** are summarized below:

disc0

```
        1                   2                   3
8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-----------+---------------+-----+-+-------+
|T|C|  unused   | logical unit  |unit | |section|
+-+-+-----------+---------------+-----+-+-------+
```

T = Transparent (diagnostic)
C = Cartridge tape device

tape0 and **tape1**

```
        1                   2                   3
8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+---+-+---------------+---------------+
|T|R|D|B|N|BPI| | logical unit  |    unused     |
+-+-+-+-+-+---+-+---------------+---------------+
```

T = Transparent (diagnostic)
R = RTE style (no tape movement on close)
D = Disable immediate reporting mode
B = Berkeley style
N = No rewind on close
BPI = Density (0=800BPI, 1=1600BPI, 2=6250BPI)

Skip Counts

The skip count, *s*, in a *devicefile* specification controls how many files must be skipped before the desired file is reached. It is relative to the beginning of the tape, and is defined only for **tape0** and **tape1** devices. It is ignored for all others. If not specified, **0** is assumed.

File Names

File names are standard HP-UX path names. No preceding slash (/) is necessary and specifying one will cause no problems. File names are not root (i.e. /) relative. For example, with **disc0**, they are relative to the section specified in the minor number (in the device part) of the *devicefile* specification.

BOOT

The **boot** operation loads an object file from an HP-UX file system or raw device as specified by the optional *devicefile*. It then transfers control to the loaded image.

The default *devicefile* for **boot** is disc0(8.0.0;0)hp-ux. Please note that the default hardware path component is variable, and is derived from the Primary Boot Path maintained by *pdc*(1M) or interactively given to *pdc*. For more details, consult the appropriate HP 9000 Series 800 architecture document. Any missing components in a specified *devicefile* are supplied from the default above. For example, a *devicefile* of vmunix.new would actually yield disc0(8.0.0;0)vmunix.new and a *devicefile* of (8.0.1), for booting from the disc at HP-IB address 1, would yield disc0(8.0.1;0)hp-ux. Regardless of how incomplete the specified *devicefile* may be, **boot** announces the complete *devicefile* specification used to find the object file. Along with this information, **boot** gives the sizes of the **TEXT**, **DATA**, and **BSS**, segments and the entry offset of the loaded image, before transferring control to it.

The **boot** operation accepts several options. Their meanings are:

–**f***number* This option takes a number (see **NUMBERS)** and passes it as the flags word to the loaded image.

–**i***string* This option accepts a string that specifies the initial *run-level* for *init*(1M). Note that the *run-level* specified will override any *run-level* specified in an *initdefault* entry in **/etc/inittab** (see *inittab*(4)).

–**r** This option is a special case of the –**f** option. It sets the **RDB_BOOT** flag in the flags word passed to the loaded image (see *rdb(1M)).*

Boot currently places some minor restrictions on object files it can load. It accepts only the HP-UX magic numbers **SHAREMAGIC** (0410) and **DEMANDMAGIC** (0413) (see *magic*(4)). The object file must contain an Auxiliary Header of the **HPUX_AUX_ID** type and it must be the first Auxiliary Header (see *a.out*(4)).

COPY

The **copy** operation is used during first-time installation to copy installation images from one device to another, for example, from tape to disc (see **EXAMPLES**). The only default for the *devicefile* specifications is the skip count. **Copy** announces the *devicefile* specifications identifying the source and destination devices. A running total of kilobytes transferred is displayed on the console during the copy. At completion, the total size of the transfer (in exact bytes) is displayed.

EXAMPLES

Before going over specific examples of the various options and operations of *hpux*, here is an outline of the steps taken in the automatic boot process. When the system **RESET** button is depressed, *pdc* executes self-test, and assuming the hardware passes, *pdc* announces itself, issues a **BELL**, and gives the user 10 seconds to override the *autoboot* sequence, by entering any character. The following is typically displayed on the console.

```
Processor Dependent Code (PDC) revision 2544

Console path = 8.1.0
Primary boot path = 8.0.0
Alternate boot path = 8.2.3

Autoboot from primary boot path enabled.
To override, press any key within 10 seconds.
```

If no character is entered within 10 seconds, *pdc* commences the *autoboot* sequence by loading *isl* and transferring control to it. Because an *autoboot* sequence is occurring, *isl* merely announces itself, finds and executes the *autoexecute* file which, on an HP-UX system, requests that *hpux* be run with default arguments. The following is displayed on the console.

```
10 seconds expired.

Booting.
```

```
Console IO Dependent Code (IODC) revision 2544
Boot IO Dependent Code (IODC) revision 2544

Booted.

ISL Revision 2610 (860303) -- Release 9.1

ISL booting  hpux
```

Next *hpux* announces the operation it is performing, in this case **boot**, the *devicefile* from which the load image comes, and the **TEXT** size, **DATA** size, **BSS** size, and start address of the load image. The following is displayed before control is passed to the image.

```
Boot
: disc0(8.0.0;0x0)hp-ux
966616+397312+409688 start 0x6c50
```

Lastly, the loaded image, in this case an HP-UX operating system kernel, starts by giving numerous configuration and status messages. The system in the following example eventually comes to *init run-level* 2 for multi-user mode of operation.

```
Beginning I/O System Configuration.
cio channel expected at 28, found at 8
cio_ca0 address = 8
  hpib0 address = 0
    disc0 lu = 0 address = 0
    disc0 lu = 1 address = 1
    disc0 lu = 2 address = 2
    disc0 lu = 3 address = 3
  mux0 lu = 0 address = 1
  hpib0 address = 2
    lpr0 lu = 1 address = 0
    lpr0 lu = 0 address = 1
    tape0 lu = 1 address = 2
    tape1 lu = 0 address = 3
    tape1 lu = 2 address = 4
    instr0 lu = 0 address = 7
  mux0 lu = 1 address = 3
  lan0 lu = 0 address = 4
  gpio1 lu = 0 address = 5
  hpib0 address = 6
    disc0 lu = 4 address = 0
    disc0 lu = 5 address = 1
    disc0 lu = 6 address = 2
    disc0 lu = 7 address = 3
  hpib0 address = 7
    lpr0 lu = 2 address = 1
    tape0 lu = 3 address = 2
    tape1 lu = 4 address = 3
```

```
      instr0 lu = 1 address = 7
    mux0 lu = 2 address = 8
    mux0 lu = 3 address = 9
    mux0 lu = 4 address = 10
    mux0 lu = 5 address = 11
    lan0 lu = 1 address = 12
    ios0 lu = 0 address = 13
  I/O System Configuration complete.
  Configure called
  @(#)1.0 HP-UX/RT (sys.ica/S800) #1: Mon Mar 3 12:26:00 PST 1986
  real mem = 8386560
  lockable mem = 3297280
  avail mem = 5197824
  using 204 buffers containing 837632 bytes of memory
  Fri Mar 7 09:37:57 PST 1986
  Checking file systems
```

In order to use the operations and options of *hpux*, *isl* must be brought up in interactive mode. To do this simply enter a character during the 10 second interval allowed by *pdc*. *Pdc* will then ask if the primary boot path is acceptable. Answering yes (**Y**) is usually appropriate. *Pdc* will then load *isl* and *isl* will interactively prompt for commands. The following is displayed.

```
Boot from primary boot path (Y or N)?> Y

Booting.

Console IO Dependent Code (IODC) revision 2544
Boot IO Dependent Code (IODC) revision 2544

Booted.

ISL Revision 2610 (860303) -- Release 9.1

ISL>
```

Although all the operations and options of *hpux* may be used from *isl* interactively, they may also be executed from an *autoexecute* file. Refer to the appropriate documentation for building *autoexecute* files. In the examples below, all user input is in boldface type.

Default Boot

```
ISL> hpux

Boot
: disc0(8.0.0;0x0)hp-ux
966616+397312+409688 start 0x6c50
```

Entering **hpux** initiates the default boot sequence. The default manager is **disc0**, the boot path read from *pdc* is **8.0.0**, the minor number is **0** specifying not only unit zero of the device but also section 0 of the disc, and the object file name is **hp–ux**.

Booting Another Kernel

```
ISL> hpux vmunix.new

Boot
: disc0(8.0.0;0x0)vmunix.new
966616+397312+409688 start 0x6c50
```

Here *hpux* initiates a **boot** operation where the name of the object file is **vmunix.new**.

Booting from Another Section

```
ISL> hpux (;3)sys.azure/S800/vmunix

Boot
: disc0(8.0.0;0x3)sys.azure/S800/vmunix
966616+397312+409688 start 0x6c50
```

In this example, a kernel is booted from another section of the root disc. For example, let's say that kernel development takes place under /mnt/azure/root.port which happens to reside in its own section, section 3 of the root (i.e. default boot) disc. By specifying a minor number of **3**, in the above example, the object file **sys.azure/S800/vmunix** is loaded from /mnt/azure/root.port.

Booting from Cartridge Tape

```
ISL> hpux (;4194336)hp-ux

Boot
: disc0(8.0.0;0x400020)hp-ux
966616+397312+409688 start 0x6c50
```

In this example, the default boot device is an HP 7914 disc with a cartridge tape at unit 1. The minor number has the cartridge tape flag set and specifies unit 1, section 0 of the device. Although the minor number was entered in decimal format, the hexadecimal form would be accepted. Since a file name is specified, it is assumed that section 0 contains a file system.

Booting from Another Disc

```
ISL> hpux (8.0.1)hp-ux

Boot
: disc0(8.0.1;0x0)hp-ux
966616+397312+409688 start 0x6c50
```

In this example, only the hardware path is specified. All other values are boot defaults. The object file comes from the file system in section 0 of the disc, at HP-IB address 1.

Booting from a Raw Device

```
ISL> hpux tape0(8.2.2;0x20000,1)

Boot
: tape0(8.2.2;0x20000,1)
966616+397312+409688 start 0x6c50
```

This example shows booting from a raw device (i.e. no file system is on the device). Note that no file name is specified in the *devicefile*. The device is an HP 7970 tape drive and therefore **tape0** is the manager used. The tape drive is at CIO slot 2, HP-IB address 2. The first file on the tape will be skipped. The minor number specifies a tape density of 1600 BPI. Note that for **tape0** the tape must be written with 512-byte blocks.

Booting to Single User Mode

```
ISL> hpux -is

Boot
: disc0(8.0.0;0x0)hp-ux
966616+397312+409688 start 0x6c50


   Kernel Startup Messages Omitted


INIT: Overriding default level with level 's'

INIT: SINGLE USER MODE
WARNING:  YOU ARE SUPERUSER !!

#
```

In this example, the **–i** option is used to make the system come up in *run-level* **s**, for single user mode of operation.

Copying an Installation Image

```
ISL> hpux copy tape0(8.2.2;0x20000) disc0(8.0.0;0)

Copy
: tape0(8.2.2;0x20000,0) disc0(8.0.0;0x0)
24576Kb
25165824 bytes copied
```

The first file on the currently mounted tape contains an image of the root file system. The image is copied from the tape to the root disc. The tape was recorded at 1600BPI and the target section is 0. The copy terminates when **EOF** or **EOT** is detected on the tape.

Getting the Version

```
ISL> hpux -v

Version:
Release: 9.1
Release Version: @(#)1.0 HP-UX/RT (sys.hpuxboot/HPUXBOOT)
                 #2: Thu Feb 27 13:37:43 PST 1986
```

The **–v** option is used to get the version numbers of *hpux*.

DIAGNOSTICS

In the instance of an error **hpux** prints diagnostic messages which indicate the cause of the error. These messages may be grouped General, Boot, Copy, Configuration, and System Call. A description of the System Call error messages may be found in *errno*(2). The remaining messages are described below.

General

bad minor number in devicefile spec
: The minor number in the *devicefile* specification is illegal.

bad path in devicefile spec
: The hardware path in the *devicefile* specification is illegal.

command too complex for parsing
: The command line contains too many arguments.

no path in devicefile spec
: The *devicefile* specification does not contain a hardware path component and must.

panic (in hpuxboot): (display==*number*, flags==*number*) *string*
: A severe internal *hpux* error has occurred. Report to your nearest HP Field Representative.

Boot

bad magic
: The specified object file does not have a legal magic number.

bad number in flags spec
: The flags specification in the **–f** option is illegal.

booting from raw character device
: In booting from a raw device, the manager specified only has a character interface. This may cause problems if the block size is incorrect.

Isl not present, please hit system **RESET** button to continue
: An unsuccessful **boot** operation has overlayed *isl* in memory. It is impossible to return control to *isl*.

short read
: The specified object file is internally inconsistent, it is not long enough.

would overlay
: Loading the specified object file would overlay *hpux*.

Copy

cannot open destination device/file
: The destination device or file could not be opened for writing.

cannot open source device/file
: The source device or file could not be opened for reading.

fchmod failure (warning only)
: The access mode of the destination file could not be changed.

fchown failure (warning only)
: The owner and/or group of the destination file could not be changed.

fstat failure (warning only)
: One or more of the owner, group, or mode of the source file could not be determined. The default values of owner and group are **0** and **0**. The default mode is **0777**.

read failure
: An error was encountered reading from the source device or file.

umount failure on destination device
: The destination device could not be dismounted. Its file system may have been damaged as a result. *Fsck* should be run before mounting the file system.

umount failure on source device
: The source device could not be dismounted. Since it was mounted read-only, the integrity of its file system is not at risk.

write failure
: An error was encountered writing to the destination device or file.

Configuration

cannot add path, error *number*
: An unknown error has occurred in adding the hardware path to the I/O tree. The internal error number is given. Contact your HP Field Representative.

driver does not exist
: The manager specified is not configured into *hpux*.

driver is not a logical device manager
: The manager name given is not that of a logical device manager and cannot be used for direct I/O operations.

error rewinding device
: An error was encountered attempting to rewind a device.

error skipping file
: An error was encountered attempting to forward-space a tape device.

negative skip count
: The skip count, if specified, must be greater than or equal to zero.

no major number
: The specified manager has no entry in the block or character device switch tables.

path incompatible with another path
: Multiple incompatible hardware pathes have been specified.

path long
: The hardware path specified contains too many components for the specified manager.

path short
: The hardware path specified contains too few components for the specified manager.

table full
: Too many devices have been specified to *hpux*.

SEE ALSO

a.out(4), boot(1M), errno(2), fsck(1M), init(1M), inittab(4), isl(1M), magic(4), pdc(1M), rdb(1M).

NAME

init, telinit – process control initialization

SYNOPSIS

/etc/init [**0123456SsQq**]

/etc/telinit [**0123456sSQqabc**]

DESCRIPTION

Init

Init is a general process spawner. Its primary role is to create processes from a script stored in the file **/etc/inittab** (see *inittab*(4)). This file usually has *init* spawn *getty*'s on each line that a user may log in on. It also controls autonomous processes required by any particular system.

Init considers the system to be in a *run-level* at any given time. A *run-level* can be viewed as a software configuration of the system where each configuration allows only a selected group of processes to exist. The processes spawned by *init* for each of these *run-levels* is defined in the *inittab* file. *Init* can be in one of eight *run-levels*, **0–6** and **S** or **s**. The *run-level* is changed by having a privileged user run **/etc/init** (which is linked to */etc/telinit*). This user-spawned *init* sends appropriate signals to the original *init* spawned by the operating system when the system was rebooted, telling it which *run-level* to change to.

Init is invoked inside the HP-UX system as the last step in the boot procedure. The first thing *init* does is to look for **/etc/inittab** and see if there is an entry of the type *initdefault* (see *inittab*(4)). If there is, *init* uses the *run-level* specified in that entry as the initial *run-level* to enter. If this entry is not in *inittab* or *inittab* is not found, *init* requests that the user enter a *run-level* from the virtual system console, **/dev/syscon.** If an **S** (**s**) is entered, *init* goes into the *SINGLE USER* level. This is the only *run-level* that doesn't require the existence of a properly formatted *inittab* file. If **/etc/inittab** doesn't exist, then by default the only legal *run-level* that *init* can enter is the *SINGLE USER* level. In the *SINGLE USER* level the virtual console terminal **/dev/syscon** is opened for reading and writing and the command **/bin/su** is invoked immediately. To exit from the *SINGLE USER run-level* one of two options can be elected. First, if the shell is terminated (via an end-of-file), *init* will reprompt for a new *run-level.* Second, the *init* or *telinit* command can signal *init* and force it to change the *run-level* of the system.

When attempting to boot the system, failure of *init* to prompt for a new *run-level* may be due to the fact that the device **/dev/syscon** is linked to a device other than the physical system teletype (**/dev/systty**). If this occurs, *init* can be forced to relink **/dev/syscon** by typing a delete on the system teletype which is collocated with the processor.

When *init* prompts for the new *run-level*, the operator may enter only one of the digits **0** through **6** or the letters **S** or **s**. If **S** is entered *init* operates as previously described in *SINGLE USER* mode with the additional result that **/dev/syscon** is linked to the user's terminal line, thus making it the virtual system console. A message is generated on the physical console, **/dev/systty**, saying where the virtual terminal has been relocated.

When *init* comes up initially and whenever it switches out of *SINGLE USER* state to normal run states, it sets the *ioctl*(2) states of the virtual console, **/dev/syscon**, to those modes saved in the file **/etc/ioctl.syscon**. This file is written by *init* whenever *SINGLE USER* mode is entered. If this file does not exist when *init* wants to read it, a warning is printed and default settings are assumed.

If a **0** through **6** is entered *init* enters the corresponding *run-level.* Any other input will be rejected and the user will be re-prompted. If this is the first time *init* has entered a *run-level* other than *SINGLE USER*, *init* first scans *inittab* for special entries of the type *boot* and *bootwait.* These entries are performed, providing the *run-level* entered matches that of the entry before any normal processing of *inittab* takes place. In this way any special initialization of the operating system, such as mounting file systems, can take place before users are allowed onto the system.

The *inittab* file is scanned to find all entries that are to be processed for that *run-level.*

Run-level **2** is usually defined by the user to contain all of the terminal processes and daemons that are spawned in the multi-user environment.

In a multi-user environment, the *inittab* file is usually set up so that *init* will create a process for each terminal on the system.

For terminal processes, ultimately the shell will terminate because of an end-of-file either typed explicitly or generated as the result of hanging up. When *init* receives a child death signal, telling it that a process it spawned has died, it records the fact and the reason it died in **/etc/utmp** and **/etc/wtmp** if it exists (see *who*(1)). A history of the processes spawned is kept in **/etc/wtmp** if such a file exists.

To spawn each process in the *inittab* file, *init* reads each entry and for each entry which should be respawned, it forks a child process. After it has spawned all of the processes specified by the *inittab* file, *init* waits for one of its descendant processes to die, a powerfail signal, or until *init* is signaled by *init* or *telinit* to change the system's *run-level.* When one of the above three conditions occurs, *init* re-examines the *inittab* file. New entries can be added to the *inittab* file at any time; however, *init* still waits for one of the above three conditions to occur. To provide for an instantaneous response the **init Q** or **init q** command can wake *init* to re-examine the *inittab* file.

If *init* receives a *powerfail* signal (*SIGPWR*) and is not in *SINGLE USER* mode, it scans *inittab* for special powerfail entries. These entries are invoked (if the *run-levels* permit) before any further processing takes place. In this way *init* can perform various cleanup and recording functions whenever the operating system experiences a power failure.

When *init* is requested to change *run-levels* (via *telinit*), *init* sends the warning signal (**SIGTERM**) to all processes that are undefined in the target *run-level. Init* waits 20 seconds before forcibly terminating these processes via the kill signal (**SIGKILL**).

Telinit

Telinit, which is linked to */etc/init*, is used to direct the actions of *init.* It takes a one-character argument and signals *init* via the kill system call to perform the appropriate action. The following arguments serve as directives to *init.*

0–6	tells *init* to place the system in one of the *run-levels* **0–6**.
a,b,c	tells *init* to process only those **/etc/inittab** file entries having the **a**, **b** or **c** *run-level* set.
Q,q	tells *init* to re-examine the **/etc/inittab** file.
s,S	tells *init* to enter the single user environment. When this level change is effected, the virtual system teletype, **/dev/syscon**, is changed to the terminal from which the command was executed.

Telinit can only be run by someone who is super-user or a member of group **sys**.

FILES

/etc/inittab
/etc/ioctl.syscon
/dev/syscon
/dev/systty
/etc/utmp
/etc/wtmp

SEE ALSO

getty(1M), login(1), sh(1), who(1), kill(2), inittab(4), utmp(4).

DIAGNOSTICS

If *init* finds that it is continuously respawning an entry from **/etc/inittab** more than 10 times in

2 minutes, it will assume that there is an error in the command string, and generate an error message on the system console, and refuse to respawn this entry until either 5 minutes has elapsed or it receives a signal from a user *init* (*telinit*). This prevents *init* from eating up system resources when someone makes a typographical error in the *inittab* file or a program is removed that is referenced in the *inittab*.

NAME

install - install commands

SYNOPSIS

/etc/install [**−c** dira] [**−f** dirb] [**−i**] [**−n** dirc] [**−o**] [**−g** group] [**−s**] [**−u** user] file [dirx ...]

DESCRIPTION

Install is a command most commonly used in "makefiles" (see *make*(1)) to install a *file* (updated target file) in a specific place within a file system. Each *file* is installed by copying it into the appropriate directory, thereby retaining the mode and owner of the original command. The program prints messages telling the user exactly what files it is replacing or creating and where they are going.

Install is useful for installing new commands, or new versions of existing commands, in the standard directories (i.e. **/bin**, **/etc**, etc.).

If no options or directories (*dirx* ...) are given, *install* will search a set of default directories (**/bin**, **/usr/bin**, **/etc**, **/lib**, and **/usr/lib**, in that order) for a file with the same name as *file*. When the first occurrence is found, *install* issues a message saying that it is overwriting that file with *file* (the new version), and proceeds to do so. If the file is not found, the program states this and exits without further action.

If one or more directories (*dirx* ...) are specified after *file*, those directories will be searched before the directories specified in the default list.

The meanings of the options are:

−c *dira*	Installs a new command (*file*) in the directory specified by *dira*, only if it is not found. If it is found, *install* issues a message saying that the file already exists, and exits without overwriting it. May be used alone or with the **−s** option.
−f *dirb*	Forces *file* to be installed in given directory, whether or not one already exists. If the file being installed does not already exist, the mode and owner of the new file will be set to **755** and **bin**, respectively. If the file already exists, the mode and owner will be that of the already existing file. May be used alone or with the **−o** or **−s** options.
−i	Ignores default directory list, searching only through the given directories (*dirx* ...). May be used alone or with any other options other than **−c** and **−f**.
−n *dirc*	If *file* is not found in any of the searched directories, it is put in the directory specified in *dirc*. The mode and owner of the new file will be set to **755** and **bin**, respectively. May be used alone or with any other options other than **−c** and **−f**.
−o	If *file* is found, this option saves the "found" file by copying it to **OLD***file* in the directory in which it was found. This option is useful when installing a normally busy text file such as **/bin/sh** or **/etc/getty**, where the existing file cannot be removed. May be used alone or with any other options other than **−c**.
−g *group*	Causes *file* to be owned by group *group*. This option is available only to the super-user. May be used alone or with any other option.
−u *user*	Causes *file* to be owned by user *user*. This option is available only to the super-user. May be used alone or with any other option.
−s	Suppresses printing of messages other than error messages. May be used alone or with any other options.

When no directories are specified (dirx ...), or when *file* cannot be placed in one of the directories specified, *install* checks for the existence of the file */etc/syslist*. If */etc/syslist* exists, it is used to determine the final destination of *file*. If */etc/syslist* does not exist, the default directory list is further scanned in order to determine where *file* is to be located.

The file */etc/syslist* contains a list of absolute pathnames, one per line. The pathname is the "official" destination (for example */bin/echo*) of the file as it appears on a file system. */etc/syslist* serves as a master list for system command destinations. If there is no entry for *file* in the file */etc/syslist* then the default directory list is further scanned in order to determine where *file* is to be located.

Cross Generation

The environment variable *ROOT* will be used to locate the locations file (in the form *£ROOT/etc/syslist*). This is necessary in the cases where cross generation is being done on a production system. Furthermore, each pathname in *£ROOT/etc/syslist* is appended to $ROOT (for example, *£ROOT/bin/echo*) and used as the destination for *file*. Also, the default directories are also appended to *£ROOT* so that the default directories are actually *£ROOT/bin*,*£ROOT/usr/bin*,*£ROOT/etc*, *£ROOT/lib*, and *£ROOT/usr/lib*.

The file */etc/syslist (*$ROOT/etc/syslist*)* does not exist on a distribution tape; it is created and used by local sites.

SEE ALSO

cpset(1M), make(1).

BUGS

Install cannot create alias links for a command (for example, *vi*(1) is an alias link for *ex*(1)).

NAME

isl – initial system loader

DESCRIPTION

Isl implements the operating system independent portion of the bootstrap process. It is loaded and executed after self-test and initialization have completed successfully.

The processor contains special purpose memory for maintaining critical configuration related parameters (e.g. Primary Boot, Alternate Boot, and Console Paths). Two forms of memory are supported: Stable Storage and Non-Volatile Memory (NVM).

Typically, when control is transferred to *isl,* an *autoboot* sequence takes place. An *autoboot* sequence allows a complete bootstrap operation to occur with no intervention from an operator. *Isl* executes commands from the *autoexecute* file in a script-like fashion. *Autoboot* is enabled by a flag in Stable Storage.

Autosearch is a mechanism that automatically locates the boot and console devices. It is currently not implemented on the Model 840 but will be will be implemented on future Series 800 processors.

During an *autoboot* sequence, *isl* displays its revision and the name of any utility it executes. However, if *autoboot* is disabled, after *isl* displays its revision, it then prompts for input from the console device. Acceptable input is any *isl* command name or the name of any utility available on the system. If a non-fatal error occurs or the executed utility returns, *isl* again prompts for input.

Commands

There are several commands available in *isl.* The following is a list of them with a short description. Parameters may be entered on the command line following the command name. They must be separated by spaces. *Isl* prompts for any necessary parameters that are not entered on the command line.

? help	Help -- Lists commands and available utilities
listf ls	Lists available utilities
autoboot	Enables or disables the *autoboot* sequence Parameter -- on or off
autosearch	Enables or disables the *autosearch* sequence Parameter -- on or off
primpath	Modify the Primary Boot Path Parameter -- Primary Boot Path in decimal
altpath	Modify the Alternate Boot Path Parameter -- Alternate Boot Path in decimal
conspath	Modify the Console Path Parameter -- Console Path in decimal
lsautofl listautofl	Lists contents of the *autoexecute* file
display	Displays the Primary Boot, Alternate Boot, and Console Paths
readnvm	Displays the contents of one word of NVM in hexadecimal Parameter -- NVM address in decimal or standard hexadecimal notation
readss	Displays the contents of one word of Stable Storage in hexadecimal Parameter -- Stable Storage address in decimal or standard hexadecimal notation

DIAGNOSTICS

Isl displays diagnostic information through error messages written on the console and display codes on the hexadecimal LED display.

For the display codes, **CE0x** are informative only. **CE1x** and **CE2x** indicate errors, some of which are fatal and cause the system to halt. Other errors merely cause *isl* to display a message. During normal operation, the self-test light is yellow. However, during fatal errors, the self-test light is red.

Non-fatal errors during an *autoboot* sequence cause the *autoboot* sequence to be aborted and *isl* to prompt for input. After non-fatal errors during an interactive *isl* session, *isl* merely prompts for input.

Fatal errors cause the system to halt. The problem must be corrected and the system **RESET** to recover.

CE00	*Isl* is executing.
CE01	*Isl* is *autoboot*ing from the *autoexecute* file.
CE02	Cannot find an *autoexecute* file. *Autoboot* aborted.
CE03	No console found, *isl* can only *autoboot.*
CE05	Directory of utilities is too big, *isl* reads only 2K bytes.
CE06	*Autoexecute* file is inconsistent. *Autoboot* aborted.
CE07	Utility file header inconsistent: SOM values invalid.
CE08	*Autoexecute* file input string exceeds 2048 characters. *Autoboot* aborted.
CE09	*Isl* command or utility name exceeds 10 characters.
CE0F	*Isl* has transferred control to the utility.
CE10	Internal inconsistency: Volume label - **FATAL.**
CE11	Internal inconsistency: Directory - **FATAL.**
CE12	Error reading *autoexecute* file.
CE13	Error reading from console - **FATAL.**
CE14	Error writing to console - **FATAL.**
CE15	Not an *isl* command or utility.
CE16	Utility file header inconsistent: Invalid System ID.
CE17	Error reading utility file header.
CE18	Utility file header inconsistent: Bad magic number.
CE19	Utility would overlay *isl* in memory.
CE1A	Utility requires more memory than is configured.
CE1B	Error reading utility into memory.
CE1C	Incorrect checksum: Reading utility into memory.
CE1D	Console needed - **FATAL.**
CE1E	Internal inconsistency: Boot device class - **FATAL.**
CE21	Destination memory address of utility is invalid.
CE22	Utility file header inconsistent: *pdc_cache* entry.

CE23	Internal inconsistency: *iodc_entry_init* - **FATAL.**
CE24	Internal inconsistency: *iodc_entry_init* - console - **FATAL.**
CE25	Internal inconsistency: *iodc_entry_init* - boot device - **FATAL.**
CE26	Utility file header inconsistent: Bad aux_id.
CE27	Bad utility file type.

SEE ALSO

boot(1M), hpuxboot(1M), pdc(1M).

NAME

killall – kill all active processes

SYNOPSIS

/etc/killall [signal]

DESCRIPTION

Killall is a procedure used by **/etc/shutdown** to kill all active processes not directly related to the shutdown procedure.

Killall is chiefly used to terminate all processes with open files so that the mounted file systems will be unbusied and can be unmounted. *Killall* sends the specified *signal* to all user processes in the system, with the following exceptions:

> the *init* process;
>
> all processes (including background processes) associated with the terminal from which *killall* was invoked;
>
> any *ps –ef* process, if owned by *root*;
>
> any *sed –e* process, if owned by *root*;
>
> any *shutdown* process;
>
> any *killall* process;
>
> any */etc/rc* process.

Killall obtains its process information from *ps*(1), and thus may not be able to perfectly identify which processes to signal.

If no *signal* is specified, a default of **9** (kill) is used.

Killall is invoked automatically by *shutdown*(1M). The use of *shutdown* is recommended over using *killall* by itself.

FILES

/etc/shutdown

SEE ALSO

fuser(1M), kill(1), ps(1), shutdown(1M), signal(2).

NAME

link, unlink – exercise link and unlink system calls

SYNOPSIS

/etc/link file1 file2
/etc/unlink file

DESCRIPTION

Link and *unlink* perform their respective system calls on their arguments, abandoning most error checking. These commands may only be executed by the super-user.

RETURNS

0 - successful *link*.
1 - input syntax error.
2 - *link* call failed (*unlink* will never report failure).

SEE ALSO

rm(1), link(2), unlink(2).

INTERNATIONAL SUPPORT

8-bit filenames.

NAME

lpadmin – configure the LP spooling system

SYNOPSIS

/usr/lib/lpadmin **–p** printer [options]
/usr/lib/lpadmin **–x** dest
/usr/lib/lpadmin **–d**[dest]

DESCRIPTION

Lpadmin configures LP spooling systems to describe printers, classes and devices. It is used to add and remove destinations, change membership in classes, change devices for printers, change printer interface programs and to change the system default destination. *Lpadmin* may not be used when the LP scheduler, *lpsched*(1M), is running, except where noted below.

Exactly one of the **–p**, **–x** or **–d** options must be present for every legal invocation of *lpadmin*.

–p*printer* names a *printer* to which all of the *options* below refer. If *printer* does not exist then it will be created.

–x*dest* removes destination *dest* from the LP system. If *dest* is a printer and is the only member of a class, then the class will be deleted, too. No other *options* are allowed with **–x**.

–d[*dest*] makes *dest*, an existing destination, the new system default destination. If *dest* is not supplied, then there is no system default destination. This option may be used when *lpsched*(1M) is running. No other *options* are allowed with **–d**.

The following *options* are only useful with **–p** and may appear in any order. For ease of discussion, the printer will be referred to as *P* below.

–c*class* inserts printer *P* into the specified *class*. *Class* will be created if it does not already exist.

–e*printer* copies an existing *printer's* interface program to be the new interface program for *P*.

–h indicates that the device associated with *P* is hardwired. This *option* is assumed when creating a new printer unless the **–l** *option* is supplied.

–i*interface* establishes a new interface program for *P*. *Interface* is the pathname of the new program.

–l indicates that the device associated with *P* is a login terminal. The LP scheduler, *lpsched*(1M), disables all login terminals automatically each time it is started. Before re-enabling *P*, its current *device* should be established using *lpadmin*.

–m*model* selects a model interface program for *P*. *Model* is one of the model interface names supplied with the LP software (see **Models** below).

–r*class* removes printer *P* from the specified *class*. If *P* is the last member of the *class*, then the *class* will be removed.

–v*device* associates a new *device* with printer *P*. *Device* is the pathname of a file that is writable by the LP administrator, *lp*. Note that there is nothing to stop an administrator from associating the same *device* with more than one *printer*. If only the **–p** and **–v** *options* are supplied, then *lpadmin* may be used while the scheduler is running.

Restrictions

When creating a new printer, the **–v** option and one of the **–e**, **–i** or **–m** options must be supplied. Only one of the **–e**, **–i** or **–m** options may be supplied. The **–h** and **–l** key letters are mutually exclusive. Printer and class names may be no longer than 14 characters and must consist entirely of the characters **A-Z**, **a-z**, **0-9** and _ (underscore).

Models

Model printer interface programs are supplied with the LP software. They are shell procedures, C programs, or other executable programs which interface between *lpsched*(1M) and devices. All models reside in the directory **/usr/spool/lp/model** and may be used as is with *lpadmin* **−m**. Models should have 644 permission if owned by lp and bin, or 664 permission if owned by bin and bin. Alternatively, LP administrators may modify copies of models and then use *lpadmin* **−i** to associate them with printers. See *mklp*(1M) for details of the printer models provided with your HP-UX system.

The LP model interface program does the actual printing on the device that is currently associated with the printer. The LP spooler sets standard in to **/dev/null** and standard out and standard error to the device specified in the **−v** option of *lpadmin*. The interface program is invoked then for printer **P** from the directory **/usr/spool/lp** as follows:

```
interface/P id user title copies options file ...
```

id	is the request returned by *lp*.
user	is the logname of the user who made the request.
title	is the optional title specified with the **−t** option of *lp*.
copies	is the number of copies to be printed.
options	is a blank separated list of class-dependent or printer-dependent options specified with the **−o** option of *lp*.
file	is the full pathname of the file to be printed.

Given the command line arguments and the output directed to the device, interface programs may format their output in any way they choose.

When the printing is completed, it is the responsibility of the interface program to exit with a code indicative of the success of the print job. A return value of **0** indicates that the job completed successfully. Values of **1** to **127** indicate that some error was encountered. This problem will not effect future print jobs. *lpsched* notifies users by mail that there was an error in printing the request. When problems are detected which are likely to effect future print jobs, the interface program would be well to disable the printer so that print requests are not lost.

EXAMPLES

Assuming there is an existing Hewlett-Packard 2934A line printer named *lp2*, it will use the **hp2934a** model interface after the command:

```
/usr/lib/lpadmin -plp2 -mhp2934a
```

FILES

/usr/spool/lp/*

SEE ALSO

accept(1M), enable(1), lp(1), lpsched(1M), lpstat(1), mklp(1M), nroff(1).

INTERNATIONAL SUPPORT

messages.

NAME

lpsched, lpshut, lpmove – start/stop the LP request scheduler and move requests

SYNOPSIS

/usr/lib/lpsched [−v]
/usr/lib/lpshut
/usr/lib/lpmove requests dest
/usr/lib/lpmove dest1 dest2

DESCRIPTION

Lpsched schedules requests taken by *lp*(1) for printing on line printers. *Lpsched*(1M) is typically invoked in **/etc/rc**. This creates a process which runs in the background until *lpshut* is executed. The activity of the process is recorded in **/usr/spool/lp/log**. If the **−v** option is invoked, a verbose record of the *lpsched* process is captured.

Lpshut shuts down the line printer scheduler. All printers that are printing at the time *lpshut* is invoked will stop printing. Requests that were printing at the time a printer was shut down will be reprinted in their entirety after *lpsched* is started again. All LP commands perform their functions even when *lpsched* is not running.

Lpmove moves requests that were queued by *lp*(1) between LP destinations. This command may be used only when *lpsched* is not running.

The first form of the command moves the named *requests* to the LP destination, *dest*. *Requests* are request ids as returned by *lp*(1). The second form moves all requests for destination *dest1* to destination *dest2*. As a side effect, *lp (1)* will reject requests for *dest1*.

Note that *lpmove* never checks the acceptance status (see *accept*(1M)) for the new destination when moving requests.

FILES

/usr/spool/lp/*

SEE ALSO

accept(1M), enable(1), lp(1), lpadmin(1M), lpstat(1).

INTERNATIONAL SUPPORT

messages.

NAME

mkdev – make device files

SYNOPSIS

/etc/mkdev

REMARKS

This command is implemented as a shell script, and will differ between the different implementations of HP-UX. This description applies to all versions, and further details will be found in the commentary in the script.

DESCRIPTION

This shell script helps the superuser install and maintain an HP-UX system. It consists of a machine-dependent list of commands which create one of each possible type of device file, with suggested default device addresses. It also creates mount directories for mountable volumes and changes permissions as appropriate for the device files.

This command makes it easier to build (or rebuild) special files all at once.

Mkdev automatically changes the working directory (using *cd*) to **/dev** before starting execution.

Mkdev is specifically intended for modification before (each) use. Command lines for non-desired devices should be commented out with "#" so that they are still available for later use. You may want to use shorter device names than those suggested, especially for default devices. For HP-UX naming conventions, see *intro*(7).

SEE ALSO

chmod(1), mkdir(1), mknod(1M), intro(7).

DIAGNOSTICS

Each command line in **mkdev** is echoed as it is executed. Error messages, if any, are generated by the commands invoked.

Since the super-user must modify this script before using it the first time, an error is given if it has not been modified.

AUTHOR

Mkdev was developed by the Hewlett-Packard Company.

NAME

mkfs – construct a file system

SYNOPSIS

/etc/mkfs special size [nsect ntrack blksize fragsize ncpg minfree rps nbpi]
/etc/mkfs special proto [nsect ntrack blksize fragsize ncpg minfree rps nbpi]

REMARKS

HFS file systems are normally created with the *newfs*(1M) command.

DESCRIPTION

Mkfs constructs a file system by writing on the special file *special. size* specifies the number of DEV_BSIZE blocks in the file system. *Mkfs* builds a file system with a root directory and a *lost+found* directory. (see *fsck*(1M)) The FS_CLEAN magic number for the file system is stored in the super block.

The optional arguments allow fine tune control over the parameters of the file system.

Nsect specifies the number of sectors per track on the disk.

Ntrack specifies the number of tracks per cylinder on the disk.

Blksize gives the primary block size for files on the file system. It must be a power of two, currently selected from 4096 or 8192.

Fragsize gives the fragment size for files on the file system. The **fragsize** represents the smallest amount of disk space that will be allocated to a file. It must be a power of two currently selected from the range DEV_BSIZE to MAXBSIZE.

Ncpg specifies the number of disk cylinders per cylinder group. This number must be in the range 1 to 32.

Minfree specifies the minimum percentage of free disk space allowed. Once the file system capacity reaches this threshold, only the super-user is allowed to allocate disk blocks. The default value is 10%. If a disk does not revolve at 60 revolutions per second, the **rps** parameter may be specified. **nbpi** specifies the number of data bytes (amount of user file space) per inode slot. The number of inodes is calculated as a function of the file system size. If **nbpi** is not valid, its value defaults to 2048.

If the second argument is a file name that can be opened, mkfs assumes it to be a prototype file proto, and will take its directions from that file. The prototype file contains tokens separated by spaces or new lines. The first token is the name of a file to be copied onto block zero as the bootstrap program (usually /etc/BOOT). If the name of a file is "" then it is ignored. The second token is a number specifying the number of DEV_BSIZE byte blocks in the file system. The next tokens comprise the specification for the root directory. File specifications consist of tokens giving the mode, the user-id, the group id, and the initial contents of the file. The syntax of the contents field depends on the mode.

The mode token for a file is a 6 character string. The first character specifies the type of the file. (The characters **–bcd** specify regular, block special, character special and directory files respectively.) The second character of the type is either **u** or – to specify set-user-id mode or not. The third is **g** or – for the set-group-id mode. The rest of the mode is a three digit octal number giving the owner, group, and other read, write, execute permissions, see *chmod*(1).

Two decimal number tokens come after the mode; they specify the user and group ID's of the owner of the file.

If the file is a regular file, the next token is a pathname whence the contents and size are copied.

If the file is a block or character special file, two decimal number tokens follow which give the major and minor device numbers.

If the file is a directory, *mkfs* makes the entries **.** and **..** and then reads a list of names and (recursively) file specifications for the entries in the directory. The scan is terminated with the token **$**.

A sample prototype specification follows:

```
/etc/BOOT
4872
d—777 3 1
usr     d—777 3 1
        sh      ——755 3 1 /bin/sh
        ken     d—755 6 1
                $
        b0      b—644 3 1 0 0
        c0      c—644 3 1 0 0
        $
$
```

HARDWARE DEPENDENCIES

The HFS file system is implemented on Series 200 beginning with HP-UX Release 5.0, and on Series 300 and Series 800.

AUTHOR

Mkfs[HFS] was developed by the Hewlett-Packard Company, and the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

chmod(1), dir(4), fs[HFS](4), fsck[HFS](1M), fsclean(1M), newfs[HFS](1M).

BUGS

No way to specify links in the **proto** file.

NAME

mklp – configure the LP spooler subsystem

SYNOPSIS

/etc/mklp

REMARKS

This command is implemented as a shell script, and will differ between the different implementations of HP-UX. This description applies to all versions, and further details will be found in the commentary in the script.

DESCRIPTION

This shell script helps the superuser configure the printers into the LP spooler which are supported on the particular HP-UX system. The administration of all printers in the LP spooler subsystem is similar, however in general there are *options* made available by the printer model which differ from printer to printer. These are described within the **mklp** script itself.

This command makes is easier to configure the LP spooler all at once. If desired, it can also be used to rebuild the subsystem.

While the **mklp** script gives some indication as to how the device special files are to be defined, the **mkdev** script should also be used in determining the major and minor number.

Mklp is specifically intended for modification before (each) use. Command lines for printers which will not be used should be commented out with "#" so that they are still available for later use.

DIAGNOSTICS

Each command line in **mklp** is echoed as it is executed. Error messages, if any, are generated by the commands invoked.

Since the super-user must modify this script before using it the first time, an error is given if it has not been modified.

AUTHOR

Mklp was developed by the Hewlett-Packard Company.

SEE ALSO

chmod(1), mkdev(1), mknod(1M).

NAME

mknod – create special and fifo files

SYNOPSIS

/etc/mknod name **c** | **b** major minor
/etc/mknod name **p**
/etc/mknod name **n** nodename

DESCRIPTION

Mknod makes a directory entry and corresponding inode for a special file. *Name* is the path name of the special file to be created.

In the first **SYNOPSIS** line shown, the second argument should be **b** if the special file is block-type (disks, tape), or **c** if it is character-type (other devices). *Major* and *minor* are numbers specifying the major device type (e.g. device driver number) and the minor device number (typically, but not exclusively, the unit, drive, HP-IB bus address and/or line number). The assignment of major and minor device numbers is specific to each HP-UX system. *Major* and *minor* may be specified in any of hexadecimal, octal, or decimal, using the C language conventions (Decimal numbers must not have a leading zero, octal must have a leading zero, and hexadecimal must have a leading zero followed by 'x'.)

Mknod can also be used to create fifo's (a.k.a named pipes) (second case in **SYNOPSIS** above).

Mknod can also be used to create a network special file (third case in **SYNOPSIS** above). A network special file addresses another node on a local area network. *Nodename* is the name by which the node is known on the network.

A real ID of 0 (super-user) is required on the first and third synopsis shown above. All users may use *mknod* in the form shown in the second synopsis.

The newly created file has a mode of 0666, as modified by the current setting of the user's *umask*.

HARDWARE DEPENDENCIES

Integral PC

Creation of network special files using the form of *mknod* shown in the third line of **SYNOPSIS** is not supported.

SEE ALSO

lsdev(1M), mknod(2), mknod(4).

NAME

mkrs – construct a recovery system

SYNOPSIS

/etc/mkrs [**–v**] [**–f** *recoverydevice*] [**–r** *rootdevice*] [**–t** *type*] [**–m** *machinetype*]

Remarks:

This page describes *mkrs* as implemented on Series 300, and 500 computers.

DESCRIPTION

Mkrs constructs a recovery system on removable media, providing the user with the capability of booting from and rooting to the recovery system. If a system is unbootable due to a corrupt root disc, the user can boot the recovery system and use the tools it provides to repair the corrupt disc.

Mkrs uses both block special and character special files for both the root device and the recovery system device. All four special files must exist for *mkrs* to work.

The following *options* are recognized:

–v	Normally, *mkrs* does its work silently. The **v** (verbose) option prints a running history of the construction process.
–f *recoverydevice*	identifies the block special device file on which the recovery system will be written. The default is **ct** which selects special files */dev/ct* and */dev/rct.*
–r *rootdevice*	is the name of the block special device file of root file system for which the recovery system is being built to protect. The default is **hd** which selects special files */dev/hd* and */dev/rhd.*
–t *type*	**–t type** specifies the type of removable media on which the recovery system is to be built. **Type** can be either **ct** specifying cartridge tape or **md** specifying 3.5-inch double-sided, double-density micro disc. If **md** is selected, two micro discs are required to contain the recovery system and you will be prompted when to change discs. The default is **ct.**
–m *machinetype*	specifies which type of machine (Series 300, or 500) is running this software. Normally, *mkrs* will properly identify the machine type. This option can be used if your machine does not identify itself to *mkrs.*

HARDWARE DEPENDENCIES

Series 300: The recovery system uses the swap area of the system being repaired for its swap space.

Series 500: The recovery system can only be built on cartridge tape; micro discs are not supported.

SEE ALSO

oscp(1M), sdfinit(1M), rootmark(1M), osck(1M), config(1M), mkfs(1M), and the *HP-UX System Administrator Manual.*

BUGS

Incorrectly specifying the recovery device may cause file system damage during recovery system construction.

WARNING

The recovery system provides super-user capabilities. It is recommended that the system administrator have exclusive responsibility for its use.

NAME

mount, umount – mount and dismount file system

SYNOPSIS

/etc/mount [special directory [**–r**] [**–f**]]
/etc/mount -a

/etc/umount special
/etc/umount -a

DESCRIPTION

Mount announces to the system that a removable file system is present on the device *special*. The *directory* must exist already; it becomes the name of the root of the newly mounted file system. *Directory* must be given as an absolute path name.

These commands maintain a table of mounted devices in /etc/mnttab. If invoked with no arguments, *mount* prints the table.

The optional argument **–r** indicates that the file system is to be mounted read-only. Physically write-protected file systems must be mounted in this way or errors will occur when access times are updated, whether or not any explicit write is attempted.

Umount announces to the system that the removable file system previously mounted on device *special* is to be removed.

The **–f** option indicates that the file system should be mounted even if the file system clean flag indicates that the file system should be fsck'ed before mounting.

If the **–a** option is present for either *mount* or *umount* , and all of the optional fields in */etc/checklist* are included and supported, all of the file systems described in */etc/checklist* are attempted to be mounted or dismounted. In this case, *special* and *directory* are taken from */etc/checklist*. The *special* file name used is the block special name from */etc/checklist*.

DIAGNOSTICS

Attempts to mount a currently-mounted volume under another name will result in an error [EBUSY].
Special and *directory* names recorded in */etc/mnttab* are truncated to MNTLEN bytes.

Umount complains if the special file is not mounted or if it is busy. The file system is busy if it contains an open file or some user's working directory.

HARDWARE DEPENDENCIES

The HFS file system is implemented on Series 200 beginning with HP-UX Release 5.0, and on Series 300 and Series 800.

WARNINGS

Some degree of validation is done on the file system, however it is generally unwise to mount garbage file systems.

FILES

/etc/checklist	file system table
/etc/mnttab	mount table

AUTHOR

Mount[HFS] was developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

fsclean(1M), mount(2), mnttab(4), checklist(4).

NAME

mount, umount – mount and unmount file system

SYNOPSIS

/etc/mount [special directory [**−r**] [**−f**]]

/etc/umount special

DESCRIPTION

Mount announces to the system that a removable file system is present on the device *special*. The *directory* must exist already; it becomes the name of the root of the newly mounted file system. *Directory* must be given as an absolute path name.

These commands maintain a table of mounted devices. If invoked with no arguments, *mount* prints the table.

The -r option indicates that the file is to be mounted read-only. Physically write-protected and magnetic tape file systems must be mounted in this way or errors will occur when access times are updated, whether or not any explicit write is attempted.

The -f option indicates that the file system should be mounted even if the file system clean flag indicates that the file system should be fsck'ed before mounting.

Umount announces to the system that the removable file system previously mounted on device *special* is to be removed.

HARDWARE DEPENDENCIES

The Structured Directory Format file system, SDF, is implemented on Series 500. The BFS (Bell File System) file system is implemented on the Integral PC and on Series 200 prior to HP-UX Release 5.0.

Series 500:

Warning: if virtual memory is brought up on a volume other than the root volume, and if that volume is then mounted, it cannot be unmounted.

FILES

/etc/mnttab	mount table

SEE ALSO

mount(2), mnttab(4).

DIAGNOSTICS

Attempts to mount a currently-mounted volume under another name will result in an error [EBUSY].

If an attempt to read and (partially) verify the disk label information fails, the mount will fail.

Umount complains if the special file is not mounted or if it is busy. The file system is busy if it contains an open file or some user's working directory.

BUGS

Some degree of validation is done on the file system, however it is generally unwise to mount garbage file systems.
The third parameter may be anything which has the effect of **−r**.
An error will occur if *mnttab* does not exist.
Names are truncated to MNTLEN bytes (see *mnttab*(4)).

NAME

mvdevs – move mass storage device files to */dev* subdirectories

SYNOPSIS

/etc/mvdevs [**–ivlu**]

Remarks:

This command is supported only on the Series 300, and is intended as a one time tool to assist in adopting the new device naming conventions for magnetic storage devices.

DESCRIPTION

This command is provided to assist the HP-UX system administrator in relocating device files for magnetic storage media (hard and floppy disks, magnetic and cartridge tapes) to the newly supported device subdirectories */dev/*[*r*]*dsk*, */dev/rmt*, */dev/*[*r*]*ct*. The command creates the new directories, taking care not to destroy existing device files with the same names, and moves all supported magnetic storage devices to these new locations, renaming them with the newly adopted naming conventions for HP-UX devices as it does so.

A full discussion of the new naming convention is contained in *intro*(7), *disk*(7), *ct*(7), *mt*(7) and in the *HP-UX System Adminstrator Manual.*

This command also automatically edits the */etc/backup* and */etc/backupf* scripts as well as the */etc/checklist* data file, changing all occurances of the old names to the new ones. It also leaves a script, */etc/sed.mvdevs*, which can be used to perform the same editing function on any user-written files that depend upon the absolute names of device files. To edit user-written files using the sed script, execute the *sed* command as follows:

sed –f /etc/sed.mvdevs *source_file* > *dest_file*

This command copies *source_file* to *dest_file*, changing any occurrences of old device file names so that they conform to the new file naming convention.

Mvdevs attempts to retain the function of specific devices. For example, */dev/rmt8* (the old default device for the *tar*(1) command) is moved to */dev/rmt/0m* (the new default device for tar). Similarly, */dev/rmt12* is moved to */dev/rmt/0mn*, the new default for the *mt*(1) command.

Mvdevs interogates CS/80 and SS/80 mass storage devices to determine their type, then selects a subdirectory for placement. For this reason, devices should be connected and power applied before running this program.

While it is not mandatory that *mvdevs* be used to move devices, it is strongly recommended because the old device locations in */dev* are considered obsolete and some commands may not work with them in future releases.

The options are:

–i Move files in interactive mode. The user is asked to verify each move. If the default name is not acknowledged, the user is prompted to provide a name, skip the file, or abort the program.

–v Verbose mode. Print out details of all actions taken.

–l Use long names instead of the short names. Devices will be assigned the full name described in Section 4 of the *HP-UX Reference*. For example, disk */dev/hd1* with a minor number of 0x0e0200 will be moved to */dev/dsk/c1402d0s0* instead of */dev/dsk/1s0*. Note that */dev/root* will still be moved to */dev/dsk/0s0*, since its minor number is derived at boot time. Use of this option defeats the rational naming of magnetic tape devices expected by tar(1) and mt(1).

–u Undo all changes made by the last previously executed *mvdevs* command. *Mvdevs* maintains a log of all changes made in file */etc/mvdevs.log*. The contents of that file are then used by the **–u** option to reverse the previous changes. If the file does not exist or is empty, no changes are made.

FILES

/etc/sed.mvdevs	File containing *sed* commands that can be used to edit user files containing references to device files that were changed by the *mvdevs* command.
/etc/mvdevs.log	File containing a record of all changes made by *mvdevs* command. This file is used by the **−u** option when reversing changes from a previous *mvdevs* command. The **−u** option removes this file when finished.

SEE ALSO

intro(7), disk(7), ct(7), mt(7), sed(1).
HP-UX System Adminstrator Manual.

NAME

mvdir – move a directory

SYNOPSIS

/etc/mvdir dir newdir

DESCRIPTION

Mvdir moves one directory tree into another existing directory (within the same file system), or renames a directory without moving it.

Dir must be an existing directory.

If *newdir* does not exist but the directory that would contain it does, *dir* is moved and/or renamed to *newdir*. Otherwise, *newdir* must be an existing directory not already containing an entry with the same name as the last pathname component of *dir*. In this case, *dir* is moved and becomes a subdirectory of *newdir*. The last pathname component of *dir* is used as the name for the moved directory.

Mvdir will refuse to move *dir* if the path specified by *newdir* would be a descendant directory of the path specified by *dir*. (For example, *mvdir* **x/y x/y/z/t** is prohibited.) Such cases are not allowed because cyclic sub-trees would be created.

Mvdir will not allow "**.**" to be moved.

Only the super-user can use *mvdir*.

WARNINGS

The restriction on names is intended to prevent the creation of cyclic sub-trees that may be inaccessible. *Mvdir* checks for such cases strictly by name, thus creating such a sub-tree is still possible. For example, "*mvdir* **x/y x/y/z/t**" will report an error, but "*mvdir* **x/y ./x/y/z/t**" (effectively the same command) will not, and a cyclic sub-tree will result. The super-user is cautioned to be very careful in the use of the names "**.**" and "**..**" while moving directories. It is possible to move "**.**" by using another name which specifies the current working directory, for example, "*mvdir* **./subdir/..** *newdir*".

SEE ALSO

mkdir(1), cp(1).

INTERNATIONAL SUPPORT

8-bit filenames.

NAME

ncheck – generate names from i-numbers

SYNOPSIS

/etc/ncheck [**–i** numbers] [**–a**] [**–s**] [file-system]

MARKETING MODEL

Level C

TECHNICAL MODEL

Large Machine
SVID

DESCRIPTION

Ncheck with no argument generates a path-name vs. i-number list of all files on the volumes specified by the file **/etc/checklist.** Names of directory files are followed by **/.**. The options are as follows:

–i	reduces the report to only those files whose i-numbers are specified on the command line in the *numbers* list.
–a	allows printing of the names **.** and **..**, which are ordinarily suppressed.
–s	reduces the report to special files and files with set-user-ID mode; it is intended to discover concealed violations of security policy.

A file system may be specified.

The report is in no useful order, and probably should be sorted.

SEE ALSO

sort(1), fsck[BFS](1M), fsck[HFS](1M), checklist(4).

DIAGNOSTICS

When the file system structure is improper, **??** denotes the "parent" of a parentless file and a path-name beginning with **...** denotes a loop.

INTERNATIONAL SUPPORT

8-bit filenames.

NAME

newfs – construct a new file system

SYNOPSIS

/etc/newfs [**-n**] [**-v**] [**mkfs-options**] **special disk-type**

DESCRIPTION

Newfs is a "friendly" front-end to the *mkfs*(1M) program. *Newfs* will look up the type of disk a file system is being created on in the disk description file */etc/disktab*, calculate the appropriate parameters to use in calling *mkfs*, then build the file system by forking *mkfs* and, if the file system is a root section, install the necessary bootstrap programs in the initial 8192 bytes of the device. The **−n** option prevents the bootstrap programs from being installed. **special** is the character special file for the disk and **disk-type** is the type of the disk as specified in /etc/disktab.

If the **−v** option is supplied, *newfs* will print out its actions, including the parameters passed to *mkfs*.

Options which may be used to override default parameters passed to *mkfs* are:

−s size The size of the file system in DEV_BSIZE blocks.

−b block-size The block size of the file system in bytes.

−f frag-size The fragment size of the file system in bytes.

−t #tracks/cylinder
The number of tracks per cylinder.

−c #cylinders/group
The number of cylinders per cylinder group in a file system. The default value used is 16.

−m free space %
The percentage of space reserved from normal users; the minimum free space threshold. The default value used is 10%.

−r revolutions/minute
The speed of the disk in revolutions per minute (normally 3600).

−i number of bytes per inode
This specifies the density of inodes in the file system. The default is to create an inode for each 2048 bytes of data space. If fewer inodes are desired, a larger number should be used; to create more inodes a smaller number should be given.

FILES

/etc/disktab	for disk geometry and file system section information

HARDWARE DEPENDENCIES

The HFS file system is implemented on Series 200 beginning with HP-UX Release 5.0, and on Series 300 and Series 800.

Series 800
Newfs will not install bootstrap programs in a root section since the boot programs are kept in a separate section.

AUTHOR

Newfs was developed by the University of California, Berkeley.

SEE ALSO

fsck[HFS](1M), mkfs[HFS](1M), tunefs[HFS](1M), disktab[HFS](4), fs[HFS](4).

NAME

opx25 – execute HALGOL programs

SYNOPSIS

opx25 [–fscriptname] [–cchar] [–ofile-descriptor] [–ifile-descriptor] [–nstring] [–d] [–v]

DESCRIPTION

HALGOL is a simple language for communicating with devices such as modems and X.25 PADs. It has simple statements like 'send xxx' and 'expect yyy' that are described below.

Options:

–f script
: Causes *opx25* to read script as the input program. If -f is not specified then *opx25* reads stdin for the script.

–c char
: Causes *opx25* to use 'char' as the first character in the input stream instead of actually reading it from the input descriptor. This is useful sometimes when the program that calls *opx25* is forced to read a character but then cannot "unread" it.

–o number
: Causes *opx25* to use 'number' for the output file descriptor (ie, the device to use for 'send'). The default is 1.

–i number
: Causes *opx25* to use 'number' for the input file descriptor (ie, the device to use for 'expect'). The default is 0.

–n string
: Causes *opx25* to save this string for use when "\#" is encountered in a "send" command.

–d Causes *opx25* to turn on debugging mode.

–v Causes *opx25* to turn on verbose mode.

An *opx25* script file contains lines of the following type:

(empty)
: Empty lines are ignored.

/ Lines beginning with a slash "/" are ignored (comments)

ID ID denotes a label. ID is limited to alphanumerics or "_".

send STRING
: STRING must be surrounded by double quotes. The text is sent to the device specified by the -o option. Non-printable characters are represented as in C, ie, as \DDD, where DDD is the octal ascii character code. "\#" in a send string is the string that followed the -n option.

break Send a break "character" to the device.

expect NUMBER STRING
: Here NUMBER is how many seconds to wait before giving up. 0 means wait forever, but this isn't advised. Whenever STRING appears in the input within the time allotted, the command succeeds. Thus, it isn't necessary to specify the entire string. For example, if you know that the PAD will send several lines followed by a "@" prompt, you could just use "@" as the string.

run program args
: The program (sleep, date, whatever) is run with the args specified. Don't use quotes here. Also, the program is invoked directly (with execp), so wild cards, redirection, etc. are not possible.

error ID
If the most recent expect or run encountered an error, go to the label ID.

exec program args
Like run, but doesn't fork.

echo STRING
Like send, but goes to stderr instead of to the device.

set debug
Sets the program in debug mode. It echoes each line to **/tmp/opx25.log,** as well as giving the result of each expect and run. This can be useful for writing new scripts. The command "set nodebug" will turn off this feature.

set log Sends subsequent incoming characters to **/usr/spool/uucp/X25LOG.** This can be used in the *.in file as a security measure, since part of the incoming data stream contains the number of the caller. There is a similar feature in *getx25;* it writes the time and the login name into the same logfile. The command "set nolog" will turn off this feature.

set numlog
Like "set log", only better in some cases, since it sends only digits to the log file, and not other characters. The command "set nonumlog" will turn off this feature.

timeout NUMBER
Sets a global timeout value. Each expect uses time in the timeout reservoir; when this time is gone, the program gives up (exit 1). If this command isn't used, there is no global timeout. Also, the global timeout can be reset any time, and a value of 0 turns it off.

exit NUMBER
Exits with this value. 0 is success, anything else is failure.

You can test configuration files, sort of, by running
opx25 by hand, using the argument "-f" followed by the name of the script file. The program in this case sends to, and expects from, standard output and input, so you can type the input, observe the output, and see messages with the "echo" command. See the file **/usr/lib/uucp/X25/ventel.out** for a good example of Halgol programming.

SEE ALSO
getx25(1), uucp(1).

AUTHOR
Opx25 was developed by the Hewlett-Packard Company.

NAME

osck – check integrity of OS in SDF boot area(s)

SYNOPSIS

/etc/osck [**−v**] volume

Remarks:

Osck is implemented on the Series 500 only.

DESCRIPTION

Osck checks the operating system in the boot area on the volume specified by *volume* (a character special file).

The OSF must be the first section of an *n*-section operating system. If *n* is greater than one, *osck* prompts for additional volumes as needed. The volumes must be mounted in order.

The **−v** (verbose) option causes *osck* to print additional information about each volume and each code segment as they are encountered. If **−v** is not specified, it is silent except for warnings, errors, and prompts for new volumes.

Osck checks the following:

OSF headers are valid and consistent across multiple volumes;

the first code segment is a power-up segment;

the code segment chain contains correct headers and lengths;

all segment checksums are correct;

the system terminates correctly after the last segment.

SEE ALSO

oscp(1M), osmark(1M), osmgr(1M), sdfinit(1M).

DIAGNOSTICS

Osck gives an appropriate error message and returns a non-zero value if *volume* cannot be accessed or is not an SDF volume, there is no boot area, or the boot area contents appear invalid. Error messages are also given if any integrity violation is found. See *osmgr*(1M) for a complete list of return values.

NAME

oscp – copy, create, append to, split operating system

SYNOPSIS

/etc/oscp [**−o**] [**−v**] fromvolume tovolume
/etc/oscp **−m** [**−v**] file ... tovolume
/etc/oscp **−a** [**−v**] file ... tovolume
/etc/oscp **−s** [**−v**] fromvolume
/etc/oscp **−f** [**−v**] fromvolume tofile

Remarks:

Oscp is implemented on the Series 500 only.

DESCRIPTION

Oscp enables you to perform:

boot-to-boot copy
Copy an operating system from the boot areas on one or more SDF volumes to the boot area on one SDF volume;

files-to-boot copy (**−m**, **−a** options)
Create a new operating system or append to an existing operating system from a list of ordinary files, and put the resulting system in one boot area;

boot-to-files copy (**−s** option)
Split up the segments in an operating system from one or more boot areas to one or more ordinary files.

boot-to-file copy (**−f** option)
Split up the segments in an operating system from one or more boot areas to a single ordinary file.

Fromvolume and *tovolume* are usually character special files.

Boot-to-Boot Copy

If **−m**, **−a**, **−s**, and **−f** are not specified, *oscp* does boot-to-boot copy. For normal, multi-volume boot-to-boot copy, *oscp* requires that the OSF on the first *fromvolume* be the first section of an *n*-section operating system. If *n* is greater than one, *oscp* prompts you for additional volumes as required. The additional volumes must be mounted in order.

Before starting the copy, *oscp* clears the OSF header on *tovolume*. The OSF header values are corrected on *tovolume* after the copy is done. This new header may include a new system ID string that you enter when you are prompted (the same ID string displayed by the boot loader).

The **−o** (one volume only) option tells *oscp* to copy only one OSF (which may be part or all of a system) from *fromvolume* to *tovolume*, without changing the OSF header.

The **−v** (verbose) option tells *oscp* to print additional information about each volume as it is encountered. Otherwise, *oscp* is silent except for warnings, errors, and prompts for new volumes and new system ID strings.

Files-to-Boot Copy

If the **−m** (merge) option is given, *oscp* does a files-to-boot copy from the specified *files*. The source files may be BASIC/9000 BIN files or HP-UX ordinary files. The *files* must all be accessible and contain valid code segments. The code segments must all be of the same system type. The last code segment in each file must be followed by two null bytes.

Note that segments of unknown type, and old power-up segments (before February 1983) are "generic donors", and may be merged with any other type. Also note that, when creating a new system, *oscp* uses the first OSF header magic number in its internal list (i.e. 0xE9C28206).

Once you enter the new system ID string, *oscp* destroys the old OSF (if any) in the boot area before writing the new system.

The **–a** (append) option allows you to append code segments from ordinary files to an existing OSF on *tovolume*. There must be enough unused space in the boot area after the OSF, and the OSF must be a complete system in itself (i.e. volume 1 of 1). The existing OSF is not invalidated until the last segment is copied to the boot area.

In conjunction with **–m** or **–a**, the **–v** (verbose) option gives you additional information about the boot area and each segment as it is encountered.

Boot-to-Files Copy

The **–s** (split) option allows you to split an operating system into one or more ordinary files (HP-UX ordinary files only, not BASIC BIN files). For each code segment in the operating system, you are prompted for a file name to which the code segment is appended. If you enter a null line, the code segment is appended to the same file as was used in the previous append operation.

If the size of the specified file is greater than zero, *oscp* backs up two bytes from the end of the file to overwrite the previous terminator before appending the code segment to the file.

The **–v** (verbose) option gives you additional information about the boot area and each segment as it is encountered.

Note that the resulting ordinary files may be owned by the owner of the *oscp* command, depending on its permissions.

Boot-to-File Copy

The **–f** option allows you to split an operating system into a single ordinary file (*tofile*), eliminating any user interaction (except possibly to change certain types of media, if that is where the boot area is located). Otherwise, this option behaves exactly like the **–s** option.

Copying to Boot Areas

Before beginning the copy, *oscp* prompts you for the 80-character operating system ID string to use for all volumes.

Before writing to *tovolume*, *oscp* first checks that it contains a boot area with sufficient unused space.

SEE ALSO

osck(1M), osmark(1M), osmgr(1M), sdfinit(1M).

DIAGNOSTICS

Oscp prints an appropriate error message and returns a non-zero value if *fromvolume* or *tovolume* cannot be accessed or is not an SDF volume, there is no boot area, the boot area contents appear invalid, or the source OSF is not section 1 of an *n*-section system.

Errors are also given if:

fromvolume and *tovolume* are the same (by name);
*fromvolume*s are mounted out of order;
a specified ordinary file is inaccessible or has invalid contents;
the first segment is not a power-up segment;
any segment has a mismatching system type.

See *osmgr*(1M) for the exact list of return values.

BUGS

Oscp –a checks that all appended segments are mutually compatible, but it does not check them against the segments in the existing OSF.

Performing an *oscp –a* to a boot area with less than 1024 free bytes results in an error before the copy completes.

Before appending, *oscp –s* backspaces over the existing two-null-byte terminator at the end of each ordinary file, but it does not check that the bytes overwritten were actually two null bytes.

A boot area of less the 1024 bytes, at the end of a volume, results in a read error.

NAME

osmark – mark SDF volume boot area as loadable/non-loadable

SYNOPSIS

/etc/osmark [**−m** | **−u**] [**−v**] volume

Remarks:

Osmark is implemented on the Series 500 only.

DESCRIPTION

Osmark marks an operating system file (OSF) in a boot area as loadable (**−m** option) or non-loadable (**−u** option). *Volume* is usually a character special file specifying the SDF volume on which the boot area is found.

If neither **−m** nor **−u** are specified, *osmark* reports the status of the OSF.

The **−v** (verbose) option causes *osmark* to print additional information about the volume in the same format as that used by *osck* and *oscp.*

When dealing with a multi-volume operating system, be sure that each OSF in the system is properly marked, not just the first.

SEE ALSO

osck(1M), oscp(1M), osmgr(1M).

DIAGNOSTICS

Osmark outputs an appropriate error message and returns a non-zero value if *filespec* cannot be accessed or is not an SDF volume, there is no boot area, or the boot area contents appear invalid. Refer to *osmgr*(1M) for a list of possible return values.

NAME

osmgr – operating system manager package description

Remarks:

This entry describes the operating system manager package, which is implemented on the Series 500 only.

DESCRIPTION

This group of three commands helps you manage the operating systems which reside in the boot areas on your Structured Directory Format (SDF) volumes. The package includes:

oscp	copy systems or create them from ordinary files;
osck	check operating system integrity;
osmark	mark an operating system file as loadable or not loadable, or inquire about current state of operating system file.

Oscp, *osck*, and *osmark* are multiple links to a single program.

Boot Areas:

Each SDF volume has one boot area consisting of zero or more contiguous logical blocks. The boot area is completely outside the file area. Its size is determined when the volume is initialized. To change the size of a boot area, you must re-initialize the volume.

Each boot area may contain at most (one part of) one operating system.

The logical block size for a boot area is the same as that for the rest of the volume (i.e., whatever size you request when you initialize the volume).

Operating Systems:

Every HP 9000 operating system consists of a series of code segments. An operating system may reside in the boot area on one volume, or it may be distributed in sections over several volumes (not necessarily with a whole number of segments per volume).

An operating system can also reside in a number of ordinary files, each containing a whole number of segments, and terminated by two null bytes. This is the same format used for BASIC/9000 BIN files. In this form, the system is not loadable, but its files can be combined into a loadable system by *oscp*.

Operating System Files:

Each boot area contains zero or one operating system files (OSF's). If an operating system resides in sections in several boot areas, each section occupies one OSF on one SDF volume.

Operating System File Headers:

Each OSF starts with a header that includes a "loadable" flag, a volume number, and the total number of volumes over which this operating system is distributed. The loader only boots an OSF if it is marked loadable. If required, it requests additional volumes until it has loaded from all volumes in the set. You should ensure that all parts of a multi-volume operating system are marked loadable.

Each OSF header also includes an 80-character identification string. The loader displays this string before it starts to load from each volume.

RETURN VALUES

The following list contains all the possible return values, mnemonics, and meanings given by OS manager commands:

0		no error;
1	USAGE	bad argument list;
2	FILESYS	error during file system access;
3	VOLSEQ	volumes mounted out of order;

4	VOLCONT	bad volume (not SDF, no boot area, etc.);
5	HEADER	invalid or inconsistent OSF header(s);
6	FIRSTSEG	first segment is not a power-up segment;
7	SEGTYPE	incompatible segment system types or revisions;
8	SEGLEN	segment length out of range or not whole words;
9	CHECKSUM	segment checksum does not match reference value;
10	TERM	system terminator ("–1" word) missing.

SEE ALSO
oscк(1M), oscp(1M), osmark(1M), sdfinit(1M).

NAME

pdc – processor dependent code (firmware)

DESCRIPTION

Pdc is the firmware that implements all processor dependent functionality including initialization and self-test of the processor. Upon completion it loads and transfers control to the initial system loader (*isl*(1M)).

In order to load *isl* from an external medium, *pdc* must know the particular device on which *isl* resides. Typically the device is identified by the Primary Boot Path that is maintained by *pdc*. A *path* specification is a series of decimal numbers separated by periods that gives the various card and slot numbers and addresses. For the Model 840, the first number is the MID-BUS module number (i.e. slot number times four) and the next the CIO slot number. If the CIO slot contains an HP-IB card, the next number is the HP-IB address, followed by the unit number of the device if the device supports units. If the CIO slot contains a terminal card, the next number is the port number, which must be zero for the console.

When the processor is reset, after initialization and self-test are complete, *pdc* announces the Primary Boot, Alternate Boot, and Console Paths. If *autoboot* (see *isl*(1M)) is enabled then *pdc* gives a 10 second delay in which the operator may override the *autoboot* sequence by entering any character on the console. If the *autoboot* sequence is overriden or not enabled in the first place, *pdc* interactively prompts the operator for the Boot Path to use. Any required path components that are not supplied default to zero. The Primary Boot, Alternate Boot, and Console Paths and *autoboot* enable may be modified via *isl*.

SEE ALSO

boot(1M), hpuxboot(1M), isl(1M).

NAME

pwck, grpck – password/group file checkers

SYNOPSIS

/etc/pwck [file]
/etc/grpck [file]

DESCRIPTION

Pwck scans the password file and notes any inconsistencies. The checks include validation of the number of fields, login name, user ID, group ID, and whether the login directory and optional program name exist. The criteria for determining a valid login name are taken from *HP-UX System Administrator's Manual* for your system. The default password file is **/etc/passwd**.

Grpck verifies all entries in the group file. This verification includes a check of the number of fields, group name, group ID, and whether all login names appear in the password file. The default group file is **/etc/group**.

FILES

/etc/group
/etc/passwd

SEE ALSO

group(4), passwd(4).

DIAGNOSTICS

Group entries in **/etc/group** with no login names are flagged.

NAME

reboot – reboot the system

SYNOPSIS

/etc/reboot [**–h** | **–r**] [**–n** | **–s**] [**–d** device] [**–f** lif_filename] [**–t** time] [**–m** message]

DESCRIPTION

Reboot terminates all currently executing processes, except those essential to the system, then halts or reboots the system. *Reboot* with no arguments syncs all disks before rebooting the system. The options are:

–h	shutdown the system and halt.
–r	shutdown the system and reboot automatically. (default)
–n	do not sync the file systems before shutdown.
–s	sync the file systems before shutdown; for file systems which were cleanly mounted, modify the fs_clean flag from FS_OK to FS_CLEAN. (default)
–d *device*	reboot from the specified device. The device must be a *lif* volume. This option cannot be used with **-h**.
–f *lif_filename*	reboot from the specified file. If the filename is the NULL string, the power up search sequence will be made for a system. Otherwise, the filename has to follow the lif filename convention. This option cannot be used with **-h**.
–t *time*	the time when *reboot* will bring the system down. *Time* may be the word **now** (indicating immediate shutdown) or specify a future time in one of two formats: *+number and hour:min.* The first form brings the system down in *number* minutes and the second brings the system down at the time of day indicated (as a 24-hour clock).
–m *message*	At intervals that get closer together as reboot *time* approaches, *message* is displayed at the terminals of all users on the system.

At shutdown time a message is written in the file **/usr/adm/shutdownlog**(ifitexists), containing the time of shutdown, who ran *reboot*, and the reason.

Reboot can be executed only by the super-user.

HARDWARE DEPENDENCIES

Series 500
Reboot is not supported. A similar non-standard facility is provided for the Series 500 by *stopsys*(1M).

Series 800
The **-d** and **-f** options and the *device* and *lif_filename* parameters are ignored.

AUTHOR

Reboot was developed by HP, and the University of California, Berkeley.

FILES

/usr/adm/shutdownlog	shutdown log

SEE ALSO

lif(1), reboot(2).

NAME

reconfig – configure an HP-UX system

SYNOPSIS

/etc/reconfig [**–m**]

Remarks:

reconfig is implemented on the Series 300 only.

DESCRIPTION

Reconfig provides a means for system upgrade in the following areas:

- Operating system functionality,
- Adding new users to the system,
- Deleting existing users from the system,
- Adding support for remote-terminal access,
- Adding line printer support to the system,
- Setting up system user access.

Reconfig provides a useful, easy-to-use tool for upgrading and customizing HP-UX systems to match particular needs. Thus, upgrading an existing operating system to add new users or peripherals becomes a simple task.

Reconfig is a menu-driven command that is easy to use, even with little knowledge. Once a menu is displayed, the following rules apply:

Softkey Label	**Definition or Use**
HELP	Displays information describing your currently available options.
MAINMENU	Immediately exits the current menu and returns to the main menu. No configuration information associated with the current menu is processed, and any modified values are destroyed.
NEXT / PREVIOUS	Moves cursor to next or previous choice on multiple-choice action menu or field.
QUIT	Exits *reconfig*.
RESTORE	restores all fields in the current menu to their original (default) values before any values were changed.
SELECT	Initiates the configuration action associated with the current cursor position (highlighted field).

Many menus also prompt for inputs. Each input is terminated by pressing [RETURN] or by pressing one of the displayed softkeys. To select a displayed default value from the field associated with a given prompt, press [RETURN] without typing an entry.

Some fields offer multiple choices where all valid choices are displayed simultaneously and the default choice is displayed following the input prompt. Type the preferred choice and press [RETURN] or use the **NEXT** or **PREVIOUS** softkey to display the next/previous choice then press **SELECT** or [RETURN] when the correct option is reached.

When responding to prompts that expect a yes/no answer, use **y** or **n** or **yes** or **no** in any combination of uppercase or lowercase letters followed by [RETURN].

Some menus (such as adding users) cycle repeatedly so you can perform the operation more than once in succession. To exit such menus, press **MAINMENU**. Any completed configurations associated with the cycling menu are left intact. If **MAINMENU** is pressed before a cycle is completed, any partial configurations are abandoned without altering system configuration.

If the **−m** option is used, a special mode is used that does not produce escape-code sequences. This option is useful when using terminals that do not support standard HP terminal escape-code sequences; that is, terminals that are not officially supported by HP-UX.

This mode of operation is the same as normal operation with the following exceptions:

Menus offering a choice of actions to perform identify each action with an associated number or letter. To choose an option, press the number or letter (uppercase or lowercase), then press [**RETURN**].

The **QUIT**, **RESTORE**, **MAINMENU**, and **HELP** softkeys are replaced with the letters **q**, **r**, **m**, and **h**, respectively. To select one, press the correct letter followed by [**RETURN**].

When prompted for a choice in a multiple-choice field, type the corresponding number or letter or type in the choice itself, then press [**RETURN**].

When a new user is added to the system, the password file in */etc/passwd* and the group file in */etc/group* are both updated to reflect the addition of the new user. At the same time, a home directory is created in the */users* directory.

When an existing user is deleted from the system, the password file, in /etc/passwd, and the group file, in /etc/group, will both be updated to reflect the fact that the user no longer has access privileges to the system. The users home directory, and all files contained therein, will be deleted, if that option is specified.

When a new printer is added to the system, all required device nodes will automatically be made, and the line printer spooler will be notified that a new printer is now available for use.

When setting up user access to the system, you can choose whether or not a user login is required. If login is required, the system must operate in *init* state **2** (multi-user state) and each user must be known to the system (use the *reconfig* option for adding new users). If login is not required, the system must operate in *init* state **1** (single-user state), and the system automatically starts running PAM (Personal Applications Manager) after power-up.

When remote terminal support is added for a new port, the file /etc/inittab will be updated, so that the next time the system is powered up, a getty will automatically be started for the specified port. Any required device nodes will be created, if necessary. Only one getty is permitted to run on any particular port.

When the user wishes to modify his operating system, he is supplied with three options:

1. Generate a fully loaded operating system.
2. Generate a minimally loaded operating system.
3. Generate a custom kernel.

If a user is not concerned with the size of his operating system, or if the user needs to support a multitude of devices and I/O cards, then option (1) is a logical choice for him. If the user does not plan to support many optional devices on his system, then the minimal system, option (2), might work well for him. However, the majority of the users will be supporting a wide variety of devices and I/O cards on their systems. For these users, option (3) makes the most sense. This option allows the user to build a version of the operating system which supports only those devices and I/O cards specified; thus, this is the most space efficient of the three options.

FILES

/etc/master	master device table
/etc/conf/dfile	description file for current operating system
/hp-ux	current operating system

/etc/inittab	system initialization tables
/etc/passwd	system password file
/etc/group	system groups identification file
/users	system users identification file

SEE ALSO

config(1m)

NAME
revck – check internal revision numbers of HP-UX files

SYNOPSIS
/etc/revck ref_files

Remarks:

Not supported on the Integral Personal Computer.

DESCRIPTION
Revck checks the internal revision numbers of lists of files against reference lists. Each *ref_file* must contain a list of absolute path names (each beginning with "/") and *whatstrings* (revision information strings from *what*(1)). Path names begin in column one of a line, and have a colon appended to them. Each path name is followed by zero or more lines of *whatstrings*, one per line, each indented by at least one tab (this is the same format in which *what*(1) outputs its results).

For each path name, *revck* checks that the file exists, and that executing *what*(1) on the current path name produces results identical to the *whatstrings* in the reference file. Only the first 1024 bytes of *whatstrings* are checked.

Ref_files are usually the absolute path names of the *revlist* files shipped with HP-UX. Each HP-UX software product includes a file named /system/*product*/revlist (for example, /system/97070A/revlist). The *revlist* file for each product is a reference list for the ordinary files shipped with the product, plus any empty directories on which the product depends.

FILES
/system/*product*/revlist	lists of HP-UX files and revision numbers

SEE ALSO
what(1).

DIAGNOSTICS
Revck is silent except for reporting missing files or mismatches. If a *ref_file* is not in the right format, you will get unpredictable results.

NAME

rootmark – mark/unmark volume as HP-UX root volume

SYNOPSIS

/etc/rootmark [**−m** | **−u**] filespec

Remarks:

Rootmark is implemented on the Series 500 only.

DESCRIPTION

Rootmark enables you to control which mass storage device contains your HP-UX root (/) directory. The HP-UX operating system searches mass storage devices and uses the first root volume it finds.

Filespec is usually a character special file which points to a mass storage volume initialized with Structured Directory Format (SDF). If invoked with no option, *rootmark* tells the current state of the specified volume. If **−m** is specified, then the specified volume is marked as a root volume. If **−u** is specified, the specified volume is marked as not a root volume. *Rootmark* is silent if successful.

RETURN VALUE

Rootmark sends an error message to standard error and returns a non-zero value if it cannot read or write a volume, or if a volume is not SDF. *Rootmark* returns 1 for incorrect syntax, 2 for a file system problem, and 3 for a volume that is not in SDF.

EXAMPLE

The following example makes **/dev/rhd** usable as root; you must super-user to execute the example:

```
# rootmark /dev/rhd      # check if /dev/rhd is a root volume
/dev/rhd is marked as NOT a root volume.
# rootmark -m /dev/rhd  # mart it as the root volume
# rootmark /dev/rhd      # check results
/dev/rhd is marked as a root volume.
```

SEE ALSO

mount(1), osmgr(1M), sdfinit(1M).

WARNINGS

A volume must not be marked as a root volume unless it contains all the directories and files that HP-UX requires for system initialization.

Never mark any media shipped from Hewlett-Packard as not a root volume, in case you need to re-install HP-UX from that media.

NAME

runacct – run daily accounting

SYNOPSIS

/usr/lib/acct/runacct [mmdd [state]]

DESCRIPTION

Runacct is the main daily accounting shell procedure. It is normally initiated via *cron*(1M). *Runacct* processes connect, fee, disk, and process accounting files. It also prepares summary files for *prdaily* or billing purposes.

Runacct takes care not to damage active accounting files or summary files in the event of errors. It records its progress by writing descriptive diagnostic messages into **active**. When an error is detected, a message is written to **/dev/console**, mail (see *mail*(1)) is sent to **root** and **adm**, and *runacct* terminates. *Runacct* uses a series of lock files to protect against re-invocation. The files **lock** and **lock1** are used to prevent simultaneous invocation, and **lastdate** is used to prevent more than one invocation per day.

Runacct breaks its processing into separate, restartable *states* using **statefile** to remember the last *state* completed. It accomplishes this by writing the *state* name into **statefile**. *Runacct* then looks in **statefile** to see what it has done and to determine what to process next. *States* are executed in the following order:

SETUP	Move active accounting files into working files.
WTMPFIX	Verify integrity of **wtmp** file, correcting date changes if necessary.
CONNECT1	Produce connect session records in **ctmp.h** format.
CONNECT2	Convert **ctmp.h** records into **tacct.h** format.
PROCESS	Convert process accounting records into **tacct.h** format.
MERGE	Merge the connect and process accounting records.
FEES	Convert output of *chargefee* into **tacct.h** format and merge with connect and process accounting records.
DISK	Merge disk accounting records with connect, process, and fee accounting records.
MERGETACCT	Merge the daily total accounting records in **daytacct** with the summary total accounting records in **/usr/adm/acct/sum/tacct**.
CMS	Produce command summaries.
USEREXIT	Any installation-dependent accounting programs can be included here.
CLEANUP	Cleanup temporary files and exit.

To restart *runacct* after a failure, first check the **active** file for diagnostics, then fix up any corrupted data files such as **pacct** or **wtmp**. The **lock** files and **lastdate** file must be removed before *runacct* can be restarted. The argument *mmdd* is necessary if *runacct* is being restarted, and specifies the month and day for which *runacct* will rerun the accounting. Entry point for processing is based on the contents of **statefile**; to override this, include the desired *state* on the command line to designate where processing should begin.

EXAMPLES

To start *runacct*.

```
     nohup runacct 2> /usr/adm/acct/nite/fd2log &
```

To restart *runacct*.

```
     nohup runacct 0601 2>> /usr/adm/acct/nite/fd2log &
```

To restart *runacct* at a specific *state*.

```
nohup runacct 0601 MERGE 2>> /usr/adm/acct/nite/fd2log &
```

FILES

/usr/adm/acct/nite/active
/usr/src/cmd/acct/ctmp.h
/usr/adm/acct/nite/daytacct
/usr/adm/acct/nite/lastdate
/usr/adm/acct/nite/lock
/usr/adm/acct/nite/lock1
/usr/adm/pacct*
/usr/adm/acct/nite/ptacct*.*mmdd*
/usr/adm/acct/nite/statefile
/usr/src/cmd/acct/tacct.h
/etc/wtmp

SEE ALSO

acctcom(1), mail(1), acct(1M), acctcms(1M), acctcon(1M), acctmerg(1M), acctprc(1M), acctsh(1M), cron(1M), fwtmp(1M), acct(2), acct(4), utmp(4), System Accounting in the *HP-UX System Administrator's Manual.*

BUGS

Normally it is not a good idea to restart *runacct* in the **SETUP** *state*. Run **SETUP** manually and restart via:

runacct *mmdd* **WTMPFIX**

If *runacct* failed in the **PROCESS** *state*, remove the last **ptacct** file because it will not be complete.

NAME

savecore – save a core dump of the operating system

SYNOPSIS

/etc/savecore [**−n**] [**−v**] dirname [system]

DESCRIPTION

Savecore is meant to be called near the end of the /etc/rc file. Its function is to save the core dump of the system (assuming one was made when the system crashed) and to write a reboot message in the shutdown log.

Savecore checks the core dump to be certain it corresponds with the current running system. If it does it saves the core image in the file *dirname*/hp-core.n and a copy of the current running system file, which contains the namelist, in the file *dirname*/hp-ux.n. The trailing ".n" in the pathnames is replaced by a number that grows every time *savecore* is run in that directory. (This number is kept in the file *dirname*/bounds, which is created if it does not already exist.)

Before *savecore* writes out a core image, it reads a number from the file *dirname*/minfree. The core dump is not done if the number of free 512-byte blocks on the file system that contains *dirname* is less than the number obtained from the minfree file. If the minfree file does not exist, *savecore* always writes out the core file (assuming that a core dump was taken). Note that repeated system crashes can result in multiple core files that use up large quantities of disk space (especially on machines with large physical memories).

Savecore also writes a reboot message in the shutdown log (if it already exists). If the system crashed as a result of a panic, *savecore* records the panic string in the shutdown log too.

If the core dump was from a system other than /hp-ux, the name of that system must be supplied as *system*.

If the −**n** option is specified, no copy of the current running system file is saved in *dirname*/hp-ux.n. Note that the user must now remember which system file, e.g. /hp-ux, corresponds to the saved core file. The core file by itself is not very useful.

If the −**v** option is specified, additional messages are printed under some conditions. This option is usually used only for debugging.

RETURNS

Savecore returns the following exit status values: A zero exit status indicates that a core dump was found and saved. An exit status of 1 indicates that a core dump could not be saved due to some error or minfree limitation. An exit status of 2 indicates that no core dump was found to save.

WARNINGS

Some implementations may place the core dump in the disk swap area while the system reboots. In such cases, *savecore* may not be able to recover the crash dump if too many programs have been swapped out before *savecore* is run.

AUTHOR

Savecore was developed by the Hewlett-Packard Company, and the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

FILES

/hp-ux	current system
/usr/adm/shutdownlog	shutdown log
dirname/bounds	crash dump number
dirname/minfree	minimum free blocks on file system

SEE ALSO

adb(1).

NAME
sdfdf – report number of free SDF disk blocks

SYNOPSIS
sdfdf device ...

DESCRIPTION
Sdfdf prints out the number of free blocks and free inodes available for SDF file systems by examining the counts kept in the super-blocks; *device* must be specified by device name.

AUTHOR
Sdfdf was developed by the Hewlett-Packard Company.

SEE ALSO
sdf(4), du(1), df(1M).

NAME

sdffsck – SDF file system consistency check, interactive repair

SYNOPSIS

sdffsck [**–y**] [**–n**] [**–s**] [**–d**] *SDFdevice* ...

DESCRIPTION

Sdffsck is intended to mimic the series 500 implementation of *fsck[SDF]*(1M).

Sdffsck checks and interactively repairs inconsistent conditions for SDF file systems. If the file system is consistent, then the number of files, the number of blocks used, the number of blocks free, and the percent of volume unused are reported. If the file system is inconsistent, the operator is prompted for concurrence before each correction is attempted. Note that many corrective actions will result in some loss of data. The amount and severity of the loss can be determined from the diagnostic output. The default action for each consistency correction is to wait for the operator to respond **yes** or **no**. If the operator does not have write permission, *sdffsck* defaults to **–n**.

Sdffsck makes multiple passes over the SDF file system, so care should be taken to ensure that the SDF device is quiescent.

The following flags are interpreted by *sdffsck*:

–y Assume a **yes** response to all questions asked.

–n Assume a **no** response to all questions asked; do not open the file system for writing.

–s Ignore the actual free list and unconditionally reconstruct a new one. This option is useful in correcting multiply claimed blocks when one of the claimants is the free list. When using this option, the number of unclaimed blocks reported by *sdffsck* includes all the blocks in the free map. This can produce extensive output if **–d** is also selected.

–s should only be selected after a previous *sdffsck* indicates a conflict between a file and the free map. After **sdffsck –s** has executed, the integrity of the conflicting file(s) should be checked.

–d Dump additional information. The more **d**'s that are present, the more information that is dumped. You may specify up to five **d**'s. Using more than two, however, can result in an overwhelming amount of output.

Sdffsck also recognizes, and ignores, the –S and **–t** options found in other versions of *fsck*. An appropriate warning is printed. The diagnostics are intended to be self-explanatory.

SDFdevice is a device file name describing the device on which the SDF file system to be checked resides (e.g., **/dev/rdsk/c1d1s4**).

Error messages from *sdffsck* are written to *stderr*. Information generated because of the **–d** option and normal output is written to *stdout*; both are unbuffered.

Inconsistencies checked include:

1. Blocks claimed by more than one inode, or by the free list;
2. Blocks claimed by an inode or the free list outside the range of the file system;
3. Incorrect link counts;
4. Blocks not accounted for anywhere;
5. Bad inode format;
6. Directory checks:
   Files pointing to unallocated inodes;
   Inode numbers out of range;

Multiply linked directories;
Link to the parent directory.

Orphaned files (allocated but unreferenced) with non-zero sizes are, with the operator's concurrence, reconnected by placing them in the **lost+found** directory on the SDF file system. The name assigned is the inode number. The only restriction is that **lost+found** must exist in the root of the SDF file system being checked, and must have empty slots in which entries can be made. This is accomplished by executing

sdfmkdir SDFdev:/lost+found

(using the name of the SDF device for *SDFdev*).

Orphaned directories and files with zero size are, with the operator's concurrence, returned directly to the free list. This will also happen if the **lost+found** directory does not exist.

WARNINGS

Sdffsck cannot check devices with a logical block size greater than 4096.

AUTHOR

Sdffsck was developed by HP.

SEE ALSO

sdf(4), fsck[SDF](1M), sdfmkdir(1),

Series 500 HP-UX System Administrator Manual.

NAME

sdffsdb – examine/modify an SDF file system

SYNOPSIS

sdffsdb SDFdevice

DESCRIPTION

Sdffdsb is intended to mimic the series 500 implementation of *fsdb[SDF]*(1M).

Sdffsdb provides you with the ability to perform the following functions on the specified *SDFdevice*:

1. Find the inode number of a file, given its full path name.
2. Examine and modify the contents of the superblock (volume header).
3. Examine and modify the contents of any inode or other file attribute.

Integer input to *sdffsdb* may be entered in decimal (default), octal (with a preceding "0"), or hexadecimal (with a preceding "0x").

SDFdevice is a raw or block special file describing the device on which the SDF file system is located.

Sdffsdb execution is interactive. Prompts consist of requests for the needed information. When execution begins, *sdffsdb* displays the following menu:

```
1 – find inode numbers.
2 – examine superblock.
3 – examine inodes.
q – quit.
```

after which you are requested to enter one of the options shown.

Typing **1** causes *sdffsdb* to accept full pathnames of files (relative to the door directory of the SDF file system); it returns the corresponding inode number. Typing **q** returns you to the main menu.

Typing **2** displays the contents of each record in the superblock. Each record is numbered. If a right parenthesis ")" follows the number, then the record can be modified. If a right curly bracket "}" follows the number, then the record cannot be modified. You are then asked whether or not you want to modify the superblock. An answer beginning with **n** sends you back to the menu; an answer beginning with **y** causes *sdffsdb* to ask for the record number to be modified. If the record number specified cannot be modified, you are told about it, and prompted for another record number. If you specify a record number which can be changed, you are prompted for the new data. Typing **q** returns you to the main menu.

Typing **3** causes *sdffsdb* to prompt you for a file attribute record number. Upon receipt of a valid number, the contents of that record are displayed, and you are prompted for the information you want to change. Parentheses and curly brackets have the same meanings as described above. Typing **q** returns you to the main menu.

Typing **q** at the main menu level terminates the *sdffsdb* command.

WARNINGS

Sdffsdb is deceptively easy to use, and therefore should be used with extreme care. Be sure you know what you are doing before you enter too deeply into options **2** or **3**. You are given the opportunity to abort any operation before you have changed anything (by typing **q**), so consider carefully what you are about to do before you do it. *Sdffsdb* does not provide an "undo" function and the changes you make are immediate.

Sdffsdb cannot examine devices with a logical block size greater than 4096.

AUTHOR

Sdffsdb was developed by the Hewlett-Packard Company.

SEE ALSO

sdf(4), fsck[SDF](1M), fsdb[SDF](1M).

NAME

sdfinit – initialize Structured Directory Format volume

SYNOPSIS

/etc/sdfinit [-i] pathname [blocksize [bootsize [interleave]]]

DESCRIPTION

Sdfinit initializes a Structured Directory Format (SDF) volume associated with a specified special file.

Pathname refers to a character or block special file which must be accessible and not mounted.

Blocksize specifies the number of bytes per logical block. It is rounded up, if necessary, to the next multiple of the physical record size for the volume. If absent or less than one (1), the system selects a reasonable default.

Bootsize specifies the number of bytes to be allocated for boot area on the volume. It is rounded up to an integer number of logical blocks. Default is zero (no boot area).

Interleave defines the sector interleave factor. Default is 1 (not necessarily the best value for all devices). In the special case of initializing memory volumes (those volumes accessed through driver number 10 as explained in the *HP-UX System Administrator Manual*), *interleave* specifies the number of 256-byte physical sectors that are to be used for the memory volume "device". The maximum number of sectors allowed is 2047 which yields 524 032 bytes.

The root directory on the newly-initialized volume is always owned by super-user and has permissions of 755.

The *-i* option inhibits certification, limiting the operation to initialization only which consists of writing a directory structure. This saves a considerable amount of time in most cases. However, the *-i* option is not recommended for most removable media, unless the media was recently certified in the same type of drive.

Sdfinit does not return until the operation is complete which may require considerable time. For example, certification can consume up to 47 minutes for an HP 7933 disk or up to 67 minutes on an HP 88140L (DC-600) cartridge tape. Initialization requires an additional one to five minutes.

Note that during this certification and initialization process, *sdfinit* monopolizes the interface select code so that no other devices can be accessed. This means that if the root device is using the same interface as the device being accessed by *sdfinit*, the root device is also inaccessible for the duration.

RETURNS

Appropriate error messages are given if the argument list is incorrect, pathname cannot be initialized or any other error occurs.

WARNINGS

The effective user ID must be zero (super-user). The disk must not be mounted.

AUTHOR

Sdfinit was developed by HP.

SEE ALSO

See section (7) device special file manual pages.

NAME

setmnt – establish mount table mnttab

SYNOPSIS

/etc/setmnt

DESCRIPTION

Setmnt creates the **/etc/mnttab** table (see *mnttab*(4)), which is needed for both the *mount*(1M) and *umount* commands. *Setmnt* reads standard input and creates a *mnttab* entry for each line. Input lines have the format:

filesys node

where *filesys* is the name of the file system's *special file* (e.g., "dsk/?s?") and *node* is the root name of that file system. Thus *filesys* and *node* become the first two strings in the *mnttab*(4) entry.

FILES

/etc/mnttab

SEE ALSO

devnm(1M), mount(1M), mnttab(4).

BUGS

Filesys and *node* are truncated to MNTLEN bytes.
Setmnt silently enforces an upper limit on the maximum number of *mnttab* entries.
It is unwise to use *setmnt* to create false entries for *mount*(1M) and *umount*.

NAME

setprivgrp – set special attributes for group

SYNOPSIS

setprivgrp –**g** | –**n** | group–name [privileges]

setprivgrp -f file

DESCRIPTION

Setprivgrp associates a group with a kernel capability. This allows subsetting of super–user like privileges for members of a particular group or groups. In the first form the first argument to *setprivgrp* is either a group name, –**g**, or –**n** which specifies a particular group, all groups or no groups respectively. The optional second and subsequent arguments are symbolic names indicating kernel capabilities. In the second form the –**f** option is used to specify a file, typically */etc/privgroup*, from which group capabilities are set. The group access privileges are changed to reflect the specified kernel capabilities.

RTPRIO
: gives access to the *rtprio*(2) system call for setting real–time priorities.

MLOCK
: gives access to the *plock*(2) system call for locking process text and data into memory, and the SHM_LOCK command used with *shmctl*(2) system call.

CHOWN
: gives access to the *chown*(2) system call.

Specifying no access privileges removes any privileges that may currently be assigned. Note that capabilities set by this command are not additive. If you wish to add a capability for a particular group, you need to respecify all capabilities that were already set for that group in addition to the new capability.

The file named using the -f option should contain one or more lines in the following format:

–**g** | –**n** | group–name [privileges]

Only the super user may use this command.

FILES

/etc/privgroup
/etc/group

ERRORS

Setprivgrp returns 1 if caller is not super user, and 2 if there is not enough table space to hold a new privileged group assignment.

AUTHOR

Setprivgrp was developed by the Hewlett-Packard Company.

SEE ALSO

getprivgrp(1), getprivgrp(2), privgrp(4), rtprio(2), plock(2), shmctl(2), chown(2).

NAME

shutdown – terminate all processing

SYNOPSIS

/etc/shutdown [**–h** | **–r**] [**–d** device] [**–f** lif_file] [grace]

DESCRIPTION

Shutdown is part of the HP-UX system operation procedures. Its primary function is to terminate all currently running processes in an orderly and cautious manner. *Shutdown* can be used to put the system in single-user mode for administrative purposes such as backup or file system consistency checks (see *fsck*(1M)), and to halt or reboot the system. The procedure is designed to interact with the operator, i.e., the person who invoked *shutdown*. *Shutdown* may instruct the operator to perform some specific tasks or to supply certain responses before execution can resume.

Shutdown goes through the following steps:

All file systems' super blocks are updated; see *sync*(1M). This must be done before rebooting the system to ensure file system integrity.

All users logged on the system are notified to log out by a broadcast message. The operator may display his/her own message at this time. Otherwise, a standard warning message is displayed.

All currently executing processes are terminated except those essential to the system or associated with the shutdown procedure.

All file systems are unmounted.

The next step depends on which of the following options are selected:

–h	Shutdown the system and halt.
–r	Shutdown the system and reboot automatically.
–d *device*	Reboot from the specified device. The device must be a *lif* volume. The **–d** option can only be used with the **–r** option.
–f *lif_file*	Reboot from the specified file. If the filename is the NULL string, the power-up search sequence will be made for a system. Otherwise, the filename has to follow the lif filename convention. The **–f** option can only be used with the **–r** option.
grace	*Grace* specifies, in seconds, a grace period for users to log off before shutting down. The default is 60 seconds. If *grace* is zero, *shutdown* runs more quickly, but gives users very little time to log out.

If neither **–r** or **–h** is specified, the system will be placed in *run-level s*; see *init*(1M).

RETURNS

The most common error diagnostic that will occur is *device busy*. This happens when a particular file system could not be unmounted; see *mount*(1M).

EXAMPLES

To immediately reboot the system and run HP–UX again:

shutdown -r 0

To halt the system in 5 minutes:

shutdown -h 300

To go to init *run–level s* in 10 minutes:

shutdown 600

HARDWARE DEPENDENCIES

Series 500, Series 800

The **-d** and **-f options** and *device* and *lif_file* parameters are not supported.

SEE ALSO

init(1M), killall(1M), mount(1M), reboot(1M), sync(1M).

NAME

stopsys – stop operating system with optional reboot

SYNOPSIS

/etc/stopsys [**–r**]

Remarks:

Stopsys is implemented on the Series 500 only.

DESCRIPTION

Stopsys dumps all system I/O buffers to mass storage volumes (i.e. performs a *sync*(1M)), and shuts down all virtual memory activity. Then, *stopsys* either stops the operating system so that the hardware may be powered down (no option), or it reboots the system (resets the machine's processor(s) to the power-on state) (**–r** option). The reboot (**–r**) option results in the activation of the system boot loader, almost exactly as if the power was just turned on, except that I/O cards are not power-cycled.

Just before it stops the system, *stopsys* writes a message to /dev/console indicating that the system is stopped and can be safely powered down.

Stopsys may be invoked only by the effective super-user. However, it may be made public by setting the set-user-ID bit and assigning ownership to *root.*

Stopsys does not ensure that the system is idle. If any user processes are running, the *sync*(1M) may be ineffective. You should execute *shutdown*(1M), or at least kill all non-essential processes, prior to running *stopsys.*

SEE ALSO

chsys(1M), killall(1M), shutdown(1M), sync(1M).

DIAGNOSTICS

Stopsys returns only if a non-fatal error occurs, in which case it writes a message to standard error and returns 1. Non-fatal errors include:

invocation with improper arguments;
invocation by other than the effective super-user;
any failure to stop the system, as long as the system is still usable.

If *stopsys* fails to stop the system for any reason, but the system is then not in a usable state, *stopsys* writes an error message to /dev/console and then attempts to reboot (if **–r** was specified). If **–r** was not specified, or if the reboot attempt fails, *stopsys* writes "system stopped" on /dev/console, and you must reboot the system yourself (using the power switch or the front panel).

Note that if the reboot fails it indicates a hardware problem with the HP 9000 Model 20 keyboard on select code 6, or the HP 9000 Model 30/40 system control module on select code 7.

BUGS

At this time, *stopsys* does not shut down Local Area Net (LAN) activity.

NAME

swapon – enable additional device for paging and swapping

SYNOPSIS

/etc/swapon –a
/etc/swapon name ...

DESCRIPTION

Swapon is used to enable additional devices on which paging and swapping are to take place. The system begins by swapping and paging on only a single device so that only one disk is required at bootstrap time. Calls to *swapon* normally occur in the system multi-user initialization file */etc/rc* making all swap devices available, so that the paging and swapping activity is interleaved across several devices.

Normally, the **–a** argument is given, causing all devices marked as "sw" swap devices in **/etc/checklist** to be made available.

The second form announces individual block devices to be used for paging and swapping. These block devices must have been setup at system configuration time. *Name* must specify a block special file.

WARNINGS

There is no way to stop paging and swapping on a device.

Exercise due caution when enabling swap space on a device that may be unmounted during system operation or removed from the system.

HARDWARE DEPENDENCIES

The HFS file system is implemented on Series 200 beginning with HP-UX Release 5.0, and on Series 300 and Series 800.

FILES

/dev/dsk/#s# Normal paging devices.

AUTHOR

Swapon was developed by the University of California, Berkeley.

SEE ALSO

swapon(2).

NAME

sync – update the super block

SYNOPSIS

sync

DESCRIPTION

Sync executes the *sync* system intrinsic. If the system is to be stopped, *sync* must be called to insure file system integrity. It will flush all previously unwritten system buffers out to disk, thus assuring that all file modifications up to that point will be saved. See *sync*(2) for details.

SEE ALSO

sync(2).

NAME

syncer – periodically sync for file system integrity

SYNOPSIS

/etc/syncer [seconds] [–d directory ...]

DESCRIPTION

Syncer is a program that periodically executes *sync*(2) at an interval determined by the input argument *seconds*. If *seconds* is not specified, the default interval is every 30 seconds. This ensures that the file system is fairly up-to-date in case of a crash. This command should not be executed directly, but should be executed at system boot time via */etc/rc*, which is invoked at boot time via */etc/inittab*.

The –d option is used to open directories for cache benefit. All directories must be specified by their full pathname. If the –d option is not used, no directories will be opened.

AUTHOR

Syncer was developed by the Hewlett-Packard Company, and the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

brc(1M), init(1M), sync(1), sync(2).

NAME

sysdiag – online diagnostic system interface

SYNOPSIS

sysdiag [filename]

DESCRIPTION

Sysdiag is the command interpreter for the online diagnostic system. Its primary role is to provide a common user interface to all of the online diagnostic programs. The set of commands understood by *sysdiag* is listed below. For a complete description of each command see the *Hardware Support Documentation Set. Sysdiag* accepts commands from either standard input or the specified *filename.*

The online diagnostic system allows the user to diagnose the computer system hardware without placing the system into single-user mode. Certain restrictions apply to running diagnostics in an online environment. These restrictions are necessary to protect user data. Each diagnostic program defines which operations destroy data and which do not. Typically, those operations that destroy data cannot be run in a multi-user environment. For further information, see the reference manual for each of the diagnostic programs.

Command Summary

abort	Abort an active diagnostic program.
ci or !	Fork and exec a shell.
exit	Exit *sysdiag.*
hardcopy	Produce a hardcopy of all *sysdiag* and diagnostic program input and output.
help or ?	Provide online help for *sysdiag* and diagnostic programs.
install	Add a diagnostic program to the diagnostic system.
list	List the installed diagnostic programs.
purge	Remove a diagnostic program from the diagnostic system.
redo	Edit and execute a previous command.
resume	Restart execution of a diagnostic program.
run	Begin execution of a diagnostic program.
showactive	Show active diagnostic programs.
suspend	Suspend execution of a diagnostic program.
use	Redirect standard input for *sysdiag.*
wait	Wait for background diagnostic programs to terminate.

HARDWARE DEPENDENCIES

Series 200, Series 300, and Series 500

The online diagnostic system is not supported on these systems.

AUTHOR

Sysdiag was developed by HP.

FILES

/usr/diag/bin/*	diagnostic programs
/usr/diag/install/*	installation information
/usr/diag/security	access list for diagnostic users
/usr/diag/cat/*	native language support catalogs
/dev/diag/*	diagnostic special files

SEE ALSO

Hardware Support Documentation Set, Volumes 4 & 5

NAME

sysrm – remove optional HP-UX products

SYNOPSIS

sysrm product ...

DESCRIPTION

Sysrm is used to remove an optional product from an HP-UX system, usually to increase the amount of available disk space.

Only the super-user can execute *sysrm.*

product is the name or number of the product to be removed. This can be found under the directory */etc/filesets* where all products currently on the system can be found. The system administrator should use *sysrm* whenever trying to recover mass storage space.

FILES

update(1M).

WARNING

The system should be in single-user mode while sysrm is in progress. This means the system administrator should ensure that only "init", "sh" and any other mandatory processes are active.

NAME

tic – terminfo compiler

SYNOPSIS

tic [**–v**[*n*]] file ...

DESCRIPTION

Tic translates terminfo files from the source format into the compiled format. The results are placed in the directory **/usr/lib/terminfo**.

The **–v** (verbose) option causes *tic* to output trace information showing its progress. If the optional integer is appended, the level of verbosity can be increased.

Tic compiles all terminfo descriptions in the given files. When a **use=** field is discovered, *tic* searches first the current file, then the master file, which is "./terminfo.src".

If the environment variable TERMINFO is set, the results are placed there instead of **/usr/lib/terminfo**.

Some limitations: total compiled entries cannot exceed 4096 bytes. The name field cannot exceed 128 bytes.

FILES

/usr/lib/terminfo/?/*	compiled terminal capability data base

SEE ALSO

untic(1M), curses(3X), terminfo(4).

BUGS

Instead of searching **./terminfo.src**, it should check for an existing compiled entry.

NAME

tunefs – tune up an existing file system

SYNOPSIS

/etc/tunefs tuneup-options special

DESCRIPTION

Tunefs is designed to change a file system's dynamic parameters that affect the layout policies. The parameters that are to be changed are indicated by the flags given below:

–a *maxcontig*	This specifies the maximum number of contiguous blocks that will be laid out before forcing a rotational delay (see **–d** below). The default value is one, since most device drivers require one interrupt per disk transfer. Device drivers that can chain several buffers together in a single transfer should set this to the maximum chain length.
–d *rotdelay*	This specifies the expected time (in milliseconds) to service a transfer completion interrupt and initiate a new transfer on the same disk. It is used to decide how much rotational spacing to place between successive blocks in a file.
–e *maxbpg*	This indicates the maximum number of blocks any single file can allocate out of a cylinder group before it is forced to begin allocating blocks from another cylinder group. Typically this value is set to about one quarter of the total blocks in a cylinder group. The intent is to prevent any single file from using up all the blocks in a single cylinder group, thus degrading access times for all files subsequently allocated in that cylinder group. The effect of this limit is to cause big files to do long seeks more frequently than if they were allowed to allocate all the blocks in a cylinder group before seeking elsewhere. For file systems with exclusively large files, this parameter should be set higher.
–m *minfree*	This value specifies the percentage of space held back from normal users, i.e., the minimum free space threshold. The default value used is 10%. This value can be set to zero; if it is, up to a factor of three in throughput will be lost over the performance obtained at a 10% threshold. Note that if the value is raised above the current usage level, users will be unable to allocate files until enough files have been deleted to get under the higher threshold.
–A	This option specifies that redundant super-blocks, as well as the super-block, are to be modified as indicated above.
special	This is the name of the file system that will be tuned. It is either a block or character special file for an unmounted volume or volume section.

HARDWARE DEPENDENCIES

The HFS file system is implemented on Series 200 beginning with HP-UX Release 5.0, and on Series 300 and Series 800.

WARNINGS

This program should work on mounted and active file systems. Because the super-block is not kept in the buffer cache, the program will only take effect if it is run on dismounted file systems. If run on the root file system, the system must be rebooted.

Remember: You can tune a file system, but you can't tune a fish.

AUTHOR

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

SEE ALSO

mkfs[HFS](1M), newfs[HFS](1M), fs[HFS](4).

NAME

uconfig – system reconfiguration

SYNOPSIS

/etc/uconfig [option boot_device]

Remarks:

Uconfig is implemented on the Series 500 only.

DESCRIPTION

Uconfig enables you to reconfigure certain system parameters. When invoked with no arguments, *uconfig* lists the current system configuration. The following *options* are recognized:

–f file reconfigures the system parameters in the boot area according to the specifications given in *file*. *File* may contain any combination of system parameters. Each line in *file* has the following format:

id value [#comment]

where *id* is a pre-defined system parameter name, *value* is one or more values associated with the parameter, and *comment* is a descriptive comment for that line. All characters between the comment delimiter (#) and a new-line are ignored. The *id*, *value*, and *comment* fields are delimited by one or more blanks and/or tabs.

The valid *id*s and *value*s are:

vm_device driver_name addr1 addr2 addr3 addr4
where *driver_name* is an integer specifying the virtual device driver, and *addr1* through *addr4* are integers specifying the device select code, HP-IB address, unit, and volume, respectively.

cache_buf_size size
where *size* is an integer in the range 256 to 524 288, specifying the number of bytes in each individual cache buffer. *Size* is rounded down to the closest multiple of 256.

cache_buf_num num
where *num* is an integer in the range 1 to (maximum memory) divided by (minimum size of cache buffers), specifying the number of individual cache buffers forming the cache.

read_ahead_level level
where *level* is an integer in the range 1 to the value of **cache_buf_num**, specifying the number of buffers that can be filled in one sequential read operation.

swap_time time
where *time* is an integer in the range of 1 to 32 767 ticks (a tick equals 10msec), specifying the time a virtual segment remains memory resident before being swapped to disc.

page_size size
where *size* is an integer in the range 512 to 8 192, specifying the size of paged data in bytes. If *size* is an odd number, it is rounded down to the next even number.

page_swap_time time
where *time* is an integer in the range 1 to 32 767 ticks (a tick equals 10 msecs), specifying the time a page remains memory resident before being swapped to disc.

vm_pool_size size
where *size* is an integer in the range 16 384 to maximum memory, specifying the

maximum size in bytes of the virtual memory page pool.

scroll_pages num_pages
where *num_pages* is an integer in the range 1 to 10, specifying the number of pages of display buffering (one page = 24 lines of display). The actual number of pages allocated depends on current available memory. This parameter applies to the Model 520 only.

max_proc_per_usr max_user_process
where *max_user_process* is an integer specifying the maximum number of processes a single user can have.

stack_size size
where *size* is an integer in the range 16 384 to maximum memory, specifying the maximum stack size in bytes for any partition.

interactive_time time
where *time* is an integer in the range 1 to 32 767 ticks (a tick equals 10 msecs), specifying the amount of CPU time a process can consume after an interactive terminal read before it is no longer favored as interactive.

max_num_msgids num_ids
where *num_ids* is an integer in the range 5 to 1000, specifying the maximum number of message queue identifiers. *Num_ids* is rounded down to the closest multiple of 5.

max_msg_size size
where *size* is an integer in the range 256 to either 65 536 or **max_msg_qbytes**, whichever is less, specifying the maximum size in bytes of any one message.

max_msg_qbytes size
where *size* is an integer in the range 256 to either 65 536 or **max_msg_space**, whichever is less, specifying the maximum size in bytes of any one message queue.

max_msg_space size
where *size* is an integer in the range 256 to 523 264, specifying the maximum size in bytes the sum of all messages on all message queues.

max_num_semids num_ids
where *num_ids* is an integer in the range 5 to 1000, specifying the maximum number of semaphore identifiers. *Num_ids* is rounded down to the closest multiple of 5.

max_num_shmids num_ids
where *num_ids* is an integer in the range 5 to 1000, specifying the maximum number of shared memory identifiers. *Num_ids* is rounded down to the closest multiple of 5.

max_num_shm_segs segs
where *segs* is an integer in the range 0 to 1000, specifying the maximum number of shared memory segment attaches per process.

max_shm_vsegsz size
where *size* is an integer in the range 0 to 523 264, specifying the upper size limit of normal virtual shared memory segments in bytes. Requests for shared memory segment sizes larger than this value will result in paged virtual shared memory segments.

work_set_ratio ratio
where *ratio* is a floating-point number in the range 0 to 1, specifying the

minimum virtual memory working set ratio.

−d reconfigures the system parameters in the boot area to their default values. The default values, as contained in the file **/etc/uconfigtab**, are:

vm_device	0 0 0 0 0; root device as determined by the system at power-up;
cache_buf_size	1 024 bytes;
cache_buf_num	0; this value is dynamically computed;
read_ahead_level	0; this value is dynamically computed;
swap_time	0; this value is dynamically computed;
page_size	1 024 bytes;
page_swap_time	50 ticks; (one tick = 10 msecs);
vm_pool_size	0; this value is dynamically computed;
scroll_pages	2;
max_proc_per_usr	500;
stack_size	0; this value is dynamically computed;
interactive_time	300 ticks; (one tick = 10 msecs);
max_num_msgids	100;
max_msg_size	8 192 bytes;
max_msg_qbytes	16 384 bytes;
max_msg_space	32 768 bytes;
max_num_semids	100;
max_num_shmids	100;
max_num_shm_segs	10;
max_shm_vsegsz	16 384 bytes;
work_set_ratio	0.002.

The −**f** and −**d** options are mutually exclusive.

Boot_device is the path name of a character special file containing a boot area. The new configuration is written out to the boot area on boot_device, and takes effect the next time the system is booted.

FILES

/etc/uconfigtab	list of default system configuration parameters

WARNING

Do not use *uconfig* to change the system parameters of an operating system in a boot area unless that operating system is identical to the operating system you are currently running. If the two operating systems differ, *uconfig* will execute successfully, but the new operating system will either fail to boot, or, if it boots successfully, exhibit strange behavior.

NAME

untic – terminfo de-compiler

SYNOPSIS

untic [*term*] [–f *file*]

DESCRIPTION

Untic translates a terminfo file from the compiled format into the source format. If the environment variable TERMINFO is set to a path name, *untic* checks for a compiled terminfo description of the terminal under that path before checking **/usr/lib/terminfo**. Otherwise, only **/usr/lib/terminfo** is checked.

Normally *untic* uses the terminal type obtained from the TERM environment variable. With the *term* (terminal type) option, however, the user can specify the terminal type used.

With the *file* option the user can specify the file used for translation. This option bypasses the use of the TERM and TERMINFO environment variables.

Untic sends the de-compiled terminfo description result to standard output.

AUTHOR

Untic was developed by HP.

FILES

/usr/lib/terminfo/?/*	compiled terminal capability data base

SEE ALSO

tic(1M), curses(3X), terminfo(4).

NAME

update – update optional HP-UX products

SYNOPSIS

update [-m]

DESCRIPTION

Update is used to receive a periodic update of an HP-UX system or optional product.

Only the super-user can execute *update*.

The **m** option is used to turn off the menu mode in update. This should be used if the system console is not an HP **supported** terminal.

The update process is interactive. It prints information about the addresses of the mass storage devices to be used, and gives the user choices to change them. After the proper addresses are set and the user has selected the choice to read the table of contents on the update source media, update will print loading options for the products distributed on the update source media. The user then must select which optional products are to be loaded. There is also a choice to load all optional products which can be used to cut down on time delays for user interaction.

When the user is finished loading the products desired then the option to exit the process is used. This will terminate the update process and reboot the system. If any customize scripts are needed then they will be run automatically after the reboot. If any of these scripts are executed then the system administrator must change init states to execute the proper inittab file.

The reader is referred to the section in the System Administrators Manual describing the update process for an indepth explanation of how to use the process.

FILES

mknod(2), sysrm(1M).

WARNING

The system **must** be in single-user mode while the update is in progress. This means the system administrator should insure that only "init", "sh" and any other mandatory processes are active. The system administrator should also have a recent backup of the system before performing an update.

BUGS

There is no way to prevent accidentally updating an older version of a product over a newer one.

SPECIAL NOTE

Hewlett-Packard Company supports only those terminals in the *terminfo* data base that are included in the current list of supported devices for the HP-UX release being used. Other terminal model entries may be included in the *terminfo* data base that are not officially supported. If you choose to use such devices, they may or may not work correctly.

NAME

uucico – uucp copy in and copy out

SYNOPSIS

/usr/lib/uucp/uucico [**–r1**] [–ssys] [–xnum]

DESCRIPTION

Uucico scans the */usr/spool* directory for work files. If such files exist, a connection to a remove system is attempted using the line protocol for the remote system specified in the **L.sys** file. *Uucico* then executes all requests for work and logs the results.

The options are as follows:

–r1	Start *uucico* in the MASTER mode; The default is SLAVE mode.
–s*sys*	Do work only for the system specified by *sys*. If there is no work for *sys* on the local spool directory, initiate a connection to *sys* to determine if *sys* has work for the local system.
–x*num*	Use debugging option. *Num* is an integer in the range 1 – 9. More debugging information is given for larger values of *num*.

Uucico is usually started by a local program (e.g., *cron*(1M), *uucp*(1), or *uuxqt*(1M). It should *only* be directly initiated by a user when debugging.

When started by a local program, *uucico* is considered the MASTER and attempts a connection to a remote system. If *uucico* is started by a remote system, it is considered to be in SLAVE mode.

For the *uucico* connection to a remote system to be successful, there must be an entry in the */etc/passwd* file on the remote system of the form:

```
uucp::5:5::/usr/spool/uucppublic:/usr/lib/uucp/uucico
```

NAME

uuclean – uucp spool directory clean-up

SYNOPSIS

/usr/lib/uucp/uuclean [options]

DESCRIPTION

Uuclean will scan the spool directory for files with the specified prefix and delete all those which are older than the specified number of hours.

The following options are available.

−d*directory*	Clean *directory* instead of the spool directory. If *directory* is not a valid spool directory it cannot contain "work files" i.e., files whose names start with "C.". These files have special meaning to **uuclean** pertaining to **uucp** job statistics.
−p*pre*	Scan for files with *pre* as the file prefix. Up to 10 **−p** arguments may be specified. A **−p** without any *pre* following will cause all files older than the specified time to be deleted.
−n*time*	Files whose age is more than *time* hours will be deleted if the prefix test is satisfied. (default time is 72 hours)
−w*file*	The default action for *uuclean* is to remove files which are older than a specified time (see **−n** option). The **−w** option is used to find those files older than *time* hours, however, the files are not deleted. If the argument *file* is present the warning is placed in *file*, otherwise, the warnings will go to the standard output.
−s*sys*	Only files destined for system *sys* are examined. Up to 10 **−s** arguments may be specified.
−m*file*	The **−m** option sends mail to the owner of the file when it is deleted. If a *file* is specified then an entry is placed in *file*.

This program is typically started by *cron*(1M).

FILES

/usr/lib/uucp	directory with commands used by *uuclean* internally
/usr/spool/uucp	spool directory

SEE ALSO

cron(1M), uucp(1), uux(1).

NAME

uuls – list spooled uucp transactions grouped by transaction

SYNOPSIS

uuls [**–m**] [*directories...*]
uuls –s [**–m**] [*directories...*]
uuls –k [**–m**] [*directories...*]

DESCRIPTION

This command lists the contents of uucp spool directories (default "/usr/spool/uucp") with the files grouped into three categories:

Transactions

Each line starts with a transaction control filename and includes the name of each local (same-directory) subfile referenced by the control file (see below). Each is possibly followed by the total size in bytes (**–s** option) or Kbytes (**–k** option) in the transaction (see below). The **–m** (meanings) option replaces the subfile names with nodename, user, and commandline information (see below).

Orphans

All subfiles not referenced by any control file.

Others

All other files in the directory (all files not listed under one of the above categories).

Filenames are columnated so there may be more than one file per line. If a transaction has more subfiles than fit on one line, it is followed by continuation lines which are indented further.

The **–s** (size in bytes) and **–k** (Kbytes) options cause the command to follow each transaction in the **Transactions** section with a total size for all stat-able, sendable files in that transaction. This includes "D.*" files only, not "C.*" or "X.*" files. It does include stat-able files outside the spool directory which are indirectly referenced by "C.*" files. Sizes are either in bytes or rounded to the nearest Kbyte (1024 bytes), respectively. A totals line is also added at the end of the **Transactions** section.

The **–m** (meanings) option causes the command to follow "C.*" and "X.*" files with a *"nodename!username commandline"* line, instead of subfilenames. For "C" files, one line is printed per remote execution ("D*X*") subfile it references. *Nodename* is truncated at seven characters, *username* at eight, and *commandline* at however much fits on one line.

If **–m** is given, for each "C" file with no remote execution files, the command instead shows the meaning of the "C" file itself on one or more lines. Each line consists of a username, then "R" (receive) or "S" (send), then the name of the file to be transferred. See below for details.

Filenames are listed in alphabetical order within each section, except that the first section is only sorted by the control filename. Every file in the directory except "." and ".." appears exactly once in the entire list, unless **–m** is used.

Details

Transaction files are those whose names start with "C." or "X.". Subfilenames, which usually start with "D.", are gleaned from control file lines, at most one per line, from blank-separated fields, as follows:

```
C.*:  R <remotefrom> <localto> <user> -<options>
C.*:  S <localfrom> <remoteto> <user> -<options> <subfile> <mode>
X.*:  F <subfile>
```

Lines that don't begin with the appropriate character ('R', 'S', or 'F') are ignored.

In the "R" (receive) case, <remotefrom> is used to print the "C"-file meaning, and its transaction size is taken as zero (unknown).

In the "S" (send) case, if <subfile> is "D.0", <localfrom> is a file not in the spool directory, resulting from a typical **uucp** call without the −**C** (copy) option. In this case <localfrom> is used for the transaction size, if stat-able, and to print the "C"-file meaning.

uucp −**C** and **uux** both set <subfile> to a true (spooled) subfile name.

Orphan files are those whose names start with "D." and which are not referenced by any control files.

This algorithm extracts from control files the names of all subfiles which should exist in the spool directory when the transaction is not being actively processed. It is not unusual to see "missing subfiles" and "orphans" if you **uuls** a spool directory while **uucico**, **uucp**, **uux**, or **uuxqt** is active.

Meanings information is gotten by reading each "D*X*" subfile referenced by each "C.*" file, and by reading "X*X*" files. *Nodename*!*username* is taken from the last line in the file which is of the form:

U <username> <nodename>

Likewise, *commandline* is taken from the last line of the form:

C <commandline>

If a subfile name is referenced more than once, references after the first show the subfile as missing. If a subfile name appears in a (corrupt) directory more than once, the name is only found once, but then it is listed again under **Orphans** .

AUTHOR

Uuls was developed by the Hewlett-Packard Company.

SEE ALSO

mail(1), uucp(1), uuto(1), uux(1), uuxqt(1M), stat(2).

DIAGNOSTICS

The program writes an appropriate message to standard error if it has any problems dealing with a specified file (directory), including failure to get heap space. It always returns zero as its exit value.

If a control file is unopenable (wrong permissions or it disappeared while **uuls** was running), its name is preceded by a "*" and the size of the transaction is zero. If a subfile is missing (filename not found in the directory being listed) or un-stat-able (if required for −**s** or −**k**), its name is preceded by a "*" and it contributes zero bytes to the size of the transaction.

If −**m** is specified and a "D*X*" file is missing or unreadable, its name is given with a "*" prepended, as usual.

BUGS

This command uses *chdir*(2) to change to each directory in turn. If more than one is specified, the second through last directories must be absolute (not relative) pathnames, or the chdir() may fail.

NAME

uusnap – show snapshot of the UUCP system

SYNOPSIS

uusnap

DESCRIPTION

Uusnap displays in tabular format a synopsis of the current UUCP situation. The format of each line is as follows:

```
site   N Cmds   N Data   N Xqts   Message
```

Where "site" is the name of the site with work, "N" is a count of each of the three possible types of work (command, data, or remote execute), and "Message" is the current status message for that site as found in the STST file.

Included in "Message" may be the time left before UUCP can re-try the call, and the count of the number of times that UUCP has tried to reach the site. The process id of UUCICO may also be shown if it is in a TALKING state.

AUTHOR

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

SEE ALSO

uucp(1).

NAME

uusub – monitor uucp network

SYNOPSIS

/usr/lib/uucp/uusub [options]

DESCRIPTION

Uusub defines a *uucp* subnetwork and monitors the connection and traffic among the members of the subnetwork. The following options are available:

–a*sys*	Add *sys* to the subnetwork.
–d*sys*	Delete *sys* from the subnetwork.
–l	Report the statistics on connections.
–r	Report the statistics on traffic amount.
–f	Flush the connection statistics.
–u*hr*	Gather the traffic statistics over the past *hr* hours.
–c*sys*	Exercise the connection to the system *sys*. If *sys* is specified as **all**, exercise the connection to all the systems in the subnetwork.

The connections report is formatted as follows:

sys #call #ok time #dev #login #nack #other

Format interpretation:

sys	remote system name,
#call	number of times the local system tried to call *sys* since the last flush was done,
#ok	number of successful connections,
time	latest successful connect time,
#dev	number of unsuccessful connections because of no available device (e.g., ACU),
#login	number of unsuccessful connections because of login failure,
#nack	number of unsuccessful connections because of no response (e.g. line busy, system down),
#other	number of unsuccessful connections because of other reasons.

Traffic statistics are reported as follows:

sfile sbyte rfile rbyte

Format interpretation:

sfile	number of files sent,
sbyte	number of bytes sent over the period of time indicated in the latest *uusub* command with the −**u***hr* option,
rfile	number of files received,
rbyte	number of bytes received.

The command:

```
uusub −c all −u 24
```

is typically started by *cron*(1M) once a day.

FILES

/usr/lib/uucp/L_sub	connection statistics
/usr/lib/uucp/R_sub	traffic statistics
/usr/spool/uucp/SYSLOG	system log file

SEE ALSO

uucp(1), uustat(1).

NAME

uuxqt – uucp command execution

SYNOPSIS

/usr/lib/uucp/uuxqt [–xnum]

DESCRIPTION

The ***uuxqt*** daemon performs local command execution of execution files (**X.***) on the */usr/spool/uucp* directory. *Uux* generates work files with an execution (X) grade which become execution files when transferred to the remote system. The command requested by the execution file is checked against the list of remotely executable commands in the COMMANDS file. The USERFILE is then searched to find the first NULL system field for path access permission.

The option –**x***num* is a parameter specifying debugging information. *Num* is an integer in the range 1 – 9. The amount of debugging information increases as the value of *num* increases.

NAME

vtdaemon – respond to vt requests

SYNOPSIS

vtdaemon [**–g**[**<ngateway>**]] [**–n**] <lan device> <lan device> ...

DESCRIPTION

vtdaemon responds to requests from other systems (via local area network) made by *vt(1). vtdaemon* will spawn a server to respond to each request that it receives.

The **–g** option causes *vtdaemon* to rebroadcast all requests received on one lan device to all of the other lan devices specified on the command line. The optional parameter *ngateway* specifies the maximum number of *vtgateway* servers that can be in operation concurrently. If *ngateway* is not specified then there will be no limit on the number of vtgateway servers that can be in operation concurrently.

The **–n** option causes *vtdaemon* to ignore all requests that have come through a gateway.

The remaining arguments are the full path names of lan devices that *vtdaemon* will look for requests on. If no lan devices are specified then the default lan device is used. The major number for this device must correspond to a CIO IEEE802.3 local area network device.

Another function of *vtdaemon* is to create *portals* and service portal requests. A *portal* is a callout device that can be used by *uucico(1M)* to communicate to another machine via local area network. Portals are created by *vtdaemon* according to the configuration information found in the file **/usr/lib/uucp/L-vtdevices.** Each line in L-vtdevices has the format:

> <calldev>[,<lan device>] <nodename>

For each line, *vtdaemon* will create a portal named *calldev* in **/dev.** Whenever this device is opened, *vtdaemon* will will spawn a server that will create a connection to the system specified by *nodename* via the lan device specified. If no lan device is specified then the first one specified on the command line when *vtdaemon* was started is used (or the default lan device is used if no lan devices were specified on the command line).

vtdaemon should be terminated by sending signal **SIGTERM** to it. When *vtdaemon* receives this signal it will remove all of the portals it created in **/dev** before exiting.

HARDWARE DEPENDENCIES

Series 500:

> The HP 2285A lan device is not supported by *vtdaemon.*

Series 200/300:

> The **–g** option is not supported.

FILES

/usr/contrib/lib/vtdaemonlog	logfile used by *vtdaemon.*
/dev/ieee	default lan device name.

SEE ALSO

uucico(1M), vt(1)

DIAGNOSTICS

Diagnostics messages produced by *vtdaemon* are written to /usr/contrib/lib/vtdaemonlog.

WARNINGS

vtdaemon uses the Hewlett-Packard **LLA** (Link Level Access) direct interface to the HP network drivers. *vtdaemon* uses the multicast address **0x01AABBCCBBAA.** It should not be used or deleted by other applications accessing the network. *vtdaemon* uses the following **IEEE 802.3** *sap* (service access point) values: **0x90, 0x94, 0x98, 0x9C, 0xA0, 0xA4, 0xA8, 0xAC,**

0xB0, 0xB4, 0xB8, 0xBC, 0xC0, 0xC4, 0xC8, 0xCC, 0xD0 and 0xD4. They should not be used by other applications accessing the network.

NAME

wall – write to all users

SYNOPSIS

/etc/wall **[-g** groupname] [file]

DESCRIPTION

When *wall* is invoked without arguments, the standard input is read until an end-of-file. It then sends this message to all currently logged-in users preceded by:

Broadcast Message from ...

If the **–g** *groupname* option is specified, *wall* sends the message to all currently logged-in *groupname* members (as specified in **/etc/group**) preceded by:

Broadcast Message from ... to group *groupname*

If the *file* option is specified, *wall* uses *file* as its standard input.

It is used to warn all users, typically prior to shutting down the system.

The sender must be super-user to override any protections the users may have invoked (see *mesg*(1)).

Wall has timing delays, and will take at least 30 seconds to complete.

FILES

/dev/tty*

SEE ALSO

mesg(1), write(1).

DIAGNOSTICS

"Cannot send to ..." when the open on a user's tty file fails.

INTERNATIONAL SUPPORT

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

NAME

whodo – which users are doing what

SYNOPSIS

/etc/whodo

DESCRIPTION

Whodo produces merged, reformatted, and dated output from the *who*(1) and *ps*(1) commands.

FILES

/etc/passwd

SEE ALSO

ps(1), who(1).

INTERNATIONAL SUPPORT

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

NAME

intro – introduction to system calls

DESCRIPTION

This section describes all of the system calls. All of these calls return a function result. This result indicates the status of the call. Typically, a zero or positive result indicates that the call completed successfully, and –1 indicates an error. The individual descriptions specify the details. An error number is also made available in the external variable *errno* (see *errno*(2)). Note: Errno is not cleared on successful calls, so it should be tested only after an error has been indicated.

SEE ALSO

intro(3), errno(2), hier(5).

The introduction to this manual.

NAME

access – determine accessibility of a file

SYNOPSIS

int access (path, amode)
char *path;
int amode;

DESCRIPTION

Path points to a path name naming a file. *Access* checks the named file for accessibility according to the bit pattern contained in *amode*, using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID. The bit pattern contained in *amode* is constructed as follows:

04	read
02	write
01	execute (search)
00	check existence of file

RETURN VALUE

If the requested access is permitted, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

Access to the file is denied if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	Read, write, or execute (search) permission is requested for a null path name.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[EROFS]	Write access is requested for a file on a read-only file system.
[ETXTBSY]	Write access is requested for a pure procedure (shared text) file that is being executed.
[EACCES]	Permission bits of the file mode do not permit the requested access and the real user ID is not the super-user.
[EFAULT]	*Path* points outside the allocated address space for the process. The reliable detection of this error will be implementation dependent.
[ENAMETOOLONG]	The named file exceeds MAXPATHLEN characters.

The owner of a file has permission checked with respect to the "owner" read, write, and execute mode bits. Members of the file's group other than the owner have permissions checked with respect to the "group" mode bits, and all others have permissions checked with respect to the "other" mode bits.

Access will report that a file currently open for execution is not writable, regardless of the setting of its mode.

HARDWARE DEPENDENCIES

Integral PC

Super-user capabilities are provided to the normal user.

A file currently open for execution is writeable.

SEE ALSO

chmod(2), stat(2).

NAME

acct – enable or disable process accounting

SYNOPSIS

int acct (path)
char *path;

DESCRIPTION

Acct is used to enable or disable the system's process accounting routine. If the routine is enabled, an accounting record will be written on an accounting file for each process that terminates. Termination can be caused by one of two things: an *exit* call or a signal; see *exit*(2) and *signal*(2). The effective user ID of the calling process must be super-user to use this call.

Path points to a path name naming the accounting file. The accounting file format is given in *acct*(4).

The accounting routine is enabled if *path* is non-zero and no errors occur during the system call. It is disabled if *path* is zero and no errors occur during the system call.

The system shuts off accounting when the file size exceeds a system dependent limit.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

Acct will fail if one or more of the following are true:

[EPERM]	The effective user ID of the calling process is not super-user.
[EBUSY]	An attempt is being made to enable accounting when it is already enabled.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	One or more components of the accounting file path name do not exist.
[EACCES]	The file named by *path* is not an ordinary file.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*Path* points to an illegal address. The reliable detection of this error will be implementation dependent.
[ETXTBSY]	*Path* points to a text file which is currently open.
[ENAMETOOLONG]	The accounting file path name exceeds MAXPATHLEN characters.

HARDWARE DEPENDENCIES

Series 200, 300, 500

The system's process accounting routine will ignore any locks placed on the process accounting file.

If the size of the process accounting file reaches 5000 blocks, records for processes terminating after that point will be silently lost. However, in that case the *turnacct* command would still sense that process accounting is still enabled. This loss of records can be prevented by the use of *ckpacct* (see *acctsh*(1M)).

Series 200, 300, 800

When the amount of free space on the file system containing the accounting file falls below a configurable threshold, the system prints a message on the console and disables process accounting. Another message is printed and the process accounting is re-enabled when the space reaches a second configurable threshold.

Series 500

Any child process created by *vfork*(2) that does not call *exec*(2) before terminating will not generate a process accounting record.

Integral PC

Process accounting is not supported.

SEE ALSO

acct(1M), exit(2), signal(2), acct(4).

NAME

alarm – set a process's alarm clock

SYNOPSIS

unsigned long alarm (sec)
unsigned long sec;

DESCRIPTION

Alarm instructs the alarm clock of the calling process to send the signal SIGALRM to the calling process after the number of real time seconds specified by *sec* have elapsed; see *signal*(2). Specific implementations may place limitations of the maximum alarm time supported. The constant MAX_ALARM defined in <*sys/param.h*> specifies the implementation specific maximum. Whenever *sec* is greater that this maximum, it is silently rounded down to it. On all implementations, MAX_ALARM is guaranteed to be at least 31 days (in seconds).

Alarm requests are not stacked; successive calls reset the alarm clock of the calling process.

If *sec* is 0, any previously made alarm request is canceled.

Alarms are not inherited by a child process across a *fork*, but are inherited across an *exec*.

On systems which support the *getitimer*(2) and *setitimer*, the timer mechanism used by *alarm* is the same as that used by *ITIMER_REAL*. Thus successive calls to *alarm, getitimer,* and *setitimer* will set and return the state of a single timer.

RETURN VALUE

Alarm returns the amount of time previously remaining in the alarm clock of the calling process.

SEE ALSO

sleep(1), getitimer(2), pause(2), signal(2), sleep(3C).

BUGS

In some implementations, error bounds for alarm are -1, +0 seconds (for the posting of the alarm, not the restart of the process). Thus a delay of 1 second can return immediately. The setitimer routine can be used to create a more precise delay.

NAME

brk, sbrk – change data segment space allocation

SYNOPSIS

int brk (endds)
char *endds;

char *sbrk (incr)
int incr;

DESCRIPTION

Brk and *sbrk* are used to change dynamically the amount of space allocated for the calling process's data segment; see *exec*(2). The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. The newly allocated space is set to zero.

Brk sets the break value to *endds* and changes the allocated space accordingly.

Sbrk adds *incr* bytes to the break value and changes the allocated space accordingly. *Incr* can be negative, in which case the amount of allocated space is decreased.

ERRORS

Brk and *sbrk* will fail without making any change in the allocated space if one or more of the following are true:

[ENOMEM]	Such a change would result in more space being allocated than is allowed by a system-imposed maximum (see *ulimit*(2)).
[ENOMEM]	Such a change would cause a conflict between addresses in the data segment and any attached shared memory segment (see *shmop*(2)).

HARDWARE DEPENDENCIES

Series 200, 300, 800:

[ENOMEM] can also be returned if there is insufficient swap space available.

Series 500:

Brk sets the break value to *endds*.

Brk and *sbrk* will also fail without making any change in the allocated space if such a change would move the program break below the beginning of your process' indirect data area. Note that it is not possible to release the direct data area with this system call.

If your process' indirect data area is paged, then the size of that data area changes in increments of the page size, which is configurable. Consequently, increasing a paged process data area by one byte may cause it to increase by one page, and decreasing it by one byte may do nothing. If your process' data area is not paged, then the size of the process data area changes similarly in increments of 32 bytes.

WARNINGS

The pointer returned by *sbrk* is not necessarily word-aligned. Loading or storing words through this pointer could cause word alignment problems.

Care should be taken when using either *brk*(2) or *sbrk*(2) in conjunction with calls to the *malloc*(3C) or *malloc*(3X) library routines. There is only one program data segment from which all three of these routines allocate and deallocate program data memory. Although it is not recommended practice, it is possible to deallocate program data memory allocated through *malloc*(3C) with a subsequent call to *brk*().

RETURN VALUE

Upon successful completion, *brk* returns a value of 0 and *sbrk* returns the old break value. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

AUTHOR

Brk and *sbrk* were developed by AT&T and HP.

SEE ALSO

exec(2), shmop(2), ulimit(2), end(3C), malloc(3C).

NAME

chdir – change working directory

SYNOPSIS

int chdir (path)
char *path;

DESCRIPTION

Path points to the path name of a directory. *Chdir* causes the named directory to become the current working directory, the starting point for path searches for path names not beginning with */*.

ERRORS

Chdir will fail and the current working directory will be unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path name is not a directory.
[ENOENT]	The named directory does not exist.
[EACCES]	Search permission is denied for any component of the path name.
[EFAULT]	*Path* points outside the allocated address space of the process. The reliable detection of this error will be implementation dependent.
[ENOENT]	*Path* is null.
[ENAMETOOLONG]	The named directory exceeds MAXPATHLEN characters.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

AUTHOR

Chdir was developed by AT&T Bell Laboratories and the Hewlett-Packard Company.

SEE ALSO

cd(1), chroot(2).

NAME

chmod, fchmod – change access mode of file

SYNOPSIS

int chmod (path, mode)
char *path;
int mode;

fchmod (fd, mode)
int fd, mode;

DESCRIPTION

Path points to a path name naming a file. *Fd* is a descriptor for a file. *Chmod* sets the access permission portion of the file's mode according to the bit pattern contained in *mode*.

Access permission bits are interpreted as follows:

04000	Set user ID on execution
02000	Set group ID on execution
02000	Set file locking mode to enforced (shared with set group ID on execution bit)
01000	Save text image after execution
00400	Read by owner
00200	Write by owner
00100	Execute (or search if a directory) by owner
00070	Read, write, execute (search) by group
00007	Read, write, execute (search) by others

The effective user ID of the process must match the owner of the file or be super-user to change the mode of a file.

If the effective user ID of the process is not super-user, mode bit 01000 (save text image on execution) is cleared.

If the effective user ID of the process is not super-user and the effective group ID of the process does not match the group ID of the file and none of the group IDs in the access group list match the group ID of the file, mode bit 02000 (set group ID on execution and enforced file locking mode) is cleared.

The set group ID on execution bit is also used to cause file-locking mode (see *lockf*(2) and *fcntl*(2)) to be enforced. Files with this bit set that are not group-executable will have enforcement set. This may affect future calls to *open*(2), *creat*(2), *read*(2), and *write*(2) on this file.

If an executable file is prepared for sharing then mode bit 01000 prevents the system from abandoning the swap-space image of the program-text portion of the file when its last user terminates. Thus, when the next user of the file executes it, the text need not be read from the file system but can simply be swapped in, saving time.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

Chmod will fail and the file mode will be unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.

[EPERM] The effective user ID does not match the owner of the file and the effective user ID is not super-user.

[EROFS] The named file resides on a read-only file system.

[EFAULT] *Path* points outside the allocated address space of the process. The reliable detection of this error will be implementation dependent.

[ENOENT] A component of the *path* does not exist.

[ENAMETOOLONG]
The named file exceeds MAXPATHLEN characters.

HARDWARE DEPENDENCIES

Series 500

Chmod changes the mode of files created only in the HP-UX environment (that is, not those created by the HP 9000 BASIC Language System).

Fchmod is not currently implemented.

Integral PC

Normal users have all super-user capabilities.

AUTHOR

Chmod was developed by AT&T, the University of California, Berkeley, and HP.

Fchmod was developed by the University of California, Berkeley.

SEE ALSO

chmod(1), chown(2), creat(2), fcntl(2), lockf(2), open(2), mknod(2), read(2), write(2).

NAME

chown, fchown – change owner and group of a file

SYNOPSIS

int chown (path, owner, group)
char *path;
int owner, group;

fchown (fd, owner, group)
int fd, owner, group;

DESCRIPTION

Path points to a path name naming a file. *Fd* is a descriptor for a file. The owner ID and group ID of the file are set to the numeric values contained in *owner* and *group* respectively. A value of –1 can be specified in *owner* or *group* to leave unchanged the file's owner ID or group ID respectively. Note that *owner* and *group* should be less than or equal to 65535, since only the least significant 16 bits are used.

Only processes with effective user ID equal to the file owner or super-user may change the ownership of a file. If privilege groups are supported, the owner of a file may change the ownership only if he is a member of a privilege group allowing *chown*, as set up by *setprivgrp*. All users get *chown* privileges by default.

The group ownership of a file can be changed to any group in the current process's access list or to the real or effective group id of the current process. If privilege groups are supported and the user is permitted the *chown* privilege, then the file can be given to any group.

If *chown* is invoked by other than the super-user, the set-user-ID and set-group-ID bits of the file mode will be cleared.

ERRORS

Chown will fail and the owner and group of the file will remain unchanged if one or more of the following are true:

[EBADF]	*Fd* is not a valid file descriptor.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[EPERM]	EPERM is set when the effective user ID is not super-user and one or more of the following conditions exist:
	The effective user ID does not match the owner of the file.
	When changing the owner of the file, if the owner of the file is not a member of a privilege group allowing *chown*.
	When changing the group of the file, if the owner of the file is not a member of a privilege group allowing *chown* and the group number is not in the current process's access list.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*Path* points outside the allocated address space of the process. The reliable detection of this error will be implementation dependent.
[ENAMETOOLONG]	

The named file exceeds MAXPATHLEN characters.

HARDWARE DEPENDENCIES

Integral Personal Computer:

For superuser capabilities described above, it is not necessary to be superuser.

Series 500:

Chown changes the owner and group of files created only in the HP-UX environment (that is, not those created by the HP 9000 BASIC Language System).

Fchown is not currently implemented.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

AUTHOR

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

SEE ALSO

chown(1), chmod(2).

NAME

chroot – change root directory

SYNOPSIS

int chroot (path)
char *path;

DESCRIPTION

Path points to a path name naming a directory. *Chroot* causes the named directory to become the root directory, the starting point for path searches for path names beginning with /. The user's working directory is unaffected by the *chroot* system call.

The effective user ID of the process must be super-user to change the root directory.

The .. entry in the root directory is interpreted to mean the root directory itself. Thus, .. cannot be used to access files outside the subtree rooted at the root directory.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

Chroot will fail and the root directory will remain unchanged if one or more of the following are true:

[ENOTDIR]	Any component of the path name is not a directory.
[ENOENT]	The named directory does not exist or a component of the *path* does not exist.
[EPERM]	The effective user ID is not super-user.
[EFAULT]	*Path* points outside the allocated address space of the process. The reliable detection of this error will be implementation dependent.
[ENAMETOOLONG]	The named directory exceeds MAXPATHLEN characters.

HARDWARE DEPENDENCIES

Integral PC
Normal users have all super-user capabilities.

SEE ALSO

chroot(1M), chdir(2).

NAME

close – close a file descriptor

SYNOPSIS

int close (fildes)
int fildes;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *Close* closes the file descriptor indicated by *fildes*. All associated file segments which have been locked by this process with the *lockf* function are released (i.e., unlocked).

ERRORS

[EBADF]	*Close* will fail if *fildes* is not a valid open file descriptor.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), exec(2), fcntl(2), lockf(2), open(2), pipe(2).

NAME

creat – create a new file or rewrite an existing one

SYNOPSIS

int creat (path, mode)
char *path;
int mode;

DESCRIPTION

Creat creates a new ordinary file or prepares to rewrite an existing file named by the path name pointed to by *path*.

If the file exists, the length is truncated to 0 and the mode and owner are unchanged. Otherwise, the file's owner ID is set to the effective user ID, of the process, the group ID is set to the effective group ID, of the process, and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows:

All bits set in the process's file mode creation mask are cleared. See *umask*(2).

The "save text image after execution bit" of the mode are cleared. See *chmod*(2).

Upon successful completion, the file descriptor is returned and the file is open for writing (only), even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across *exec* system calls. See *fcntl*(2). No process may have more than a system defined maximum number of files open simultaneously. This is discussed under *open*(2). A new file may be created with a mode that forbids writing.

ERRORS

Creat will fail if one or more of the following are true:

[ENOSPC]	Not enough space on the file system.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist (for example, *path* is null, or a component of *path* does not exist).
[EACCES]	Search permission is denied on a component of the path prefix.
[EACCES]	The file does not exist and the directory in which the file is to be created does not permit writing.
[EROFS]	The named file resides or would reside on a read-only file system.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed.
[EACCES]	The file exists and write permission is denied.
[EISDIR]	The named file is an existing directory.
[EMFILE]	More than the maximum number of file descriptors are currently open.
[EFAULT]	*Path* points outside the allocated address space of the process. The reliable detection of this error will be implementation dependent.
[ENFILE]	The system file table is full.
[ENXIO]	The named file is a character special or block special file, and the device associated with this special file does not exist.
[ENAMETOOLONG]	The path specified exceeds MAXPATHLEN characters.
[EAGAIN]	The file exists, enforcement mode file and record locking is set and there are outstanding record locks on the file.

RETURN VALUE

Upon successful completion, a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2), close(2), dup(2), fcntl(2), lseek(2), open(2), read(2), truncate(2), umask(2), lockf(2), write(2).

NAME

dup – duplicate an open file descriptor

SYNOPSIS

int dup (fildes)
int fildes;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *Dup* returns a new file descriptor having the following in common with the original:

Same open file (or pipe).

Same file pointer (i.e., both file descriptors share one file pointer).

Same access mode (read, write or read/write).

Same file status flags (see *fcntl*(2), F_DUPFD).

The new file descriptor is set to remain open across *exec* system calls. See *fcntl*(2).

The file descriptor returned is the lowest one available.

ERRORS

Dup will fail if one or more of the following are true:

[EBADF]	*Fildes* is not a valid open file descriptor.
[EMFILE]	The maximum number of file descriptors are currently open.

RETURN VALUE

Upon successful completion a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

AUTHOR

Dup was developed by AT&T Bell Laboratories and the Hewlett-Packard Company.

SEE ALSO

close(2), creat(2), dup2(2), exec(2), fcntl(2), open(2), pipe(2).

NAME

dup2 – duplicate an open file descriptor to a specific slot

SYNOPSIS

int dup2(fildes, fildes2)
int fildes, fildes2;

REMARKS

This facility is provided for backwards compatability with Version 7 and BSD systems. *Fcntl* should be used for all new code.

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *Fildes2* is a non-negative integer less than the maximum value allowed for file descriptors. *Dup2* causes *fildes2* to refer to the same file as *fildes*. If *fildes2* already referred to an open file, it is closed first. The file descriptor returned by *dup2* has the following in common with the original:

Same open file (or pipe).

Same file pointer (i.e., both file descriptors share one file pointer.)

Same access mode (read, write or read/write).

Same file status flags (see *fcntl*(2), F_DUPFD).

The new file descriptor is set to remain open across *exec* system calls. See *fcntl*(2).

This call can be accessed by giving either (for Version 7) the **-lV7** or (for 4.1 or 4.2bsd) the **-lBSD** option to *ld*(1).

ERRORS

Dup2 will fail if one or more of the following are true:

[EBADF]	*Fildes* is not a valid open file descriptor.
[EINVAL]	*Fildes2* is not in the range of legal file descriptors.

RETURN VALUE

Upon successful completion a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

close(2), creat(2), dup(2), exec(2), fcntl(2), open(2), pipe(2).

NAME

ems – Extended Memory System

SYNOPSIS

#include <sys/ems.h>

NOTE

System V shared memory is an industry standard mechanism for sharing memory between processes. When portability to other HP-UX system or other vendor's systems is desired, use System V shared memory instead of EMS (see *shmctl*(2), *shmget*(2), and *shmop*(2)).

DESCRIPTION

Extended Memory System consists of system calls which allocate and deallocate address space, map files into address spaces, support shared memory, and change the protection of address spaces. There are separate manual pages for the system calls. This page describes features in common to all the system calls in EMS.

Definitions

memory space	This is the actual physical memory of a machine.
address space	This refers to the logical memory of a process. Memory space is shared by having processes' address space refer to the same memory space.
segment	A contiguous piece of address space.

Properties of a Segment

During the allocation of a segment, the following types of segments can be requested:

MEM_SHARED
The address space is to be sharable with other processes. The data is shared across *fork*(2) (i.e. not copied on a fork).

MEM_PRIVATE
The address space is process local, and is copied on a *fork*(2). All memory segments will be either MEM_SHARED or MEM_PRIVATE; the default is MEM_PRIVATE.

MEM_CODE The address space may, at some time in its lifetime, be made executable.

MEM_DATA The address space may, at some time in its lifetime, be read and/or written. A segment may be MEM_CODE, MEM_DATA, or both. The default type is derived from the initial access permissions:

MEM_R | MEM_W	→	MEM_DATA
MEM_X	→	MEM_CODE
(MEM_R | MEM_W) && MEM_X	→	MEM_CODE | MEM_DATA.

MEM_PAGED
Requests that a segment be created as a paged object. (This is ignored if not significant for a particular implementation).

File Mapping

EMS provides the facility for mapping a file into process address space. This is done via *memallc*(2). Files can be either private or shared.

For private file mapped segments, the address space will contain an image of the file as it existed at the time of the *memallc*(2) call. Subsequent alterations of the file will have no effect on the contents of the address space, and vice-versa.

For shared file mapped segments, the address space is identically the file (at least the mapped portion thereof). Changes to the address space represent changes to the file, and vice-versa. For example, a write or read to or from the address space is, in all ways, equivalent to a file system write or read. Similarly, re-creating (using *creat*(2)) the file will result in the address space containing all zeros.

The access permissions (e.g. read, write) applied to a shared mapped file are established by the first *memallc*(2) referencing that file. Subsequent mappings of the same file by other processes must request identical access permissions.

File mapping, as described above, is only guaranteed to apply to regular local files and block structured device files. File mapping is not applicable to remote files at this time. Attempting to map an unsupported file type will result in error EINVAL.

Note that file mapping, either MEM_PRIVATE or MEM_SHARED, *always* requires read permission on *fileid*. Access modes cannot exceed those on *fileid* for shared, mapped files.

Shared Memory

It is possible to share a memory space between processes. Access to shared memory can occur in two ways. The first way is to associate a file name as the name of the shared memory space. Each related or unrelated process performs a *memallc*(2) to gain access to the shared memory through mapping the file.

Another method of sharing, without the file, is for related processes: a process can allocate a non-file-mapped shared segment; upon a *fork*(2), the child process will have access to the same memory space as the parent.

AUTHOR

Ems was developed by HP.

SEE ALSO

memadvise(2), memallc(2), memchmd(2), memlck(2), memvary(2), shmget(2), shmop(2), shmctl(2), vsadv(2), vson(2),

NAME

errinfo – error indicator

SYNOPSIS

extern int errinfo;

Remarks:

Errinfo is implemented on the Series 500 only.

DESCRIPTION

When an error occurs in a system call, the external variable *errno* is set to the standard HP-UX error number, and more detailed information is stored in the external variable *errinfo*. *Errinfo* obtains its value from the escape code returned by the underlying HP-UX kernel.

Errinfo is not cleared on successful system calls, so it should only be checked after an error has been indicated.

Software that is intended to be portable across HP-UX implementations should not reference *errinfo*.

The *errinfo* values and their meanings are as follows:

VALUE	MEANING
*4	NVM address out of range;
5	buffer request is not within valid range;
6	buffer address space overflow;
*7	address specified does not reference a valid buffer;
*10	specified process priority level out of range;
*11	a non-existent code segment is specified;
*12	attempt to delete non-existent partition;
*13	system parameter not addressable;
*14	system parameter cannot be referenced with an EDS pointer;
*20	invalid message link;
*21	invalid message link;
*22	message limit exceeded;
*23	link limit exceeded;
*24	link being deleted contains processes waiting for messages;
*30	timer canceled;
*31	timer stopped;
*32	cancel already done for specified timer ID;
*33	stop already done for specified timer ID;
*34	timer ID not stopped before cleared;
*35	timer ID not canceled before cleared;
*36	attempt to set time and date to a value outside accessible range (midnight January 1, 1900 to midnight December 31, 25599);
*37	stack extension error;
40	memory overflow (private partition);
41	memory overflow;
42	no free partition available for allocation;
43	segment table overflow;
44	memory controller block overflow;
45	partition overflow;
46	pointer passed as an argument does not point to a valid segment;
47	segment size is out of range;
*48	free space chains are inconsistent, segment map corruption;

*49	free space chains are inconsistent, block map corruption;
50	pointer passed as an argument does not point to a valid segment;
*51	block address within a segment is invalid;
56	device or interface card timed out;
57	system call aborted by *signal*(2);
*59	improper resource management in operating system;
*60	improper resource management in operating system;
*63	routine called for wrong I/O device or at wrong time;
*64	routine called for wrong I/O device or at wrong time;
*65	used in BASIC only;
66	hardware or firmware error in interface card;
*67	I/O transaction aborted by device or interface card;
68	an HP-IO interface card failed its self test;
*69	used during power-up, produces "System halted - incompatible IOP's" message;
*70	no such object;
*73	out of timer ID's;
*74	timer ID out of range;
*75	start_partition parameters not consistent;
*76	parameter to start_partition not addressable;
*77	attempt to change to non-existent partition;
*78	must be a system process to change to partition;
80	device not ready for request, may be busy with some other operation, or power may be off;
81	media is write-protected and cannot be altered;
82	media has been mis-inserted;
83	format switch disables driver from doing a media format operation;
84	media error was detected, usually a CRC, parity, or checksum error; data may not be valid;
85	cannot find record on media; usually indicates trouble in reading the header/servo information on the media;
86	the read check of data written to a record has failed;
88	media may have been changed since last access; buffered data may have to be thrown away;
*89	used to implement internally generated re-tries;
*90	software failure was detected; perhaps data structures were corrupted, or an unexpected event occurred;
91	unknown error; indicates some type of device or interconnect malfunction;
*94	media_active (true) request must be made before first access;
95	a parameter for a particular request is not supported by this driver; usually indicates that the type of card does not support a special function;
97	termination mode is not supported by this device driver;
98	EOI must have a data byte associated with it before it can be written;
*99	driver must be opened for request;
100	record number out of allowed range; usually indicates corrupt directory structure;
*101	the transfer length was negative, zero, or odd for a halfword read or write request; can also indicate a transfer past the end of the media volume;
102	halfword or byte mode transfers are not supported by this driver;
*103	cannot close a locked driver; this is a fault of the calling code;
105	the argument specified for this ioctl request is out of range or points to the wrong type of structure;
106	The ioctl command given is not recognized by this device;

107	an attempt was made to attach two different drivers to the same device; these drivers are incompatible and cannot co-exist; the new driver is not attached, but the old driver remains unchanged;
108	the size of the string is not correct for this string register access;
109	interleave factor not supported by disc; it is either zero, negative, or too big;
110	invalid address was detected by the driver, or the interface card occupies the same subaddress as the device;
*111	capacity of disc exceeds 32-bit record address range assumed by driver;
112	reference to an unsupported pseudo-register was made; if the request accessed multiple registers, the previous (if valid) register accesses were made;
113	HP-IB TCT byte must be at the end of the ATN sequence because you have passed control;
114	a request is not supported by this driver;
115	no driver with that name was found;
116	no driver is available for that card, or the device address value is out of range;
117	write verify is not supported for this mass storage device;
118	length of −1 specifying that a transfer should be used is invalid;
119	an invalid value was assigned to a pseudo-register;
120	data transfer was terminated due to the reception of a secondary address;
121	for buffered devices, a data transfer cannot be satisfied due to un-transferred data from the other direction; for example, a write may not be possible if there is still unread data present on the device;
122	device cannot satisfy this request because of a previous request or the current state of the device;
123	the beginning of the tape was encountered before the operation could be completed;
124	the interface cannot be the HP-IB active controller when doing this operation;
125	synchronous data rate could not be met to complete the operation; system may be too heavily loaded, or the specified bandwidth parameters for this or another device may be wrong;
126	a hardware fault was detected; controller/status card should be examined for further information;
127	the device/interface was not found at the specified address; power may be off, or the address could be wrong;
128	the end of tape was encountered before the operation was complete;
129	the device failed its self test or a diagnostic; no further access to this device should be attempted;
130	the HP-IB interface is too slow for this synchronous device;
131	tape end of file was encountered before request could be completed;
132	the device was busy and could not handle the request;
133	the media is absent from the device;
134	the media is not formatted, and must be formatted before use;
135	too many media errors prevent formatting to complete; formatting operation may be only partially done;
136	the media has no more spares left but had to spare some data; the sparing was not done;
137	the HP-IB interface must be the active controller to execute this operation;
138	the HP-IB interface must be the system controller to execute this operation;
139	no data seen on media after a device specific length of media; this is a sequential tape error;
140	more data was found in the record than was requested for the read operation; the remaining data was lost, and cannot be read by the next read request;

141	the media physical format is incorrect for this disc;
142	media failure has occurred, or the media has deteriorated such that replacement is suggested; writing is no longer allowed; media may only last long enough for a back-up;
143	the HP-IB interface is not addressed to read or write as requested, and because it is active controller, it cannot become addressed;
144	the read or write request data transfer was aborted by an HP-IB IFC or an HP-IB device clear operation;
145	not all the data (or commands) were accepted by the device;
146	not all the data was sourced by the device;
147	controller or unit fault was reported by the device;
148	some failure occurred in receiving the device status result; usually means that not all the status was returned, or the controller reported a failure when the driver attempted to receive the status;
149	the operation cannot be completed because a user programmed hold off has occurred;
*150	system problem or failure;
*151	successful completion of task; should not be visible;
157	the volume label specified in the volume specifier does not match the volume label on the volume;
158	links may not be removed if the file has been opened with the "no purge link" option;
160	cannot open a directory with write access;
161	two or more volumes have the same volume label and the file system is unable to distinguish between them for this request;
162	an attempt was made to access an open file in a way forbidden by the file system;
163	the disc format does not support the requested operation;
164	the file cannot be opened for writing because it is currently being *exec*ed, or the file may not be opened with execute access because it is currently opened for writing;
165	the file/device could not be opened because the system open file table is full; this is caused by a memory overflooverflow
166	a file may not be opened in both "shared" and "exclusive" modes; your access mode conflicts with the current mode;
167	a signal was received while waiting to read or write to a pipe;
168	the request cannot be performed because the designated file is open or in use at the current time;
169	an attempt was made to purge a link to the file without obtaining the necessary access rights;
170	not enough disc space could be allocated to satisfy the request;
171	a file with the same name already exists in the directory;
172	the file ID passed to the system was bad;
173	an attempt was made to read beyond the physical end of the file;
174	tried to write to a pipe for which there are no readers;
*175	the request made is not supported by the file system;
176	same as error 162, except that the file may not be open;
177	a "position" (*lseek*) request was made on a pipe;
178	the device driver specified in the volume specifier does not match the current device driver being used for the volume;
179	the disc format specified in the volume specifier does not match the disc format on the volume;

181	some file in the file path could not be found;
182	the device specified is not a random access blocked device;
183	the disc format on the disc does not support volume labels;
184	the disc format on the disc does not support file passwords;
185	the disc does not contain a recognizable disc format; the disc format name given for an initialize request is not known to the system;
188	the region of the file that was accessed is currently locked;
189	a volume may not be initialized while there are open files on it;
193	a non-directory was specified where a directory was required;
198	the request cannot be satisfied because another file cannot be added to the directory; no i-nodes were available;
201	the request cannot be satisfied because the directory is not empty;
204	the file system was unable to extend a "contiguous" file without creating another extent;
*210	invalid file code;
216	the select code in the device address in the volume specifier is not within the acceptable range for this hardware configuration;
*217	an attempt was made to remove or change a password which does not exist;
*218	an attempt was made to put two identical passwords on a file with different capability sets;
*219	a simple deadlock was encountered when locking a file;
221	the file name is too long (LIF discs support 10 characters, HP 9845 format discs support 6 characters, and SDF discs support 16 characters);
222	invalid character in LIF or HP 9845 format disc file name;
*223	invalid character in LIF or HP 9845 format disc password;
*224	volume label is too long on a LIF or HP 9845 format disc;
*225	password too long on a LIF or HP 9845 format disc;
*226	invalid character in volume label on a LIF or HP 9845 format disc;
*227	invalid date on LIF or HP 9845 format disc;
*228	invalid record size on LIF or HP 9845 format disc;
229	invalid record mode on LIF or HP 9845 format disc;
230	a file name was expected and none was specified, or an attempt was made to purge the "." or ".." links from a directory;
231	a subdirectory was specified when the disc format does not support subdirectories;
232	links not supported on LIF or HP 9845 format discs;
233	non-UNIX systems are not allowed to establish duplicate links to a directory;
234	the device (file) specified for the *mount/umount* request is not a block special device;
235	the device (file) specified for the *umount* request is not currently mounted;
236	a volume could not be unmounted because it is currently being used (there are open files or working directories established on the mounted volume); a volume could not be mounted because it is already mounted; the directory being mounted on is open or is the root directory;
237	an attempt was made to establish a link from one volume to another;
238	raw discs must be *lseek*ed and read/write sizes must be multiples of the device's physical sector size (256 bytes for discs, 1024 bytes for cartridge tapes).
241	the byte address on a file access was outside the acceptable range for the file; the byte address must be non-negative;
242	the file system saw a directory, i-node, or bit map record which contains inconsistent data;
244	an attempt was made to read beyond the logical end of the file;

249	an attempt was made to unlock an unlocked file;
*252	time value out of range;
*253	hours, minutes, or seconds value out of range;
*254	day, month, or year value out of range;
*255	invalid date;
256	specified segment does not exist;
257	page table has not been initialized;
258	page has not been initialized;
259	lock count has overflowed;
260	lock count has underflowed;
261	entire working set cannot be locked;
262	lock length is invalid;
263	segment is not locked;
264	locked segment cannot be extended;
265	page is not locked;
266	segment is not paged;
267	segment is not shared;
268	requested segment lengths are inconsistent;
269	minimum working set request cannot be satisfied;
270	frame pool cannot be expanded;
271	virtual memory device table overflow;
272	virtual memory device index is invalid;
273	default virtual memory device cannot be removed;
274	virtual memory device index is inactive;
275	virtual memory device index is in use;
276	a locked page was encountered;
301	escape through user code for *exec*;
302	target process not found in *kill* call;
303	target process has the wrong real or saved user ID in *kill* call;
304	no processes found in a broadcast signal attempt;
305	signal number out of range;
306	not super-user; requires super-user permission;
307	a bad argument was supplied to a system call;
308	an attempt was made to wait with no children;
309	an intrinsic was aborted by a signal;
310	process stack overflow;
311	unrecognized *ulimit* command;
312	your DB relative argument had an offset greater than 512 Kbytes;
313	fix-up offset exceeds segment size (see *a.out*(5));
314	stack pointer passed to *brk*;
315	invalid segment number in user pointer;
316	an attempt was made to *kill*(0,sig) with no current process group;
317	file number out of range;
318	specified file ID not open;
319	*ioctl* call not implemented;
320	inappropriate *ioctl* command for device;
321	ID not in the range 0 to 65535;
322	invalid function address in *signal*(2) or *sigvector*(2);
323	floating point divide-by-zero;
324	floating point overflow;
325	floating point underflow;
327	wrong number of system call parameters;

328	inconsistent executable file;
329	front panel timeout (series 500, models 30 and 40 only);
330	graphics to internal CRT timed out;
331	graphics hardware does not respond;
*332	unexpected error when performing an open;
*333	unexpected error when performing a close;
334	illegal mode of driver was requested;
335	a buffer was passed to an intrinsic that is too large;
336	DMA terminated abnormally;
337	received one more x coordinate than y coordinate;
343	user program called missing kernel segment;
345	attempt to execute a file which is too small;
346	attempt to execute a file with a bad magic number;
347	unimplemented configure function;
348	maximum stack exceeded;
349	fatal stack overflow;
350	the requested heap size is too big;
358	there is no tty device at this address;
359	this request is not supported by this device;
360	semid, msqid or shmid is not a valid IPC identifier;
361	semnum in semctl(2) or mtype in msgsnd(2) out of range;
362	invalid cmd to semctl(2), msgctl(2), or shmctl(2);
363	nsems out of range in semget(2);
364	ID for key exists but nsems or size inconsistent with existing ID;
365	mtext is greater than msgsz and msg_noerror is false in msgrcv(2);
366	IPC key exists but operation permission denied;
367	IPC operation permission denied;
368	operation requires caller to be super-user or owner or creator of specified IPC ID;
369	ID does not exist and IPC_CREATE not specified;
370	system-imposed limit on number of IDs exceeded; ID not created;
371	ID exists for key, but IPC_CREATE and IPC_EXCL both specified;
372	nsops is greater than the system-imposed maximum;
373	sem_num is less than zero or greater than or equal to the number of semaphores in the set associated with semid;
374	operation would result in suspension of the calling process but IPC_NOWAIT specified;
375	operation would cause semval or semadj value overflow;
376	specified semaphore or message queue ID has been removed from the system;
377	insufficient memory for IPC structure;
378	message queue does not contain message of desired type and IPC_NOWAIT specified;
379	shared memory size or message size (msgsz) out of range;
380	shmaddr is invalid (non-zero);
381	number of shared memory segments per user exceeded;
382	shmflg is invalid (SHM_RDONLY set);
383	no line discipline of the requested value was found;
384	the ioctl command given is not recognized by this device;
385	the argument specified for this ioctl request is out of range or points to the wrong type of structure;
386	an attempt was made to enable process accounting when it was already enabled.
387	the file specified for process accounting is not an ordinary file;
388	lockf deadlock detected;

389	lockf no more free locks;
390	plock permission invalid (not superuser);
391	PROCLOCK is invalid (PROCLOCK, TXTLOCK, or DATLOCK exists);
392	TXTLOCK is invalid (PROCLOCK or TXTLOCK exists);
393	DATLOCK is invalid (PROCLOCK or DATLOCK exists);
394	UNLOCK is invalid (no lock exists);
395	op is invalid (not PROCLOCK, TXTLOCK, DATLOCK, or UNLOCK);
396	*plock* invalid in [vfork,exec] window;
397	*get/setitimer* invalid in [vfork,exec] window;
398	timer specification is invalid;
399	timeval is invalid;
400	no interrupt packet for this file descriptor;
401	illegal mode mask used in hpib_io function call;
*440	internal error;
441	protection modes do not match with existing segment;
442	device is not a 'CS80' device;
443	attempt to add a device not specified with a device file;
444	attempt to pass an EMS intrinsic a parameter which is out of range;
445	attempt to *memchmd* segment codes which are shared by more than one process;
446	attempt to filemap a file which has already been filemapped by process;
447	insufficient memory available to complete *memallc* request;
448	the specified memory address is invalid;
449	attempt to use EMS intrinsic on memory not allocated by *memallc*;
450	super-user capability is required to create this kind of file;
451	specified file or directory does not exist;
452	an invalid RPM program descriptor was used;
453	an RPM child process was interrupted;
455	attempt to close file failed;
456	abortive file close occurred; data may have been lost;
457	attempt at an abortive file close failed;
458	incorrect select code; device or address does not exist;
459	too much data was given for an RPM request;
460	a string is too long;
461	a name used for RPM is too long;
462	an invalid file ID was used;
463	an open file could not be found;
464	attempt to create a process has failed;
465	connection limit set by the super-user was reached;
466	login not allowed;
467	RPM was not allowed to create a remote process;
470-483	not enough memory could be found; check the network memory limit set with **npowerup**;
490	TCP security mismatch;
491	remote login failed;
493	an RPM login is invalid;
494	consumer login sequence is invalid;
496	login sequence is invalid;
497	connection attempt was not accepted by the remote system;
498	new inbound path rejected, possibly due to lack of local resources;
500	RPM cannot set up the login environment;
501	RPM service is denied;

502	service instance is denied;
503	login on the producer system is invalid;
505	illegal socket name length was used for IPC;
506	illegal node name length was used for IPC;
507	too many file name sets were given for RFA;
508	too many node names were given in an RFA path specifier;
510	attempt was made to copy a directory;
511	parameter contained an illegal value;
513-516	register number or value is unacceptable;
517	internal error; contact qualified HP support personnel;
518	incorrect file type; cannot create RFA remote file;
519	flag specified for RPM is invalid;
520	an option specified for RPM is invalid;
521	unacceptable format for an RPM option;
522	address given could not be used;
*523-524	internal error; contact qualified HP support personnel;
525	illegal characters in an IPC name;
526	incorrect IPC socket descriptor used;
529	illegal IPC flag value was used;
530	illegal IPC data length was used;
532	illegal IPC control request was used;
533	illegal IPC option structure was used;
535	illegal IPC request value was used;
536	illegal IPC timeout value was specified;
537	IPC receive size too big;
540	IPC send size too big;
541	data unit is too large;
543	IPC socket specified is not a virtual circuit socket;
544	illegal address format;
545	nested remote path names are not allowed;
546	IPC socket specified is not a destination socket;
547	IPC socket specified is not a source socket;
548	error in field endpoint;
559	no local IPC socket descriptors are available;
*560-685	internal error; contact qualified HP support personnel;
690	network is already up;
691-692	network is down;
*694	internal error; contact qualified HP support personnel;
695	network is going down;
700	incorrectly formatted network directory was specified;
701	2285A LAN Unit download file is bad;
705	a LAN Interface hardware problem has been detected;
706	LAN Interface failed its selftest;
707	LAN Interface failed during a transmit attempt;
708	LAN Interface failed during a receive attempt;
709-710	2285A LAN Unit failed during a download;
711	HP-IB Interface failed;
720-722	network transport timeout occurred;
723	remote system did not respond to retransmission attempts;
724-725	no activity on a connection; the connection has been aborted;

726	attempt to establish a connection has failed;
730-732	remote system has violated network protocol;
733	a message is too long;
734	request was made that is unacceptable to the transport or to a remote service;
735	unrecognized RFA request;
736	request is unserviceable at this time;
737	unrecognized RFA request;
738	invalid response from the remote system;
739	remote RPM process has violated network protocol;
740	remote RPM process has reported an unrecognized error;
*741	an unrecoverable network protocol error has occurred;
745	requested service cannot be supplied;
747	system cannot support an interchange operation;
748	system cannot support a restart operation;
749	checkpointing not supported;
750	system cannot support a transient operation;
751	unknown system type;
752	buffer too small;
753	invalid remote file request;
754	an error response was received;
755	RPM does not support the requested feature;
756	remote node's version of IPC is incompatible;
*757	internal error; contact qualified HP support personnel;
761	incorrect or unknown path name;
762-763	destination is unreachable;
764	file specified is not a network special file;
*765	internal error; contact qualified HP support personnel;
*767	internal error; contact qualified HP support personnel;
768	system name used is unknown to the local node;
770-774	connection has been lost;
777	IPC connection request failed;
778	connection to producer is down;
780	name specified for the producer system could not be found;
782	name specified for the consumer system could not be found;
784	insufficient resources on the producer system;
785	insufficient resources on the consumer system;
786-787	not enough memory could be obtained on the remote system. The remote system could be out of physical memory or the network memory limit on the remote node could be too small;
788	IPC socket already exists;
790	IPC socket name could not be found;
792	IPC virtual circuit connection was killed;
794	IPC virtual circuit socket cannot be named;
796	IPC connection is pending;
798	IPC process does not own the socket;
800	IPC operation would block;
804	the program for RPM is invalid;
806	the program for RPM could not be loaded;
808	LAN Interface failed. If resetting the Interface does not eliminate the problem, contact qualified HP personnel.

All *errinfo* values marked with an asterisk (*) indicate a serious system problem which should be checked by qualified HP support personnel.

For errinfo values 360-382, IPC refers to the interprocess communications facilities provided by message queues, shared memory, and semaphores. For errinfo values 450-999, IPC refers to the interprocess communications facilities provided by local area networking.

SEE ALSO

err(1), errnet(2), errno(2), perror(3C).

WARNING

Errinfo is intended for diagnostic purposes only. Values and meanings may change in future releases of HP-UX.

NAME

errno – error indicator for system calls

SYNOPSIS

#include <errno.h>
extern int errno;

DESCRIPTION

Errno is an external variable whose value is set whenever an error occurs in a system call. This value can be used to obtain a more detailed description of the error. An error condition is indicated by an otherwise impossible returned value. This is almost always −1; the individual descriptions specify the details. *Errno* is not cleared on successful system calls, so its value should be checked only when an error has been indicated.

Each system call description attempts to list all possible error numbers. The following is a complete list of the error names. The numeric values can be found in **<errno.h>** but should not normally be used.

E2BIG	Arg list too long An argument and or environment list longer than maximum supported size is presented to a member of the *exec* family. Other possibilities include: message size or number of semaphores exceeds system limit *(msgop, semop)*, or too many privileged groups have been set up *(setprivgrp)*.
EACCES	Permission denied An attempt was made to access a file or IPC object in a way forbidden by the protection system.
EADDRINUSE	Address already in use Only one usage of each address is normally permitted.
EADDRNOTAVAIL	Can not assign requested address Normally results from an attempt to create a socket with an address not on this machine.
EAFNOSUPPORT	Address family not supported by protocol family An address incompatible with the requested protocol was used. For example, you should not necessarily expect to be able to use PUP Internet addresses with ARPA Internet protocols.
EAGAIN	No more processes A *fork* failed because the system's process table is full or the user is not allowed to create any more processes, or a *semop* or *msgop* call would have to block.
EALREADY	Operation already in progress An operation was attempted on a non-blocking object which already had an operation in progress.
EBADF	Bad file number Either a file descriptor refers to no open file, a read (respectively write) request is made to a file which is open only for writing (respectively reading), or the file descriptor is not in the legal range of file descriptors.
EBUSY	Device or resource busy An attempt to mount a device that was already mounted or an attempt was made to dismount a device on which there is an active file (open file, current directory, mounted-on file, active text segment). It will also occur if an attempt is made to enable accounting when it is already enabled. The device or resource

is currently unavailable, such as when a non-shareable device file is in use.

ECHILD	No child processes A *wait* was executed by a process that had no existing or unwaited-for child processes.
ECONNABORTED	Software caused connection abort A connection abort was caused internal to your host machine.
ECONNREFUSED	Connection refused No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host.
ECONNRESET	Connection reset by peer A connection was forcibly closed by a peer. This normally results from the peer executing a *shutdown*(2) call.
EDEADLK	Resource deadlock would occur A process which has locked a system resource would have been put to sleep while attempting to access another process' locked resource.
EDESTADDRREQ	Destination address required A required address was omitted from an operation on a socket.
EDOM	Math argument The argument of a function in the math package (3M) is out of the domain of the function.
EEXIST	File exists An existing file was mentioned in an inappropriate context, e.g., *link*.
EFAULT	Bad address The system encountered a hardware fault in attempting to use an argument of a system call; can also result from passing the wrong number of parameters to a system call. The reliable detection of this error will be implementation dependent.
EFBIG	File too large The size of a file exceeded the maximum file size (for the file system) or ULIMIT was exceeded (see *ulimit*(2)), or a bad semaphore number in a *semop*(2) call.
EHOSTDOWN	Host is down A socket operation encountered a dead host. Networking activity on the local host has not been intiated.
EHOSTUNREACH	No route to host A socket operation was attempted to an unreachable host.
EIDRM	Identifier Removed This error is returned to processes that resume execution due to the removal of an identifier from the file system's name space (see *msgctl*(2), *semctl*(2), and *shmctl*(2)).
EINPROGRESS	Operation now in progress An operation which takes a long time to complete was attempted on a non-blocking object (see *ioctl*(2) and *fcntl*(2)).

EINTR	Interrupted system call An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition unless the system call is restarted (see *sigvector*(2)).
EINVAL	Invalid argument Some invalid argument (e.g., dismounting a non-mounted device; mentioning an undefined signal in *signal*, or *kill*; reading or writing a file for which *lseek* has generated a negative pointer). Also set by the math functions described in the (3M) entries of this manual.
EIO	I/O error Some physical I/O error. This error may in some cases occur on a call following the one to which it actually applies.
EISCONN	Socket is already connected A *connect* request was made on an already connected socket, or, a *sendto* or *sendmsg* request on a connected socket specified a destination other than the connected party.
EISDIR	Is a directory An attempt to open a directory for writing.
EMFILE	Too many open files No process may have more than a system defined number of file descriptors open at a time.
EMLINK	Too many links An attempt to make more than the maximum number of links to a file.
EMSGSIZE	Message too long The socket requires that the message be sent atomically, and the size of the message to be sent made this impossible.
ENAMETOOLONG	File name too long A path specified exceeds the maximum path length for the system. The maximum path length is specified by MAXPATHLEN and is kept in <sys/param.h>. If this MAXPATHLEN is not defined for an implementation, that implementation does not generate this error. MAXPATHLEN is guaranteed to be at least 1023 characters.
ENET	Local area network error An error occurred in the software or hardware associated with your local area network.
ENETDOWN	Network is down A socket operation encountered a dead network.
ENETRESET	Network dropped connection on reset The host you were connected to crashed and rebooted.
ENETUNREACH	Network is unreachable A socket operation was attempted to an unreachable network.
ENFILE	File table overflow The system's table of open files is full, and temporarily no more *opens* can be accepted.

ENOBUFS	No buffer space available An operation on a socket was not performed because the system lacked sufficient buffer space.
ENODEV	No such device An attempt was made to apply an inappropriate system call to a device; e.g., read a write-only device.
ENOENT	No such file or directory This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a path name does not exist. It also occurs with *msgget, semget, shmget* when *key* does not refer to any object and the *IPC_CREAT* flag is not set.
ENOEXEC	Exec format error A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid magic number (see *a.out*(4)), or the file is too small to have a valid executable file header.
ENOMEM	Not enough space During a system call such as *exec*, *brk*, *fork*, or *sbrk*, a program asks for more space than the system is able to supply. This may not be a temporary condition; the maximum space size is a system parameter. The error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers, or if there is not enough swap space during a *fork*.
ENOMSG	No message of desired type An attempt was made to receive a message of a type that does not exist on the specified message queue; see *msgop*(2).
ENOPROTOOPT	Protocol not available A bad option was specified in a *getsockopt*(2) or *setsockopt*(2) call.
ENOSPC	No space left on device During a *write* to an ordinary file, there is no free space left on the device; or, no space in system table during *msgget*(2), *semget*(2), or *semop*(2) while **SEM_UNDO** flag is set.
ENOTBLK	Block device required A non-block file was mentioned where a block device was required, e.g., in *mount*.
ENOTCONN	Socket is not connected An request to send or receive data was disallowed because
ENOTDIR	Not a directory A non-directory was specified where a directory is required, for example in a path prefix or as an argument to *chdir*(2).
ENOTEMPTY	Directory not empty An attempt was made to remove a non-empty directory.
ENOTSOCK	Socket operation on non-socket An operation was attempted on something that is not a socket.
ENOTTY	Not a typewriter The (*ioctl*(2)) command is inappropriate to the selected device type.
ENXIO	No such device or address I/O on a special file refers to a subdevice which does not exist, or beyond the

limits of the device. It may also occur when, for example, a tape drive is not online or no disk pack is loaded on a drive.

EOPNOTSUPP	Operation not supported on socket For example, trying to *accept* a connection on a datagram socket.
EPERM	Not owner Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or super-user. It is also returned for attempts by ordinary users to do things allowed only to the super-user.
EPFNOSUPPORT	Protocol family not supported The protocol family has not been configured into the system or no implementation for it exists. the socket is not connected.
EPIPE	Broken pipe A write on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.
EPROTONOSUPPORT	Protocol not supported The protocol has not been configured into the system or no implementation for it exists.
EPROTOTYPE	Protocol wrong type for socket A protocol was specified that does not support the semantics of the socket type requested. For example you cannot use the ARPA Internet UDP protocol with type SOCK_STREAM.
ERANGE	Result too large The value of a function in the math package (3M) is not representable within machine precision, or a *semop*(2) call would cause either a semaphore value or a semaphore adjust value to exceed it system-imposed maximum.
EROFS	Read-only file system An attempt to modify a file or directory was made on a device mounted read-only.
ESHUTDOWN	Cannot send after socket shutdown A request to send data was disallowed because the socket had already been shut down with a previous *shutdown*(2) call.
ESOCKTNOSUPPORT	Socket type not supported The support for the socket type has not been configured into the system or no implementation for it exists.
ESPIPE	Illegal seek An *lseek* was issued to a pipe.
ESRCH	No such process No process can be found corresponding to that specified by *pid* in *kill, rtprio* or *ptrace*, or the process is not accessible.
ETIMEDOUT	Connection timed out A *connect* request failed because the connected party did not properly respond after a period of time. (The timeout period is dependent on the communication protocol.)

ETXTBSY	Text file busy An attempt to execute an executable file which is currently open for writing (or reading). Also, an attempt to open for writing an otherwise writable file which is currently open for execution.
EWOULDBLOCK	Operation would block An operation which would cause a process to block was attempted on a object in non-blocking mode (see *ioctl*(2) and *fcntl*(2)).
EXDEV	Cross-device link A link to a file on another device was attempted.

HARDWARE DEPENDENCIES

Series 500:

In the definition of error 12 (ENOMEM), the maximum space size is not a system parameter. Also, the terms "text, data, and stack segments", "segmentation registers", and "swap space" are invalid.

In the definition of error 31 (EMLINK), the maximum number of links is 32767.

This additional *errno* value is implemented:

EUNEXPECT	Unexpected error An unexpected error was returned from the system, indicating some type of system problem. This error should never occur; if it does, it indicates a system bug.

A second error indicator, *errinfo*, is implemented in addition to *errno.*

Series 800:

In the definition of error ENOMEM, the term "segmentation registers" is invalid.

NAME

execl, execv, execle, execve, execlp, execvp – execute a file

SYNOPSIS

```
int execl (path, arg0, arg1, ..., argn, 0)
char *path, *arg0, *arg1, ..., *argn;

int execv (path, argv)
char *path, *argv[ ];

int execle (path, arg0, arg1, ..., argn, 0, envp)
char *path, *arg0, *arg1, ..., *argn, *envp[ ];

int execve (path, argv, envp)
char *path, *argv[ ], *envp[ ];

int execlp (file, arg0, arg1, ..., argn, 0)
char *file, *arg0, *arg1, ..., *argn;

int execvp (file, argv)
char *file, *argv[ ];
```

DESCRIPTION

Exec, in all its forms, loads a program from an ordinary, executable file onto the current process, replacing the current program. This file is either an executable object file, or a file of data for an interpreter, called a script file.

An executable object file consists of a header (see *a.out*(4)), a text segment, and a data segment. The data segment contains an initialized portion and an uninitialized portion (bss). For *execlp* and *execvp* the shell (*/bin/sh*) may be loaded to interpret a script instead. There can be no return from a successful *exec* because the calling program is overlaid by the new program.

When a C program is executed, it is called as follows:

```
main (argc, argv, envp)
int argc;
char **argv, **envp;
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least one and the first member of the array points to a string containing the name of the file. (The exit conditions from *main* are discussed in *exit*(2)).

Path points to a path name that identifies the executable file containing the new program.

File (in *execlp* or *execvp*) points to a file name identifying the executable file containing the new program. The path prefix for this file is obtained by a search of the directories passed as the *environment* line "PATH =" (see *environ*(5)). The environment is supplied by the shell (see *sh*(1)). If *file* does not have an executable magic number (*magic*(4)), then it is passed to */bin/sh* under the assumption that *file* is a shell script.

Arg0, *arg1*, ..., *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new program. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or its last component).

Argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new program. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or its last component). *Argv* is terminated by a null pointer.

Envp is an array of character pointers to null-terminated strings. These strings constitute the environment in which the new program will run. *Envp* is terminated by a null pointer. For *execl* and *execv*, the C run-time start-off routine places a pointer to the environment of the calling program in the global cell:

extern char **environ;

and it is used to pass the environment of the calling program to the new program.

File descriptors open in the calling process remain open in the new program, except for those whose close-on-exec flag is set; see *fcntl*(2). For those file descriptors that remain open, the file pointer is unchanged.

Signals set to terminate the calling program will be set to terminate the new program. Signals set to be ignored by the calling program will be set to be ignored by the new program. Signals set to be caught by the calling program will be set to their default value in the new program; see *signal*(2).

If the set-user-ID mode bit of the executable file pointed to by *path* or *file* is set (see *chmod*(2)), *exec* sets the effective user ID of the new program to the owner ID of the executable file. Similarly, if the set-group-ID mode bit of the executable file is set, the effective group ID of the new program is set to the group ID of the executable file. The real user ID and real group ID of the new program remain the same as those of the calling program. Note that the set-user(group)-id function does not apply to scripts, and thus if *execlp* or *execvp* executes a script, even if it has the set-user(group)-id bits set, they will be ignored.

The saved process group ID is set equal to the current process group ID.

The shared memory segments attached to the calling program will not be attached to the new program (see *shmop*(2)).

Profiling is disabled for the new program; see *profil*(2).

The new program also inherits the following attributes from the calling program:

interval timers (see *getitimer*(2)).
nice value (see *nice*(2))
process ID
parent process ID
process group ID
semadj values (see *semop*(2))
tty group ID (see *exit*(2) and *signal*(2))
trace flag (see *ptrace*(2) request 0)
time left until an alarm clock signal (see *alarm*(2))
current working directory
root directory
file mode creation mask (see *umask*(2))
file size limit (see *ulimit*(2))
utime, *stime*, *cutime*, and *cstime* (see *times*(2))
real-time priority (see *rtprio*(2))
signal mask (see *sigvector*(2))
pending signals

A script file begins with a line of the form "#! interpreter" or "#! interpreter argument", where #! must be the first two bytes of the file. The interpreter name begins with the first character other than space or tab following the #!. When such a file is *exec*'d, the system *exec*'s the specified interpreter, as an executable object file, in its place. Even in the case of *execlp* or *execvp*, no path searching is done on the interpreter name.

The argument is anything after any tabs or spaces following the interpreter name on the #! line, including any imbedded tabs or spaces. If there is an argument, it is passed to the interpreter as *argv*[1] and the name of the script file is passed as *argv*[2]. Otherwise, the name of the script file is passed as *argv*[1]. *argv*[0] is passed as specified in the exec call, unless either *argv* or *argv*[0] is null as specified, in which case a pointer to a null string is passed as *argv*[0]. All other arguments specified in the exec call are passed following the name of the script file (that is, beginning at *argv*[3] if there is an argument; otherwise at *argv*[2]).

If the #! line exceeds some system defined maximum number of characters, an error will be posted and *exec* will not succeed; the line is terminated by either a new line or null character. The minimum value for this limit is 32.

Set-user-id and set-group-id bits are honored for the script and not for the interpreter.

RETURN VALUE

If *exec* returns to the calling program, an error has occurred; the return value will be -1 and *errno* will be set to indicate the error.

ERRORS

Exec will fail and return to the calling program if one or more of the following are true:

[ENOENT]	One or more components of the executable file's path name or the interpreter's path name do not exist.
[ENOTDIR]	A component of the executable file's path prefix or the interpreter's path prefix is not a directory.
[EACCES]	Search permission is denied for a directory listed in the executable file's or the interpreter's path prefix.
[EACCES]	The executable file or the interpreter is not an ordinary file.
[EACCES]	The file pointed to by *path* or *file* is not executable. The super-user cannot *exec* a file unless at least one of the three execute bits is set in the file's mode.
[EACCES]	Read permission is denied for the executable file or the interpreter and the process's trace flag (see *ptrace*(2) request 0) is set.
[ENOEXEC]	The exec is not an *execlp* or *execvp*, and the executable file has the appropriate access permission but there is neither a valid magic number nor a #! in its header.
[ETXTBSY]	The executable file is currently open for writing. Note: normal executable files are only open for a short time when they start execution. Other executable file types may be kept open for a long time, or indefinitely under some circumstances.
[ENOMEM]	The new program requires more memory than is available, or than is allowed by the system-imposed maximum MAXMEM.
[E2BIG]	The number of bytes in the new program's argument list is greater than the system-imposed limit. This limit will be at least 5120 bytes on HP-UX systems.
[EFAULT]	The executable file is not as long as indicated by the size values in its header, or is otherwise inconsistent. The reliable detection of this error will be implementation dependent.
[EFAULT]	*Path*, *argv*, or *envp* point to an illegal address. The reliable detection of this error will be implementation dependent.
[ENOENT]	*Path* is null.
[ENOEXEC]	The number of bytes in the #! line of a script file exceeds the system's maximum.
[ENAMETOOLONG]	The executable file's path name or the interpreter's path name exceeds MAXPATHLEN characters.

HARDWARE DEPENDENCIES

Series 500

References to memory, such as "text segment", "data segment", "initialized portion", "uninitialized portion", and "bss", are invalid. See *a.out*(4) for the Series 500.

Series 800

Unsharable executable files (EXEC_MAGIC magic number produced via the –**N** option of *ld*(1)) are not supported.

Integral PC

Normal users have all super-user capabilities.

SEE ALSO

sh(1), alarm(2), exit(2), fork(2), nice(2), ptrace(2), semop(2), signal(2), times(2), ulimit(2), umask(2), a.out(4), environ(5).

NAME

exit, _exit – terminate process

SYNOPSIS

void exit (status)
int status;

void _exit (status)
int status;

DESCRIPTION

Exit terminates the calling process and passes *exit*'s argument to the system for inspection, see *wait*(2). Returning from *main* in a C program has the same effect as *exit*; the *exit* value is the function value returned by *main*. (This value will be undefined if *main* does not take care to return a value or explicitly call *exit*.)

Exit is equivalent to *_exit,* except that *exit* flushes stdio buffers, while *_exit* does not. Both *exit* and *_exit* terminate the calling process with the following consequences:

All of the file descriptors open in the calling process are closed.

If the parent process of the calling process is executing a *wait*, it is notified of the calling process's termination and the low order eight bits, i.e., bits 0377, of *status* are made available to it, see *wait*(2).

If the parent process of the calling process is not executing a *wait*, and does not have SIGCLD set to SIG_IGN, the calling process is transformed into a *zombie* process. A *zombie* process is a process that only occupies a slot in the process table. It has no other space allocated either in user or kernel space. Time accounting information is recorded for use by *times*(2).

The parent process ID of all of the calling process's existing child processes and zombie processes is set to 1. This means the initialization process (proc1) inherits each of these processes.

Each attached shared memory segment is detached and the value of **shm_nattach** in the data structure associated with its shared memory identifier is decremented by 1, see *shmop*(2).

For each semaphore for which the calling process has set a semadj value, see *semop*(2), that semadj value is added to the semval of the specified semaphore.

If the process has a process, text, or data lock, an *unlock* is performed, see *plock*(2).

An accounting record is written on the accounting file if the system's accounting routine is enabled, see *acct*(2).

If the process ID, tty group ID, and process group ID of the calling process are equal, the SIGHUP signal is sent to each process that has a process group ID equal to that of the calling process. If the process ID and process group ID of the calling process are equal, the process group ID is removed from all processes that belong to the process group of the calling process. However, if the calling process has previously called *setpgrp2*(2) without subsequently calling *setpgrp*(2), the SIGHUP signal will not be sent and process groups will be left unaltered.

If the calling process caused a controlling terminal to be allocated (caused the tty group ID for the terminal to be defined), the controlling terminal is now deallocated and all processes that share this terminal as a controlling terminal have their process group ID removed.

If the current process has any child processes that are being traced, they will be sent a SIGKILL signal. If it has other child processes that are stopped, they will be sent SIGHUP and SIGCONT signals.

HARDWARE DEPENDENCIES

Series 200, 300, 500
  Job control is not supported.

AUTHOR

Exit was developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

Exit conditions (**$?**) in sh(1), acct(2), plock(2), semop(2), shmop(2), signal(2), times(2), vfork(2), wait(2).

NAME

fcntl – file control

SYNOPSIS

```
#include <fcntl.h>

int fcntl (fildes, cmd, arg)
int fildes, cmd;

union {
        int val;
        struct flock *lockdes;
       } arg;
```

DESCRIPTION

Fcntl provides for control over open files. *Fildes* is an open file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

The *cmd*s available are:

F_DUPFD	Return a new file descriptor that has the following characteristics: Lowest numbered available file descriptor greater than or equal to *arg.val*. Same open file (or pipe) as the original file. Same file pointer as the original file (i.e., both file descriptors share one file pointer). Same access mode (read, write or read/write). Same file status flags (i.e., both file descriptors share the same file status flags). The close-on-exec flag associated with the new file descriptor is set to remain open across *exec*(2) system calls.
F_GETFD	Get the close-on-exec flag associated with the file descriptor *fildes*. If the low-order bit is **0** the file will remain open across *exec*(2), otherwise the file will be closed upon execution of *exec*(2).
F_SETFD	Set the close-on-exec flag associated with *fildes* to the low-order bit of *arg.val* (see F_GETFD).
F_GETFL	Get *file* status flags; see *fcntl*(5).
F_SETFL	Set *file* status flags to *arg.val*. Only certain flags can be set; see *fcntl*(5).
F_GETLK	Get the first lock which blocks the lock description given by the variable of type *struct flock* pointed to by *arg*. The information retrieved overwrites the information passed to *fcntl* in the *flock* structure. If no lock is found that would prevent this lock from being created, the structure is passed back unchanged except for the lock type which will be set to F_UNLCK.
F_SETLK	Set or clear a file segment lock according to the variable of type *struct flock* pointed to by *arg.lockdes* (see *fcntl*(5)). The *cmd* F_SETLK is used to establish read (F_RDLCK) and write (F_WRLCK) locks, as well as remove either type of lock (F_UNLCK). If a read or write lock cannot be set, *fcntl* will return immediately with an error value of −1.
F_SETLKW	This *cmd* is the same as F_SETLK except that if a read or write lock is blocked by other locks, the process will sleep until the segment is free to be locked.

A read lock prevents any process from write-locking the protected area. More than one read lock can exist for a given segment of a file at a given time. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any process from read-locking or write-locking the protected area. Only one write lock may exist for a given segment of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access.

The structure *flock* describes the type (*l_type*), starting offset (*l_whence*), relative offset (*l_start*), size (*l_len*), and process id (*l_pid*) of the segment of the file to be affected. The process id field is only used with the F_GETLK *cmd* to return the value for a block in lock. Locks may start and extend beyond the current end of a file, but may not be negative relative to the beginning of the file. A lock may be set to always extend to the end of file by setting *l_len* to zero (0). If such a lock also has *l_start* set to zero (0), the whole file will be locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments for either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process in a *fork*(2) system call.

When enforcement-mode file and record locking is active on a file, see *chmod*(2), future *read* and *write* system calls on the file will be affected by the record locks in effect.

ERRORS

Fcntl will fail if one or more of the following conditions are true thus errno is set accordingly:

[EBADF]	*Fildes* is not a valid open file descriptor.
[EMFILE]	*Cmd* is F_DUPFD and the maximum number of file descriptors is currently open.
[EMFILE]	*Cmd* is F_SETLK or F_SETLKW, the type of lock is a read or write lock and there are no more file-locking headers available (too many files have segments locked).
[EINVAL]	*Cmd* is F_DUPFD and *arg.val* is negative, greater than or equal to the maximum number of file descriptors.
[EINVAL]	*Cmd* is F_GETLK, F_SETLK, or F_SETLKW and *arg.lockdes* or the data it points to is not valid.
[EINVAL]	*Cmd* is not a valid command.
[EACCES]	*Cmd* is F_SETLK, the type of lock (*l_type*) is a read (F_RDLCK) lock or write (F_WRLCK) lock and the segment of a file to be locked is already write-locked by another process, or the type is a write lock (F-WRLCK) and the segment of a file to be locked is already read- or write-locked by another process.
[ENOSPC]	*Cmd* is F_SETLK or F_SETLKW, the type of lock is a read or write lock and there are no more file-locking headers available (too many files have segments locked), or there are no more record locks available (too many file segments locked).
[EDEADLK]	*Cmd* is F_SETLKW, when the lock is blocked by some lock from another process and sleeping (waiting) for that lock to become free. This causes a deadlock situation.

RETURN VALUE

Upon successful completion, the value returned depends on *cmd* as follows:

F_DUPFD	A new file descriptor.
F_GETFD	Value of close-on-exec flag (only the low-order bit is defined).

F_SETFD	Value other than -1.
F_GETFL	Value of file status flags.
F_SETFL	Value other than -1.
F_GETLK	Value other that -1.
F_SETLK	Value other than -1.
F_SETLKW	Value other than -1.

Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

HARDWARE DEPENDENCIES

Series 200, 500

The F_GETLK, F_SETLK, and F_SETLKW commands are not supported.

AUTHOR

Fcntl was developed by the Hewlett-Packard Company, AT&T Bell Laboratories, and The University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

APPLICATION USAGE

Because in the future the variable errno will be set to EAGAIN rather than EACCES when a section of a file is already locked by another process, portable application programs should expect and test for either value, for example:

```
flk->l_type = F_RDLCK;
if (fcntl(fd, F_SETLK, flk) == -1)
        if ((errno == EACCES) || (errno == EAGAIN))
                /*
                 * section locked by another process,
                 * check for either EAGAIN or EACCES
                 * due to different implementations
                 */
        else if ...
                /*
                 * check for other errors
                 */
```

SEE ALSO

close(2), exec(2), lockf(2), open(2), fcntl(5).

FUTURE DIRECTIONS

The error condition which currently sets errno to EACCES will instead set errno to EAGAIN [see also APPLICATION USAGE above].

NAME

fork – create a new process

SYNOPSIS

int fork ()

DESCRIPTION

Fork causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). This means the child process inherits the following attributes from the parent process:

environment
close-on-exec flag (see *exec*(2))
signal handling settings (i.e., **SIG_DFL**, **SIG_IGN**, function address)
set-user-ID mode bit
set-group-ID mode bit
profiling on/off status (see *profil*(2))
nice value (see *nice*(2))
all attached shared memory segments (see *shmop*(2))
process group ID
tty group ID (see *exit*(2) and *signal*(2))
trace flag (see *ptrace*(2) request 0)
current working directory
root directory
file mode creation mask (see *umask*(2))
file size limit (see *ulimit*(2))
real-time priority (see *rtprio*(2))
saved process group ID

The child process differs from the parent process in the following ways:

The child process has a unique process ID.

The child process has a different parent process ID (i.e., the process ID of the parent process).

The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.

All semadj values are cleared (see *semop*(2)).

Process locks, text locks and data locks are not inherited by the child (see *plock*(2)).

The child process's *utime*, *stime*, *cutime*, and *cstime* are set to 0; see *times*(2). The time left until an alarm clock signal is reset to 0, and all interval timers are set to 0 (disabled).

Note that standard I/O buffers are duplicated in the child. Thus, if you fork after a buffered I/O operation that was not flushed, you may get duplicate output.

Vfork is provided as a higher performance version of *fork* on some systems. See *vfork*(2) for details.

RETURN VALUE

Upon successful completion, *fork* returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of −1 is returned to the parent process, no child process is created, and *errno* is set to indicate the error.

The parent and child processes resume execution immediately after the *fork* call; they are identified by the value returned by *fork*.

ERRORS

Fork will fail and no child process will be created if one or more of the following are true:

[EAGAIN]	The system-imposed limit on the total number of processes under execution would be exceeded.
[EAGAIN]	The system-imposed limit on the total number of processes under execution by a single user would be exceeded.
[ENOMEM]	*Fork* will fail if there is not enough swapping and/or physical memory to create the new process.

AUTHOR

Fork was developed by AT&T Bell Laboratories and the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

exec(2), nice(2), plock(2), ptrace(2), semop(2), shmop(2), signal(2), times(2), ulimit(2), umask(2), wait(2).

NAME

fsync – synchronize a file's in-core state with its state on disk

SYNOPSIS

int fsync(fildes)
int fildes;

DESCRIPTION

Fsync causes all modified data and attributes of *fildes* to be moved to a permanent storage device. This normally results in all in-core modified copies of buffers for the associated file to be written to a disk. *Fsync* applies to ordinary files, and applies to block special devices on systems which permit I/O to block special devices.

Fsync should be used by programs which require a file to be in a known state; for example in building a simple transaction facility.

ERRORS

Fsync will fail if one of the following conditions is true and *errno* will be set accordingly:

[EBADF]	*Fildes* is not a valid descriptor.
[EINVAL]	*Fildes* refers to a file type to which *fsync* does not apply.

RETURN VALUE

A 0 value is returned on success. A –1 value indicates an error.

BUGS

The current implementation of this call is expensive for large files.

AUTHOR

Fsync was developed by the Hewlett-Packard Company, and the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

fcntl(2), fcntl(5), open(2), select(2), sync(2), sync(1M).

NAME

ftime – get date and time more precisely

SYNOPSIS

```
#include <sys/types.h>
#include <sys/timeb.h>
ftime(tp)
struct timeb *tp;
```

REMARKS

This facility is provided for backwards compatibility with Version 7 systems. Either *time* or *gettimeofday* should be used for all new code.

DESCRIPTION

Ftime entry fills in a structure pointed to by its argument, as defined by *<sys/timeb.h>*:

```
/*
 * Structure returned by ftime system call
 */
struct timeb {
        time_t   time;
        unsigned short millitm;
        short    timezone;
        short    dstflag;
};
```

The structure contains the time in seconds since 00:00:00 GMT, January 1, 1970, up to 1000 milliseconds of more-precise interval, the local timezone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year. *Gettimeofday* should be consulted for more details on the meaning of the timezone field.

This call can be accessed by giving the **-lV7** option to *ld*(1).

Ftime can fail for exactly the same reasons as *gettimeofday*(2).

HARDWARE DEPENDENCIES

Series 500:

Ftime is not currently supported.

SEE ALSO

date(1), gettimeofday(2), stime(2), time(2), ctime(3C).

BUGS

The millisecond value usually has a granularity greater than one due to the resolution of the system clock. Depending on any granularity (particularly of one) will render code non-portable.

NAME

getgroups – get group access list

SYNOPSIS

#include <sys/param.h>

getgroups(ngroups, gidset)
int ngroups, *gidset;

DESCRIPTION

Getgroups gets the current group access list of the user process and stores it in the array *gidset*. The parameter *ngroups* indicates the number of entries which may be placed in *gidset*. No more than NGROUPS, as defined in <*sys/param.h*>, will ever be returned.

EXAMPLES

The following call to *getgroups*(2) retrieves the group access list of the calling process and stores the group ids in array mygidset:

```
int ngroups = NGROUPS;
int mygidset[NGROUPS];
int ngrps;

                  ngrps = getgroups (ngroups, mygidset);
```

RETURN VALUE

A nonnegative value indicates that the call succeeded, and is the number of elements returned in *gidset*. A value of −1 indicates that an error occurred, and the error code is stored in the global variable *errno*.

ERRORS

The possible errors for *getgroups* are:

[EFAULT]	*Gidset* specifies an invalid address. The reliable detection of this error will be implementation dependent.
[EINVAL]	*Ngroups* is less than the number of groups in the current group access list of the process.

HARDWARE DEPENDENCIES

Series 500:
Getgroups not available.

AUTHOR

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

SEE ALSO

setgroups(2), initgroups(3C)

NAME

gethostname – get name of current host

SYNOPSIS

char hostname[];
gethostname(hostname, sizeof (hostname));

DESCRIPTION

Gethostname returns the standard host name for the current processor, as set by *sethostname*(2). The name is truncated to sizeof(hostname)-1 and is null-terminated.

ERRORS

Gethostname can fail if:

[EFAULT]	*Hostname* points to an illegal address. The reliable detection of this error will be implementation dependent.

AUTHOR

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

SEE ALSO

hostname(1), uname(1), sethostname(2), uname(2).

NAME

getitimer, setitimer – get/set value of interval timer

SYNOPSIS

#include <sys/time.h>

getitimer(which, value)
int which;
struct itimerval *value;

setitimer(which, value, ovalue)
int which;
struct itimerval *value, *ovalue;

DESCRIPTION

The system provides each process with three interval timers, defined in *<sys/time.h>*. The *getitimer* call returns the current value for the timer specified in *which*, while the *setitimer* call sets the value of a timer (optionally returning the previous value of the timer).

A timer value is defined by the *itimerval* structure:

```
struct itimerval {
        struct timeval   it_interval;   /* timer interval */
        struct timeval   it_value;      /* current value */
};
```

If *it_value* is non-zero, it indicates the time to the next timer expiration. If *it_interval* is non-zero, it specifies a value to be used in reloading *it_value* when the timer expires. Setting *it_value* to 0 disables a timer. Setting *it_interval* to 0 causes a timer to be disabled after its next expiration (assuming *it_value* is non-zero).

Time values smaller than the resolution of the system clock are rounded up to this resolution. The machine-dependent clock resolution is *1/HZ* seconds, where the constant *HZ* is defined in *<sys/param.h>*. Time values larger than an implementation-specific maximum value are rounded down to this maximum. The maximum values for the three interval timers are specified by the constants *MAX_ALARM, MAX_VTALARM,* and *MAX_PROF* defined in *<sys/param.h>*. On all implementations, these values are guaranteed to be at least 31 days (in seconds).

The _which_ parameter specifies which timer to use. The possible values are ITIMER_REAL, ITIMER_VIRTUAL, and ITIMER_PROF.

The ITIMER_REAL timer decrements in real time. A SIGALRM signal is delivered when this timer expires.

The ITIMER_VIRTUAL timer decrements in process virtual time. It runs only when the process is executing. A SIGVTALRM signal is delivered when it expires.

The ITIMER_PROF timer decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by interpreters in statistically profiling the execution of interpreted programs. Each time the ITIMER_PROF timer expires, the SIGPROF signal is delivered. Because this signal may interrupt in-progress system calls, programs using this timer must be prepared to restart interrupted system calls.

Interval timers are not inherited by a child process across a *fork*, but are inherited across an *exec*.

NOTES

Three macros for manipulating time values are defined in *<sys/time.h>*. *Timerclear* sets a time value to zero, *timerisset* tests if a time value is non-zero, and *timercmp* compares two time values (beware that >= and <= do not work with this macro).

The timer used with *ITIMER_REAL.* is the same as that used by *alarm*(2). Thus successive calls to *alarm, getitimer,* and *setitimer* will set and return the state of a single timer.

EXAMPLES

The following call to *setitimer*(2) sets the real-time interval timer to expire initially after 10 seconds and every 0.5 seconds thereafter:

```
struct itimerval rttimer;
struct itimerval old_rttimer;

          rttimer.it_value.tv_sec     = 10;
          rttimer.it_value.tv_usec    = 0;
          rttimer.it_interval.tv_sec  = 0;
          rttimer.it_interval.tv_usec = 500000;

          setitimer (ITIMER_REAL, &rttimer, &old_rttimer);
```

HARDWARE DEPENDENCIES

Series 500:

An error is generated if a call is made to *getitimer* or *setitimer* in the [*vfork, exec*] window.

RETURN VALUE

If the calls succeed, a value of 0 is returned. If an error occurs, the value −1 is returned, and a more precise error code is placed in the global variable *errno.*

ERRORS

Getitimer or *setitimer* can fail if:

[EFAULT]	The *value* structure specified a bad address. The reliable detection of this error will be implementation dependent.
[EINVAL]	A *value* structure specified a microsecond value less that zero or greater than or equal to one million.
[EINVAL]	*Which* does not specify one of the three possible timers.

AUTHOR

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

SEE ALSO

alarm(2), gettimeofday(2), signal(2).

NAME

getpid, getpgrp, getppid, getpgrp2 – get process, process group, and parent process ID

SYNOPSIS

int getpid ( )
int getpgrp ( )
int getppid ( )
int getpgrp2 (pid)

DESCRIPTION

Getpid returns the process ID of the calling process.

Getpgrp returns the process group ID of the calling process.

Getppid returns the parent process ID of the calling process.

Getpgrp2 returns the process group ID of the specified process. If *pid* is zero, the call applies to the current process. For this to be allowed, the real or effective user ID of the current process must match the real or saved user ID of the referenced process, the effective user ID of the current process must be super-user, or the referenced process must be a descendant of the current process.

ERRORS

Getpgrp2 will fail if any of the following are true:

[EPERM]	The effective user ID of the current process is not super-user, the real or effective user ID of the current process does not match the real or saved user ID of the specified process, and the specified process is not a descendant of the current process.
[ESRCH]	No process can be found corresponding to that specified by *pid*.

HARDWARE DEPENDENCIES

Series 200, 300, 500

Getpgrp2 is not supported.

AUTHOR

Getpid, *getppid*, *getpgrp*, and *getpgrp2* were developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

exec(2), fork(2), setpgrp(2), signal(2).

NAME

getprivgrp, setprivgrp – get and set special attributes for group

SYNOPSIS

#include <sys/privgrp.h>

int getprivgrp(grplist)
struct privgrp_map grplist[PRIV_MAXGRPS];

int setprivgrp(grpid, mask)
int grpid, mask[PRIV_MASKSIZ];

DESCRIPTION

Setprivgrp associates a kernel capability with a group id. This allows subsetting of super–user like privileges for members of a particular group or groups. *Setprivgrp* takes two arguments: the integer group id and a mask of permissions. The mask is created by treating the access types defined in <sys/privgrp.h> as bit numbers (using 1 for the least significant bit). Thus, privilege number 5 would be represented by the bit 1<<(5-1) or 16. More generally, privilege **p** is represented by:

mask[((**p**-1) / BITS_PER_INT)] & (1 << ((**p**-1) % BITS_PER_INT)).

As it is possible to have more than **word size** distinct privileges, mask is a pointer to an integer array of size **PRIV_MASKSIZ**.

Setprivgrp privileges include those specified in the file **<sys/privgrp.h>**. A process may access the system call protected by a specific privileged group if it belongs to or has an effective group id of a group having access to the system call. All processes are considered to belong to the pseudo-group **PRIV_GLOBAL**.

Specifying a *grpid* of **PRIV_NONE** causes privileges to be revoked on all privileged groups having any of the privileges specified in *mask*. Specifying a *grpid* of **PRIV_GLOBAL** causes privileges to be granted to all processes.

The constant **PRIV_MAXGRPS** in <sys/privgrp.h> defines the system limit on the number of groups which can be assigned privileges. One of these is always the psuedo-group **PRIV_GLOBAL**, allowing for **PRIV_MAXGRPS**-1 actual groups.

Getprivgrp returns a table of the privileged group assignments into a user supplied structure. *Grplist* points to an array of structures of type **privgrp_map** associating a groupid with a privilege mask. Privilege masks are formed by *or*ing together elements from the access types specified in <sys/privgrp.h>. The array may have gaps in it distinguished as having a **priv_groupno** field of **PRIV_NONE**. The group number **PRIV_GLOBAL** gives the global privilege mask. Only information about groups which are in the user's group access list, or about his real or effective group id, is returned to an ordinary user. The complete set is returned to the super-user.

EXAMPLES

The following example prints out PRIV_GLOBAL and the group ids of the privilege groups to which the user belongs:

```
struct privgrp_map pgrplist[PRIV_MAXGRPS];
int i;
int pgid;

        getprivgrp (pgrplist);
        for (i=0; i<PRIV_MAXGRPS; i++) {
                if ((pgid = pgrplist[i].priv_groupno) != PRIV_NONE) {
                        if (pgid == PRIV_GLOBAL)
                                printf ("(PRIV_GLOBAL) ");
                        printf ("privilege group id = %d\n", pgid);
                }
        }
```

NOTES

Only the super-user may use *setprivgrp*.

ERRORS

Setprivgrp returns -1 and an error code in *errno* if:

[EPERM]	The caller is not super user.
[EFAULT]	*Mask* points to an illegal address. The reliable detection of this error will be implementation dependent.
[EINVAL]	*Mask* has bits set for one or more unknown privileges.
[E2BIG]	The request would require assigning privileges to more than **PRIV_MAXGRPS** groups.

Getprivgrp returns -1 and an error code in *errno* if:

[EFAULT]	*Grplist* points to an illegal address. The reliable detection of this error will be implementation dependent.

Both calls return 0 on success.

AUTHOR

Getprivgrp was developed by HP.

SEE ALSO

getprivgrp(1), setgroups(2), setprivgrp(1M), privgrp(4).

NAME

gettimeofday, settimeofday – get/set date and time

SYNOPSIS

```
#include <time.h>

gettimeofday(tp, tzp)
struct timeval *tp;
struct timezone *tzp;

settimeofday(tp, tzp)
struct timeval *tp;
struct timezone *tzp;
```

DESCRIPTION

Gettimeofday returns the system's notion of the current Greenwich time and the system's notion of the current time zone. Time returned is expressed relative in seconds and microseconds since midnight January 1, 1970.

The structures pointed to by *tp* and *tzp* are defined in <*time.h*> as:

```
struct timeval {
     unsigned long    tv_sec;     /* seconds since Jan. 1, 1970 */
     long             tv_usec;    /* and microseconds */
};

struct timezone {
     int     tz_minuteswest;    /* of Greenwich */
     int     tz_dsttime;        /* type of dst correction to apply */
};
```

The *timezone* structure indicates the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year. Programs should use this timezone information only in the absence of the TZ environment variable.

Only the super-user may set the time of day.

EXAMPLES

The following example calls *gettimeofday*(2) twice. It then computes the lapsed time between the calls in seconds and microseconds and stores the result in a timeval structure:

```
struct timeval    first,
                  second,
                  lapsed;
struct timezone tzp;


        gettimeofday (&first, &tzp);

        /* lapsed time */

        gettimeofday (&second, &tzp);

        if (first.tv_usec > second.tv_usec) {
                  second.tv_usec += 1000000;
                  second.tv_sec--;
        }
        lapsed.tv_usec = second.tv_usec - first.tv_usec;
        lapsed.tv_sec  = second.tv_sec  - first.tv_sec;
```

RETURNS

A 0 return value indicates that the call succeeded. A -1 return value indicates an error occurred, and in this case an error code is stored into the global variable *errno*.

ERRORS

The following error codes may be set in *errno*:

[EFAULT]	An argument address referenced invalid memory. The reliable detection of this error will be implementation dependent.
[EPERM]	A user other than the super-user attempted to set the time.

WARNINGS

The microsecond value usually has a granularity much greater than one due to the resolution of the system clock. Depending on any granularity (particularly of one) will render code non-portable.

HARDWARE DEPENDENCIES

Normal users have all super-user capabilities.

AUTHOR

Gettimeofday was developed by the University of California, Berkeley.

SEE ALSO

date(1), stime(2), time(2), ctime(3C).

NAME

getuid, geteuid, getgid, getegid – get real user, effective user, real group, and effective group IDs

SYNOPSIS

unsigned short getuid ()

unsigned short geteuid ()

unsigned short getgid ()

unsigned short getegid ()

DESCRIPTION

Getuid returns the real user ID of the calling process.

Geteuid returns the effective user ID of the calling process.

Getgid returns the real group ID of the calling process.

Getegid returns the effective group ID of the calling process.

SEE ALSO

setuid(2).

NAME

ioctl – control device

SYNOPSIS

#include <sys/ioctl.h>

ioctl (fildes, request, arg)
int fildes, request;

DESCRIPTION

Ioctl performs a variety of functions on character special files (devices). The write-ups of various devices in Section (7) discuss how *ioctl* applies to them. The type of *arg* is dependent on the specific *ioctl* call, as described in Section (7).

Request is made up of several fields. They encode the size and direction of the argument (referenced by *arg*), as well as the desired command. An enumeration of the request fields are:

IOC_IN	Argument is read by the driver. (That is, the argument is copied from the application to the driver.)
IOC_OUT	Argument is written by the driver. (That is, the argument is copied from the driver to the application.)
IOCSIZE_MASK	Number of bytes in the passed argument. A nonzero size indicates that *arg* is a pointer to the passed argument. A zero size indicates that *arg* is the passed argument (if the driver wants to use it), and is not treated as a pointer.
IOCCMD_MASK	The request command itself.

When both IOC_IN and IOC_OUT are zero, it can be assumed that *request* is not encoded for size and direction, for compatibility purposes. Requests which do not require any data to be passed and requests which use *arg* as a value (as opposed to a pointer), have the IOC_IN bit set to one and the IOCSIZE_MASK field set to zero.

The following macros are used to create the request argument. X and y are concatenated ((x<<8) | y) to form IOCCMD and shifted into the proper location according to **IOCCMD_MASK.** T is the type (e.g. struct hpib_cmd) of the actual argument that the request references, and its size is taken and shifted into the appropriate place according to **IOCSIZE_MASK.**

_IOR(x,y,t)	Sets IOC_OUT and initializes the values at IOCCMD_MASK and IOCSIZE_MASK accordingly.
_IOW(x,y,t)	Sets IOC_IN and initializes the values at IOCCMD_MASK and IOCSIZE_MASK accordin gly.
_IOWR(x,y,t)	Sets both IOC_IN and IOC_OUT and initializes the values at IOCCMD_MASK and IOCSIZE_MASK.

Note: any data structure referenced by *arg* may *not* contain any pointers.

RETURNS

If an error has occurred, a value of −1 is returned and *errno* is set to indicate the error.

Ioctl will fail if one or more of the following are true:

[EBADF]	*Fildes* is not a valid open file descriptor.
[ENOTTY]	The request is not appropriate to the selected device.
[EINVAL]	*Request* or *arg* is not valid.
[EINTR]	A signal was caught during the *ioctl* system call.

[EPERM]	Typically this error indicates that an ioctl request was attempted that is forbidden in some way to the calling process.

WARNINGS

Check all references to *signal*(2) for appropriateness on systems that support *sigvector*(2). *Sigvector*(2) can affect the behavior described on this page.

HARDWARE DEPENDENCIES

Series 200, 300, 500

Asynchronous I/O is not supported.

AUTHOR

Ioctl was developed by AT&T Bell Laboratories and the Hewlett-Packard Company.

SEE ALSO

ioctl(5), termio(7).

NAME

kill – send a signal to a process or a group of processes

SYNOPSIS

int kill (pid, sig)
int pid, sig;

DESCRIPTION

Kill sends a signal to a process or a group of processes. The process or group of processes to which the signal is to be sent is specified by *pid*. The signal that is to be sent is specified by *sig* and is either one from the list given in *signal*(2), or **0**. If *sig* is **0** (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The real or effective user ID of the sending process must match the real or saved user ID of the receiving process, unless the effective user ID of the sending process is super-user. As a single special case on those systems that support job control, the continue signal SIGCONT can be sent to any process that is a descendant of the current process.

The value KILL_ALL_OTHERS is defined in the file **<sys/signal.h>** and is guaranteed not to be the ID of any process in the system or the negation of the ID of any process in the system.

If *pid* is greater than zero and not equal to KILL_ALL_OTHERS, *sig* will be sent to the process whose process ID is equal to *pid*. *Pid* can equal **1** unless *sig* is SIGKILL or SIGSTOP.

If *pid* is **0**, *sig* will be sent to all processes excluding special system processes whose process group ID is equal to the process group ID of the sender.

If *pid* is **−1** and the effective user ID of the sender is not super-user, *sig* will be sent to all processes excluding special system processes whose real or saved user ID is equal to the real or effective users ID of the sender.

If *pid* is **−1** and the effective user ID of the sender is super-user, *sig* will be sent to all processes excluding special system processes.

If *pid* is KILL_ALL_OTHERS the behavior is the same as for *pid* equal to **−1** except that *sig* is not sent to the calling process.

If *pid* is negative but not **−1** or KILL_ALL_OTHERS, *sig* will be sent to all processes (excluding special system processes) whose process group ID is equal to the absolute value of *pid*, and whose real and/or effective *uid* meet the constraints described above for matching *uid*s.

ERRORS

Kill will fail and no signal will be sent if one or more of the following are true:

[EINVAL]	*Sig* is not a valid signal number or zero.
[EINVAL]	*Sig* is SIGKILL or SIGSTOP and *pid* is **1** (proc1).
[EPERM]	The user ID of the sending process is not super-user, and its real or effective user ID does not match the real or effective user ID of the receiving process.
[ESRCH]	No process can be found corresponding to that specified by *pid*.

RETURN VALUE

Upon successful completion, a value of **0** is returned. Otherwise, a value of **−1** is returned and *errno* is set to indicate the error.

HARDWARE DEPENDENCIES

Integral Personal Computer
  Normal users have all super-user capabilities.

Series 200, 300, 500
  Job control is not supported.

AUTHOR

Kill was developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

kill(1), getpid(2), setpgrp(2), signal(2).

NAME

link - link to a file

SYNOPSIS

int link (path1, path2)
char *path1, *path2;

DESCRIPTION

Path1 points to a path name naming an existing file. *Path2* points to a path name naming the new directory entry to be created. *Link* creates a new link (directory entry) for the existing file.

ERRORS

Link will fail and no link will be created if one or more of the following are true:

[ENOTDIR]	A component of either path prefix is not a directory.
[ENOENT]	A component of either path prefix does not exist.
[ENOSPC]	The directory to contain the file cannot be extended.
[EACCES]	A component of either path prefix denies search permission.
[ENOENT]	The file named by *path1* does not exist.
[EEXIST]	The link named by *path2* exists.
[EPERM]	The file named by *path1* is a directory and the effective user ID is not super-user.
[EXDEV]	The link named by *path2* and the file named by *path1* are on different logical devices (file systems).
[ENOENT]	*Path2* points to a null path name.
[EACCES]	The requested link requires writing in a directory with a mode that denies write permission.
[EROFS]	The requested link requires writing in a directory on a read-only file system.
[EFAULT]	*Path* points outside the allocated address space of the process. The reliable detection of this error will be implementation dependent.
[ENOENT]	*Path1* or *path2* is null.
[EMLINK]	The maximum number of links to a file would be exceeded.
[ENAMETOOLONG]	Either path specified exceeds MAXPATHLEN characters.

HARDWARE DEPENDENCIES

Series 500:

For Structured Directory Format (SDF) disks, if *path2* is "..", then that directory's inode will be altered such that its ".." entry points to the directory specified by *path1*. In this way, the super-user can establish the parent directory of an existing directory.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

cp(1), link(1M), unlink(2).

NAME

lockf – provide semaphores and record locking on files

SYNOPSIS

```
#include <unistd.h>

lockf(fildes, function, size)
long size;
int fildes, function;
```

DESCRIPTION

Lockf will allow regions of a file to be used as semaphores (advisory locks) or accessible only by the locking process (enforcement mode record locks). Other processes which attempt to access the locked resource will either return an error or sleep until the resource becomes unlocked. All the locks for a process are removed when the process closes the file or terminates.

Fildes is an open file descriptor.

Function is a control value which specifies the action to be taken. The permissible values for *function* are defined in **<unistd.h>** as follows:

```
#define F_ULOCK    0    /* Unlock a region */
#define F_LOCK     1    /* Lock a region */
#define F_TLOCK    2    /* Test and Lock a region */
#define F_TEST     3    /* Test region for lock */
```

All other values of *functions* are reserved for future extensions and will result in an error return if not implemented.

F_TEST is used to detect if a lock by another process is present on the specified region. **F_TEST** returns zero if the region is accessable and minus one (–1) if it is not; in this case *errno* will be set to EACCES. **F_LOCK** and **F_TLOCK** both lock a region of a file if the region is available. **F_ULOCK** removes locks from a region of the file.

Size is the number of contiguous bytes to be locked or unlocked. The resource to be locked starts at the current offset in the file, and extends forward for a positive *size*, and backward for a negative *size* (the preceding bytes up to but not including the current offset). If *size* is zero the region from the current offset through the end of the largest possible file is locked (i.e., from the current offset through the present or any future end-of-file). An area need not be allocated to the file in order to be locked, as such locks may exist past the end of the file.

The regions locked with **F_LOCK** or **F_TLOCK** may, in whole or part, contain or be contained by a previously locked region for the same process. When this occurs or if adjacent regions occur, the regions are combined into a single region. If the request requires that a new element be added to the table of active locks and this table is already full, an error is returned, and the new region is not locked.

F_LOCK and **F_TLOCK** requests differ only by the action taken if the resource is not available: **F_LOCK** will cause the calling process to sleep until the resource is available, and the **F_TLOCK** will return an [EACCES] error if the region is already locked by another process.

F_ULOCK requests may, in whole or part, release one or more locked regions controlled by the process. When regions are not fully released, the remaining regions are still locked by the process. Releasing the center section of a locked region requires an additional element in the table of active locks. If this table is full, an [EDEADLK] error is returned, and the requested region is not released.

Regular files with the file mode of **S_ENFMT** not having the group execute bit set will have an enforcement policy enabled. With enforcement enabled, reads and writes that would access a locked region will sleep until the entire region is available if O_NDELAY is cleared, but will return −1 with errno set if O_NDELAY is set. File access by other system functions, such as exec, are not subject to the enforcement policy. Locks on directories, pipes, and special files are advisory only; no enforcement policy will be used.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by accessing another process's locked resource. Thus, calls to *fcntl*, *lockf*, *read*, or *write* scan for a deadlock prior to sleeping on a locked resource. Deadlock is not checked for the *wait* and *pause* system calls, so potential for deadlock is not eliminated. A *creat* call or an *open* call with the O_CREATE and O_TRUNC flags set on a regular file will return [EAGAIN] error if another process has locked part of the file and the file is currently in enforcement mode.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

Lockf will fail if any of the following occur:

[EBADF]	If *fildes* is not a valid, open file descriptor.
[EACCES]	If the region is already locked by another process.
[EDEADLK]	If a deadlock would occur; or if the number of entries in the lock table would exceed a system-dependent maximum. HP-UX guarantees this value to be at least 50.
[EINVAL]	If *function* is not one of the functions specified above.
[EINVAL]	If size plus current offset produces a negative offset into the file.

WARNINGS

Deadlock conditions may arise when either the *wait* or *pause* system calls are used in conjunction with enforced locking, see *wait*(2) and *pause*(2) for details.

File and record locking using file descriptors obtained through *dup*(2) or *link*(2) may not work as expected, e.g., unlocking regions which were locked using either file descriptor may also unlock regions which were locked using the other file descriptor.

BUGS

Unexpected results may occur in process that do buffering in the user address space. The process may later read/write data which is/was locked. the standard I/O package, *stdio*(3S), is the most common source of unexpected buffering.

In a hostile environment locking may be misused by holding key public resources locked. This is particularly true with public read files that have enforcement mode enabled.

APPLICATION USAGE

Because in the future the variable errno will be set to EAGAIN rather than EACCES when a section of a file is already locked by another process, portable application programs should expect and test for either value, for example:

```
if (lockf(fd, F_TLOCK, siz) == -1)
    if ((errno == EAGAIN) || (errno == EACCES))
        /*
         * section locked by another process
         * check for either EAGAIN or EACCES
         * due to different implementations
         */
    else if ...
        /*
         * check for other errors
         */
```

HARDWARE DEPENDENCIES

Series 200, 300, 800

For an EINVAL error, the resulting upper bound of the region to be locked would be greater than 2^30. The current offset is greater than 2^30.

SEE ALSO

chmod(2), close(2), creat(2), fcntl(2), open(2), pause(2), read(2), stat(2), wait(2), write(2).

FUTURE DIRECTIONS

The error condition which currently sets errno to EACCES will instead set errno to EAGAIN [see also APPLICATION USAGE above].

NAME

lseek – move read/write file pointer; seek

SYNOPSIS

#include <unistd.h>
long lseek (fildes, offset, whence)
int fildes;
long offset;
int whence;

DESCRIPTION

Fildes is a file descriptor returned from a *creat*, *open*, *dup*, or *fcntl* system call. *Lseek* sets the file pointer associated with *fildes* as follows:

If *whence* is 0, the pointer is set to *offset* bytes.

If *whence* is 1, the pointer is set to its current location plus *offset*.

If *whence* is 2, the pointer is set to the size of the file plus *offset*.

Upon successful completion, the resulting pointer location, as measured in bytes from the beginning of the file, is returned.

RETURN VALUE

Upon successful completion, a non-negative integer indicating the file pointer value is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

Lseek will fail and the file pointer will remain unchanged if one or more of the following are true:

[EBADF]	*Fildes* is not an open file descriptor.
[ESPIPE]	*Fildes* is associated with a pipe or fifo.
[EINVAL and SIGSYS signal]	*Whence* is not 0, 1 or 2.
[EINVAL]	The resulting file pointer would be negative.

WARNINGS

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

Using *lseek* with a whence of 2 on device special files is not supported and the results are not defined.

SEE ALSO

creat(2), dup(2), fcntl(2), open(2).

NAME

memadvise – advise OS about segment reference patterns

SYNOPSIS

```
#include <sys/ems.h>
#include <sys/types.h>

memadvise(addr, len, behav, adrtype)
caddr_t      addr;
int          len, behav;
enum         memtype {mem_code, mem_data} adrtype;
```

DESCRIPTION

The purpose of this call is to allow an application program to notify the system of its known patterns of reference in specific areas of process memory. The intent is to allow the system to then adapt its memory management algorithms and/or policies based on this knowledge to maximize the performance of the program. For example, a program that uses a very large hash table might inform the system of its random patterns of reference to this area. The system might, then, elect not to do any pre-fetching or clustered reads in this area.

Addr is the starting address of the area in question and *len* is the length in bytes. *Addr* may be any legal address in the process's address space. Since some implementations use different (and indistinguishable) addressing formats for code and data space, *adrtype* is used to indicate whether *addr* is a code or data address. On systems with a uniform addressing format for code and data, *adrtype* will have no effect.

The boundaries of the address space for which the advice is applied may be rounded up and/or down to appropriate system dependent values (e.g. pages, segments, blocks, etc).

Variable *behav* describes the reference pattern in the specified area:

MEM_NORMAL	No known extraordinary patterns of reference.
MEM_SEQ	References are highly sequential in nature.
MEM_RANDOM	References are totally random and unpredictable.
MEM_NEEDED	Area is expected to be highly referenced in near future.
MEM_NOTNEEDED	Area is not expected to be referenced in the near future.

Memadvise may be reduced to a no-op, or some of the behavior types may be ignored (treated as no-ops).

AUTHOR

Memadvise was developed by HP.

SEE ALSO

ems(2), memallc(2).

NAME

memallc, memfree – allocate and free address space

SYNOPSIS

```
#include <sys/ems.h>
#include <sys/types.h>

caddr_t     memallc(fileid, offset, len, maxlen, type, mode);
int         fileid, offset, len;
int         maxlen, type, mode;

int         memfree(addr);
caddr_t     addr;
```

DESCRIPTION

Memallc allocates a memory segment (i.e. a contiguous piece of process address space) and returns a pointer to it. The memory segment may be shared (i.e. accessible by other processes) or private. Private segments are copied on *fork*(2), giving separate, per-process images of the segment. Shared segments are not copied across *fork*(2) but, instead, both processes have access to the same memory space. The segment may optionally be initialized to the contents of a specific open file (private mapped file) or can be made equivalent to a specific file (shared mapped file).

Fileid is the HP-UX file id of an open file which will be mapped into the process's address space. *Fileid* must refer to a file on a CS-80 disk. If *fileid* is –1, the allocated address space will be initialized to zeros. A mapping of a file (either shared or private) generates an implicit reference to the file (similar to the result of *dup*(2)). Subsequent to the mapping, *fileid* may safely be closed.

Offset specifies the starting point in *fileid* (i.e. byte offset) where mapping is to begin. The value returned by *memallc* is a pointer to the byte in the new address space that corresponds to byte *offset*. If *fileid* is not specified (i.e. set to -1), *offset* is ignored.

Len specifies the size (in bytes) of the address space. The guaranteed range of accessibility is from *ptr* thru *ptr+len-1* (where *ptr* is the value returned by the *memallc* call). Depending on the value of *offset*, *len*, and the specific implementation, additional data space MAY be accessible at addresses less than *ptr* and/or greater than *ptr+len-1* but the effects of reading and/or writing these areas are undefined.

If *len+offset* is greater than the size of the file, the additional address space is initialized to zeros. If the segment is shared, the file is extended to the required size (if fileid is not writable, the call fails). A *creat*(2) call on a file that has a shared mapping applied to it will zero the file but will not alter the file size.

Maxlen specifies the maximum length to which a segment may grow using *memvary*(2).

Type specifies the attributes assigned to the segment, which is constructed by taking the union of the desired attributes: MEM_SHARED, MEM_PRIVATE, MEM_PAGED, MEM_DATA, or MEM_CODE (see *ems*(2)).

Mode specifies the access permissions assigned to the segment for the requesting process.

MEM_R, MEM_W, MEM_X:
: Initial access modes to be assigned to segment (see *memchmd*(2)).

Note that all MEM_SHARED mappings of a specific file must use identical access modes. An attempt to map a file with access modes different than those already in effect will return an error [EACCES].

Memfree deallocates a memory segment created by *memallc*. It takes, as an argument, a pointer returned by *memallc*. When the segment is shared, the memory will not be deallocated until the last reference to the memory is removed.

The number of segments allocated to a given process at any one time may be limited to a system dependent maximum of **MAXSEGS** found in **ems.h**.

RETURN VALUE

Upon successful completion, *memallc* returns the byte pointer to the address space. Otherwise, a value of -1 is returned and *errno* is set to indicate error.

AUTHOR

Memallc was developed by the Hewlett-Packard Company.

SEE ALSO

ems(2), memchmd(2), memvary(2), shmget(2), shmop(2).

NAME

memchmd – change memory segment access modes

SYNOPSIS

```
#include <sys/ems.h>
#include <sys/types.h>

int         memchmd (addr, mode);
caddr_t     addr;
int         mode;
```

DESCRIPTION

This procedure may be used to change the access mode of a memory segment created by *memallc*(2). The procedure returns the previous access mode (or -1 if there is an error).

Addr is the segment pointer returned by *memallc*(2).

The access modes for a shared segment is an attribute of the segment and is the same for all processes sharing the segment or any portion thereof. The access mode of a segment may not be changed if it is being shared with any other process (e.g. more than one *memallc* of a peculiar file, or a *memallc* followed by a *fork*(2)). An attempt to *memchmd* such a shared segment will return an error [EACCES].

The access mode of a MEM_PRIVATE segment may be changed without restrictions.

The definition of the access modes are:

MEM_X	Execute capability
MEM_W	Write capability
MEM_R	Read capability

An error is returned if *addr* is not a valid segment pointer.

Access modes granted to a MEM_SHARED file mapped segment may not exceed the access modes granted to the user of the file when it was opened.

RETURN VALUE

Upon successful completion, *memchmd*(2) returns the old set of access modes. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

AUTHOR

Memchmd was developed by the Hewlett-Packard Company.

SEE ALSO

ems(2), memallc(2), memvary(2), shmctl(2).

NAME
memlck, memulck – lock/unlock process address space or segment

SYNOPSIS
```
#include <sys/ems.h>
#include <sys/types.h>

int         memlck (addr, len, adrtype);
caddr_t     addr;
int         len;
enum        memtype {mem_code, mem_data} adrtype;

int         memulck (addr, len, adrtype);
caddr_t     addr;
int         len;
enum        memtype {mem_code, mem_data} adrtype;
```

DESCRIPTION
Memlck is used to lock a section of process address space into physical memory. This call may take a substantial amount of time to complete, but the address space in question is guaranteed to be in memory and locked upon successful completion of the call. The locked address space will not migrate to backing store regardless of process state and will, furthermore, remain at the same physical address space for the duration of the lock. Locks are not inherited across *fork*(2). Multiple locks on any address range can occur (unlocking requires that as many unlocks as locks occur). The locks will be segment local, and unlocking may be done by a process unrelated to the one which did the locking. A locked segment will be released when there are no processes with references to the locked segment. (This may occur either via *memfree* on *memallc*(2) or process death.)

Addr is the starting address of the area in question and *len* is the length in bytes. *Addr* may be any legal address in the process's address space. Since some implementations use different (and indistinguishable) addressing formats for code and data space, *adrtype* is used to indicate whether addr is a code or data address. On systems with a uniform addressing format for code and data, *adrtype* will have no effect.

The boundaries of the locked address space may be rounded up (on the upper end of the address range) and down (on the lower end of the address range) to appropriate system dependent values (e.g. pages, segments, blocks, etc). Locking will not cross segment boundaries. For example, one *memlck* call cannot lock part of a text segment and part of a data segment.

Memulck undoes the effects of a *memlck*.

The use of this call is restricted to the super-user.

This call may be reduced to a no-op.

RETURN VALUE
Upon successful completion, *memlck* and *memulck* return a value of 0. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

AUTHOR
Memlck was developed by HP.

SEE ALSO
ems(2), memallc(2), plock(2), shmctl(2).

NAME

memvary – modify segment length

SYNOPSIS

```
#include <sys/ems.h>
#include <sys/types.h>

int          memvary(addr, len);
caddr_t      addr;
int          len;
```

DESCRIPTION

Memvary allows the modification of the size of the memory space allocated by *memallc*(2).

Addr is the pointer to the address space which can be either shared or private. If the address space has been file mapped and is extended beyond the the end of the file, then the file will also reflect the change in length. When the file mapped address space is reduced, the actual file length will be unchanged and the file space after the end of the mapped file space will also remain unchanged. A change in length for a private file mapped address space will have no effect on the source file.

Len specifies the new length of the address space. In the case of an error, the address space and file space will be the same as before the system call.

When private file mapped address space is extended, the additional address space is initialized to zeroes. When shared file mapped address space is extended, the additional space is initialized to the contents of the file, or zeros if the file is extended.

The address space cannot be extended beyond the 'maxlen' specified during the *memallc*(2) system call.

ERRORS

Memvary will fail if one or more of the following are true:

ERRNO	ERRINFO	DESCRIPTION
[ENOMEM]	5, 41	Insufficient memory
[EINVAL]	47	Invalid segment size
	256	Segment does not exist
	264	Cannot extend locked segment
	276	locked page encountered
	449	Segment not allocated with *ems*(2) subsystem

RETURN VALUE

Upon successful completion, *memvary* returns 0. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

AUTHOR

Memvary was developed by the Hewlett-Packard Company.

SEE ALSO

ems(2), memallc(2), memchmd(2).

NAME

mkdir – make a directory file

SYNOPSIS

mkdir(path, mode)
char *path;
int mode;

REMARKS

Not all systems implement this as a system call; some use a library call to the *mkdir*(1) command to achieve the same effect. The errors documented below will appear in any case, and no error messages will ever be printed.

DESCRIPTION

Mkdir creates a new directory file with name *path*. The mode of the new file is initialized from *mode*. (The protection part of the mode is modified by the process's mode mask; see *umask*(2)).

The directory's owner ID is set to the process's effective user ID. The directory's group ID is set to the process's effective group ID.

The low-order 9 bits of mode are modified by the process's file mode creation mask: all bits set in the process's file mode creation mask are cleared. See *umask*(2).

RETURN VALUE

A 0 return value indicates success. A −1 return value indicates an error, and an error code is stored in *errno*.

ERRORS

Mkdir will fail and no directory will be created if:

[ENOSPC]	Not enough space on the file system.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[EROFS]	The named file resides on a read-only file system.
[EEXIST]	The named file exists.
[EFAULT]	*Path* points outside the process's allocated address space. The reliable detection of this error will be implementation dependent.
[EIO]	An I/O error occured while writing to the file system.
[ENAMETOOLONG]	The path specified exceeds MAXPATHLEN characters.

AUTHOR

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

SEE ALSO

chmod(2), stat(2), umask(2)

NAME

mknod - make a directory, or a special or ordinary file

SYNOPSIS

int mknod (path, mode, dev)
char *path;
int mode;
dev_t dev;

DESCRIPTION

Mknod creates a new file named by the path name pointed to by *path*. The mode of the new file is initialized from *mode*, where the value of *mode* is interpreted as follows:

- 0170000 file type; one of the following:
  - 0010000 fifo special
  - 0020000 character special
  - 0040000 directory
  - 0060000 block special
  - 0100000 or 0000000 ordinary file
- 0004000 set user ID on execution
- 0002000 set group ID on execution
- 0001000 save text image after execution
- 0000777 access permissions; constructed from the following:
  - 0000400 read by owner
  - 0000200 write by owner
  - 0000100 execute (search on directory) by owner
  - 0000070 read, write, execute (search) by group
  - 0000007 read, write, execute (search) by others

Values of *mode* other than those above are undefined and should not be used.

The owner ID of the file is set to the effective user ID of the process. The group ID of the file is set to the effective group ID of the process.

The low-order 9 bits of *mode* are modified by the process's file mode creation mask: all bits set in the process's file mode creation mask are cleared. See *umask*(2).

Dev is meaningful only if *mode* indicates a block or character special file, and is ignored otherwise. It is an implementation and configuration dependent specification of a character or block I/O device. *Dev* may be created by using the *makedev* macro defined in <**sys/sysmacros.h**>. The argument to *makedev* are the major and minor device numbers, the value and interpretation of which are implementation dependent. The result of *makedev* is an object of type dev_t.

Mknod may be invoked only by the super-user for file types other than FIFO special.

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

Mknod will fail and the new file will not be created if one or more of the following are true:

[ENOSPC]	Not enough space on the file system.
[EPERM]	The effective user ID of the process is not super-user, and the file type is not FIFO special.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.

[EROFS]	The directory in which the file is to be created is located on a read-only file system.
[EACCES]	A component of the path prefix denies search permission.
[EEXIST]	The named file exists.
[EFAULT]	*Path* points outside the process's allocated address space. The reliable detection of this error will be implementation dependent.
[ENOENT]	*Path* is null.
[EACCES]	*Path* is in a directory that denies write permission, *mode* is for fifo special file, and the caller is not super-user.
[ENAMETOOLONG]	The path specified exceeds MAXPATHLEN characters.

HARDWARE DEPENDENCIES

Series 200, 300, 500

An additional value is available for network special files under file type. Its value is 0110000.

Integral PC

Normal users have all super-user capabilities.

AUTHOR

Mknod was developed by AT&T and HP.

SEE ALSO

mkdir(1), mknod(1M), chmod(2), exec(2), umask(2), fs(4), mknod(4).

NAME

mount – mount a file system

SYNOPSIS

int mount (spec, dir, rwflag)
char *spec, *dir;
int rwflag;

DESCRIPTION

Mount requests that a removable file system contained on the block special device file identified by *spec* be mounted on the directory identified by *dir*. *Spec* and *dir* are pointers to path names.

Upon successful completion, references to the file *dir* will refer to the root directory on the mounted file system.

The low-order bit of *rwflag* is used to control write permission on the mounted file system; if **1**, writing is forbidden, otherwise writing is permitted according to individual file accessibility.

Mount may be invoked only by the super-user.

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

Mount will fail if one or more of the following are true:

[EPERM]	The effective user ID is not super-user.
[ENOENT]	The named file does not exist (for example, *path* is null or a component of *path* does not exist).
[ENOTDIR]	A component of a path prefix is not a directory.
[ENOTBLK]	*Spec* is not a block special device.
[ENXIO]	The device associated with *spec* does not exist.
[ENOTDIR]	*Dir* is not a directory.
[EFAULT]	*Spec* or *dir* points outside the allocated address space of the process. The reliable detection of this error will be implementation dependent.
[EBUSY]	*Dir* is currently mounted on, is someone's current working directory, or is otherwise busy.
[EBUSY]	The device associated with *spec* is currently mounted.
[EBUSY]	There are no more mount table entries.
[ENOENT]	*Spec* or *dir* is null.
[EACCES]	A component of the path prefix denies search permission.
[ENAMETOOLONG]	Any of the named files exceeds MAXPATHLEN characters.

WARNINGS

If *mount* is called from the program level (i.e. not called from *mount*(1M)), the table of mounted devices contained in **/etc/mnttab** is not updated.

HARDWARE DEPENDENCIES

Integral PC
    Normal users have all super-user capabilities.

SEE ALSO

mount(1M), umount(2).

NAME

msgctl – message control operations

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (msqid, cmd, buf)
int msqid, cmd;
struct msqid_ds *buf;

DESCRIPTION

Msgctl provides a variety of message control operations as specified by *cmd*. The following *cmds* are available:

IPC_STAT Place the current value of each member of the data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in the *glossary*.

IPC_SET Set the value of the following members of the data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*:

msg_perm.uid
msg_perm.gid
msg_perm.mode /* only low 9 bits */
msg_qbytes

This *cmd* can only be executed by a process that has an effective user ID equal to either that of super user or to the value of either **msg_perm.uid** or **msg_perm.cuid** in the data structure associated with *msqid*. Only super user can raise the value of **msg_qbytes**.

IPC_RMID
Remove the message queue identifier specified by *msqid* from the system and destroy the message queue and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of super-user or to the value of either **msg_perm.uid** or **msg_perm.cuid** in the data structure associated with *msqid*.

ERRORS

Msgctl will fail if one or more of the following are true:

[EINVAL]	*Msqid* is not a valid message queue identifier.
[EINVAL]	*Cmd* is not a valid command.
[EACCES]	*Cmd* is equal to **IPC_STAT** and {READ} operation permission is denied to the calling process (see the *glossary).*
[EPERM]	*Cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of super-user and it is not equal to the value of either **msg_perm.uid** or **msg_perm.cuid** in the data structure associated with *msqid*.
[EPERM]	*Cmd* is equal to **IPC_SET,** an attempt is being made to increase to the value of **msg_qbytes,** and the effective user ID of the calling process is not equal to that of super user.
[EFAULT]	*Buf* points to an illegal address. The reliable detection of this error will be implementation dependent.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

msgget(2), msgop(2), stdipc(3C).

NAME

msgget – get message queue

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget (key, msgflg)
key_t key;
int msgflg;
```

DESCRIPTION

Msgget returns the message queue identifier associated with *key*.

A message queue identifier and associated message queue and data structure are created for *key* if one of the following are true:

> *Key* is equal to **IPC_PRIVATE**.
>
> *Key* does not already have a message queue identifier associated with it, and (*msgflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

> **Msg_perm.cuid**, **msg_perm.uid**, **msg_perm.cgid**, and **msg_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.
>
> The low-order 9 bits of **msg_perm.mode** are set equal to the low-order 9 bits of *msgflg*.
>
> **Msg_qnum**, **msg_lspid**, **msg_lrpid**, **msg_stime**, and **msg_rtime** are set equal to 0.
>
> **Msg_ctime** is set equal to the current time.
>
> **Msg_qbytes** is set equal to the system limit.

ERRORS

Msgget will fail if one or more of the following are true:

[EACCES]	A message queue identifier exists for *key*, but operation permission as specified by the low-order 9 bits of *msgflg* would not be granted.
[ENOENT]	A message queue identifier does not exist for *key* and (*msgflg* & **IPC_CREAT**) is "false".
[ENOSPC]	A message queue identifier is to be created but the system-imposed limit on the maximum number of allowed message queue identifiers system wide would be exceeded.
[EEXIST]	A message queue identifier exists for *key* but ((*msgflg* & IPC_CREAT) **&** (*msgflg* & **IPC_EXCL**)) is "true".

RETURN VALUE

Upon successful completion, a non-negative integer, namely a message queue identifier, is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

msgctl(2), msgop(2), stdipc(3C).

NAME

msgop – message operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (msqid, msgp, msgsz, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz, msgflg;

int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz;
long msgtyp;
int msgflg;
```

MARKETING MODEL

Level A
Level B
Level C

TECHNICAL MODEL

Core System Extended
SVID

DESCRIPTION

Msgsnd is used to send a message to the queue associated with the message queue identifier specified by *msqid*. *Msgp* points to a structure containing the message. This structure is composed of the following members:

```
long    mtype;     /* message type */
char    mtext[];   /* message text */
```

Mtype is a positive integer that can be used by the receiving process for message selection (see *msgrcv* below). *Mtext* is any text of length *msgsz* bytes. *Msgsz* can range from 0 to a system-imposed maximum.

Msgflg specifies the action to be taken if one or more of the following are true:

The number of bytes already on the queue is equal to **msg_qbytes.**

The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

If (*msgflg* & **IPC_NOWAIT**) is "true", the message will not be sent and the calling process will return immediately.

If (*msgflg* & **IPC_NOWAIT**) is "false", the calling process will suspend execution until one of the following occurs:

The condition responsible for the suspension no longer exists, in which case the message is sent.

Msqid is removed from the system (see *msgctl*(2)). When this occurs, *errno* is set equal to EIDRM, and a value of –1 is returned.

The calling process receives a signal that is to be caught. In this case the message is not sent and the calling process resumes execution in the manner prescribed in *signal*(2).

Msgrcv reads a message from the queue associated with the message queue identifier specified by *msqid* and places it in the structure pointed to by *msgp*. This structure is composed of the following members:

```
long    mtype;      /* message type */
char    mtext[];    /* message text */
```

Mtype is the received message's type as specified by the sending process. *Mtext* is the text of the message. *Msgsz* specifies the size in bytes of *mtext*. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* & **MSG_NOERROR**) is "true". The truncated part of the message is lost and no indication of the truncation is given to the calling process.

Msgtyp specifies the type of message requested as follows:

If *msgtyp* is equal to 0, the first message on the queue is received.

If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.

If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

Msgflg specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

If (*msgflg* & **IPC_NOWAIT**) is "true", the calling process will return immediately with a return value of -1 and *errno* set to ENOMSG.

If (*msgflg* & **IPC_NOWAIT**) is "false", the calling process will suspend execution until one of the following occurs:

A message of the desired type is placed on the queue.

Msqid is removed from the system. When this occurs, *errno* is set equal to EIDRM, and a value of -1 is returned.

The calling process receives a signal that is to be caught. In this case a message is not received and the calling process resumes execution in the manner prescribed in *signal*(2)).

ERRORS

Msgsnd will fail and no message will be sent if one or more of the following are true:

[EINVAL]	*Msqid* is not a valid message queue identifier.
[EACCES]	Operation permission is denied to the calling process.
[EINVAL]	*Mtype* is less than 1.
[EAGAIN]	The message cannot be sent for one of the reasons cited above and (*msgflg* & **IPC_NOWAIT**) is "true".
[EINVAL]	*Msgsz* is less than zero or greater than the system-imposed limit.
[EFAULT]	*Msgp* points to an illegal address. The reliable detection of this error will be implementation dependent.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid*.

Msg_qnum is incremented by 1.

Msg_lspid is set equal to the process ID of the calling process.

Msg_stime is set equal to the current time.

Msgrcv will fail and no message will be received if one or more of the following are true:

[EINVAL]	*Msqid* is not a valid message queue identifier.
[EACCES]	Operation permission is denied to the calling process.
[EINVAL]	*Msgsz* is less than 0.
[E2BIG]	Mtext is greater than *msgsz* and (*msgflg* & **MSG_NOERROR**) is "false".
[ENOMSG]	The queue does not contain a message of the desired type and (*msgtyp* & **IPC_NOWAIT**) is "true".
[EFAULT]	*Msgp* points to an illegal address. The reliable detection of this error will be implementation dependent.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid*.

Msg_qnum is decremented by 1.

Msg_lrpid is set equal to the process ID of the calling process.

Msg_rtime is set equal to the current time.

RETURN VALUES

If *msgsnd* or *msgrcv* return due to the receipt of a signal, a value of -1 is returned to the calling process and *errno* is set to EINTR. If they return due to removal of *msqid* from the system, a value of -1 is returned and *errno* is set to EIDRM.

Upon successful completion, the return value is as follows:

Msgsnd returns a value of 0.

Msgrcv returns a value equal to the number of bytes actually placed into *mtext*.

Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

WARNING

Check all references to *signal*(2) for appropriateness on systems that support *sigvector*(2). *Sigvector*(2) can affect the behavior described on this page.

SEE ALSO

msgctl(2), msgget(2), signal(2), stdipc(3C).

NAME

nice – change priority of a process

SYNOPSIS

int nice (incr)
int incr;

DESCRIPTION

Nice adds the value of *incr* to the nice value of the calling process. A process's *nice value* is a positive number for which a more positive value results in lower CPU priority.

A maximum nice value of 39 and a minimum nice value of 0 are imposed by the system. Requests for values above or below these limits result in the nice value being set to the corresponding limit.

RETURN VALUE

Upon successful completion, *nice* returns the new nice value minus 20. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

Note that *nice* assumes a user process priority value of 20. If the super-user of your system has changed the user process priority value to something less than 20, certain increments can cause *nice* to return –1, which is indistinguishable from an error return.

ERRORS

[EPERM]	*Nice* will fail and not change the nice value if *incr* is negative or greater than 40 and the effective user ID of the calling process is not super-user.

HARDWARE DEPENDENCIES

Integral PC

Normal users have all super-user capabilities.

SEE ALSO

nice(1), exec(2).

NAME

open – open file for reading or writing

SYNOPSIS

#include <fcntl.h>
int open (path, oflag [, mode])
char *path;
int oflag, mode;

DESCRIPTION

Path points to a path name naming a file; it may not exceed 1024 bytes in length. *Open* opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. *Oflag* values are constructed by OR-ing flags from the list below.

Note that exactly one of the first three flags below must be used. Several of the other flags can be changed during the time the file is open using *fcntl*. See *fcntl*(2) and *fcntl*(5) for details.

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY This flag may affect subsequent reads and writes. See *read*(2) and *write*(2).

When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NDELAY is set:

An *open* for reading-only will return without delay. An *open* for writing-only will return an error if no process currently has the file open for reading.

If O_NDELAY is clear:

An *open* for reading-only will block until a process opens the file for writing. An *open* for writing-only will block until a process opens the file for reading.

When opening a file associated with a communication line:

If O_NDELAY is set:

The open will return without waiting for carrier.

If O_NDELAY is clear:

The open will block until carrier is present.

O_APPEND
If set, the file pointer will be set to the end of the file prior to each write.

O_CREAT
If the file exists, this flag has no effect. Otherwise, the owner ID of the file is set to the effective user ID of the process, the group ID of the file is set to the effective group ID of the process, and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows (see *creat*(2)):

All bits set in the file mode creation mask of the process are cleared. See *umask*(2).

The "save text image after execution", set-user-id and set-group-id bits of the mode is cleared. See *chmod*(2).

O_TRUNC
If the file exists, its length is truncated to 0 and the mode and owner are unchanged.

O_EXCL
If O_EXCL and O_CREAT are set, *open* will fail if the file exists.

O_SYNCIO
If a file is opened with O_SYNCIO or is set with the F_SETFL option of *fcntl*, file system writes for that file will be done through the cache to the disk as soon as possible, and the process will block until this is completed. This flag is ignored by all I/O calls except *write*, and is ignored for files other than ordinary files and block special devices on those systems which permit I/O to block special devices.

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across *exec* system calls, see *fcntl*(2).

EXAMPLES

The following call to *open* opens file *inputfile* for reading only and returns a file descriptor for *inputfile*. For an example of reading from file *inputfile*, see the *read*(2) manual page.

```
int myfd;

          myfd = open ("inputfile", O_RDONLY);
```

The following call to *open* opens file *outputfile* for writing and returns a file descriptor for *outputfile*. For an example of preallocating disk space for *outputfile*, see the *prealloc*(2) manual page. For an example of writing to *outputfile*, see the *write*(2) manual page.

```
int outfd;

          outfd = open ("outputfile", O_WRONLY);
```

RETURN VALUE

Upon successful completion, the file descriptor is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

Open will fail and the file will not be opened if one of the following conditions is true. **Errno** will be set accordingly:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	O_CREAT is not set.
[ENOENT]	The named file does not exist (for example, *path* is null or a component of *path* does not exist).
[EACCES]	A component of the path prefix denies search permission.
[EACCES]	*Oflag* permission is denied for the named file.
[EISDIR]	The named file is a directory and *oflag* is write or read/write.
[EROFS]	The named file resides on a read-only file system and *oflag* is write or read/write.
[EMFILE]	The maximum number of file descriptors allowed are currently open.
[ENXIO]	The named file is a character special or block special file, and the device associated with this special file does not exist.

[ENXIO]	O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.
[ETXTBSY]	The file is open for execution and *oflag* is write or read/write. Normal executable files are only open for a short time when they start execution. Other executable file types may be kept open for a long time, or indefinitely under some circumstances. The conditions are described in HARDWARE DEPENDENCIES below.
[EEXIST]	O_CREAT and O_EXCL are set, and the named file exists.
[EINTR]	A signal was caught during the *open* system call, and the system call was not restarted (see *signal*(2) and *sigvector*(2)).
[ENFILE]	The system file table is full.
[EAGAIN]	One or more segments of a pre-existing file have been locked with *lockf* or *fcntl* by some other process, and **O_TRUNC** is set.
[EAGAIN]	The file exists, enforcement mode file/record locking is set, and there are outstanding record locks on the file (see *chmod*(2)).
[EFAULT]	*Path* points outside the allocated address space of the process.
[EINVAL]	*Oflag* specifies both O_WRONLY and O_RDWR.
[ENAMETOOLONG]	The path specified exceeds MAXPATHLEN characters.

HARDWARE DEPENDENCIES

Series 500

Execute and write access are mutually exclusive.

Shared program files remain open for execution as long as there exists a process executing the program.

Once a shared program file with its sticky bit set has been loaded, it appears to be open indefinitely, even if the actual number of processes executing the program drops to zero. Refer to the system administrator's manual for a description of the sticky bit.

Demand loaded program files that are not shared remain open until all of the code and data have been loaded. Then they are closed.

AUTHOR

Open was developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

chmod(2), close(2), creat(2), dup(2), fcntl(2), lseek(2), read(2), select(2), umask(2), write(2), lockf(2).

NAME

pause – suspend process until signal

SYNOPSIS

pause ( )

DESCRIPTION

Pause suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored or blocked (masked) by the calling process.

If the signal causes termination of the calling process, *pause* will not return.

If the signal is *caught* by the calling process and control is returned from the signal-catching function (see *signal*(2)), the calling process resumes execution from the point of suspension; with a return value of –1 from *pause* and *errno* set to EINTR.

WARNING

Check all references to *signal*(2) for appropriateness on systems that support *sigvector*(2). *Sigvector*(2) can affect the behavior described on this page.

SEE ALSO

alarm(2), kill(2), signal(2), sigvector(2), wait(2).

NAME

pipe – create an interprocess channel

SYNOPSIS

int pipe (fildes)
int fildes[2];

DESCRIPTION

Pipe creates an I/O mechanism called a pipe and returns two file descriptors, *fildes*[0] and *fildes*[1]. *Fildes*[0] is opened for reading and *fildes*[1] is opened for writing.

Writes up to 5120 bytes of data are buffered by the pipe before the writing process is blocked. A read only file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out (FIFO) basis.

EXAMPLES

The following example uses *pipe* to implement the command string "ls | sort":

```
int pid;
int pipefd[2];

        /* Assumes file descriptor 0 and 1 are open  */
        pipe (pipefd);

        if ((pid = fork()) == 0) {
                close(1);        /* close stdout */
                dup (pipefd[1]);
                execlp ("ls", "ls", 0);
        }
        else if (pid > 0) {
                close(0);/* close stdin  */
                dup (pipefd[0]);
                execlp ("sort", "sort", 0);
        }
```

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

Pipe will fail if one or more are true:

[EMFILE]	NFILE - 1 or more file descriptors are currently open.
[ENFILE]	The system file table is full.
[ENOSPC]	Not enough space on file system.

HARDWARE DEPENDENCIES

Integral PC

Writes of up to 10240 bytes of data are buffered by the pipe before the writing process is blocked.

SEE ALSO

sh(1), read(2), write(2), popen(3S).

NAME

plock – lock process, text, or data in memory

SYNOPSIS

```
#include <sys/lock.h>

int plock (op)
int op;
```

DESCRIPTION

Plock allows the calling process to lock the text segment of the process (text lock), its data segment (data lock), or both its text and data segment (process lock) into memory. Locked segments are immune to all routine swapping. *Plock* also allows these segments to be unlocked. To use this call, the calling process must be a member of a privilege group allowing ***plock*** (see ***setprivgrp*** on *getprivgrp*(2)) or the effective user ID of the calling process must be super-user. *Op* specifies the following:

PROCLOCK	lock text and data segments into memory (process lock)
TXTLOCK	lock text segment into memory (text lock)
DATLOCK	lock data segment into memory (data lock)
UNLOCK	remove locks

EXAMPLES

The following call to *plock* locks the calling process in memory:

```
plock (PROCLOCK);
```

RETURN VALUE

Upon successful completion, a value of 0 is returned to the calling process. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

Plock will fail and not perform the requested operation if one or more of the following are true:

[EPERM]	The effective user ID of the calling process is not super-user and the user does not have PRIV_MLOCK.
[EINVAL]	*Op* is equal to PROCLOCK and a process lock, a text lock, or a data lock already exists on the calling process.
[EINVAL]	*Op* is equal to TXTLOCK and a text lock, or a process lock already exists on the calling process.
[EINVAL]	*Op* is equal to DATLOCK and a data lock, or a process lock already exists on the calling process.
[EINVAL]	*Op* is equal to UNLOCK and no type of lock exists on the calling process.
[EINVAL]	*Op* is not equal to either PROCLOCK, TXTLOCK, DATLOCK, or UNLOCK.
[EINVAL]	*Plock* not allowed in [vfork, exec] window (see *vfork*(2)).

SEE ALSO

exec(2), exit(2), fork(2).

NAME

prealloc – preallocate fast disk storage

SYNOPSIS

int prealloc (fildes, size)
int fildes;
unsigned size;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup* or *fcntl* system call for an ordinary file of zero length. *Size* is the size in bytes to be preallocated for the file specified by *fildes*, at least *size* bytes will be allocated. The space will be allocated in an implementation dependent fashion for fast sequential reads and writes. The EOF in an extended file will be left at the end of the preallocated area. The current file pointer is left at zero. The file is zero-filled.

Using *prealloc* on a file does **not** give the file an attribute which is inherited when copying or restoring the file using a program like *cp*(1) or *tar*(1). It simply guarantees that the disk space has been preallocated for *size* bytes in a manner suited for sequential access. The file can be extended beyond these limits by *write* operations past the original end of file, however this space will be not necessarily be allocated using any special strategy.

EXAMPLES

Assuming a process opened a file for writing, the following call to *prealloc* preallocates at least 50000 bytes on disk for the file represented by file descriptor *outfd*:

```
prealloc (outfd, 50000);
```

HARDWARE DEPENDENCIES

As the exact effect, and performance benefits, to be obtained by using this call vary with the implementation of the file system, the performance related details are described in the *System Administrator's Manual* for each specific machine.

ERRORS

Prealloc will fail and no disk space will be allocated if one or more of the following are true:

[EBADF]	*Fildes* is not a valid open file descriptor.
[ENOTEMPTY]	*Fildes* not associated with an ordinary file of zero length.
[ENOSPC]	Not enough space left on device to allocate the requested amount; no space was allocated.
[EFBIG]	*Size* exceeds the maximum file size or the process's file size limit. See *ulimit*(2). Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

AUTHOR

Prealloc was developed by the Hewlett-Packard Company.

SEE ALSO

prealloc(1), creat(2), dup(2), fcntl(2), open(2), read(2), ulimit(2), write(2).

BUGS

The allocation of the file space is highly dependent on the current disk usage. A successful return does not tell you how fragmented the file actually might be if the disk is reaching its capacity.

NAME

profil – execution time profile

SYNOPSIS

#include <sys/param.h>

void profil (buff, bufsiz, offset, scale)
char *buff;
int bufsiz, offset, scale;

DESCRIPTION

Buff points to an area of core whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick, *offset* is subtracted from it, and the result is multiplied by *scale*. If the resulting number corresponds to a word inside *buff*, that word is incremented. The number of samples per second for a given implementation is given by HZ as found in **<sys/param.h>**

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 0177777 (octal) gives a 1-1 mapping of pc's to words in *buff*; 077777 (octal) maps each pair of instruction words together. 02(octal) maps all instructions onto the beginning of *buff* (producing a non-interrupting core clock).

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when an *exec* is executed, but remains on in child and parent both after a *fork*. Profiling will be turned off if an update in *buff* would cause a memory fault.

RETURN VALUE

Not defined.

SEE ALSO

prof(1), monitor(3C).

NAME

ptrace – process trace

SYNOPSIS

int ptrace (request, pid, addr, data);
int request, pid, addr, data;

MARKETING MODEL

Level A
Level B
Level C

TECHNICAL MODEL

Core System Extended
SVID

REMARKS

Much of the functionality of this capability is highly dependent on the underlying hardware. An application which uses this system call should not be expected to be portable across architectures or implementations.

DESCRIPTION

Ptrace provides a means by which a parent process may control the execution of a child process. Its primary use is for the implementation of breakpoint debugging; see *adb*(1). The child process behaves normally until it encounters a signal (see *signal*(2) for the list), at which time it enters a stopped state and its parent is notified via *wait*(2). When the child is in the stopped state, its parent can examine and modify its "core image" using *ptrace*. Also, the parent can cause the child either to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

The *request* argument determines the precise action to be taken by *ptrace* and is one of the following:

0 This request must be issued by the child process if it is to be traced by its parent. It turns on the child's trace flag that stipulates that the child should be left in a stopped state upon receipt of a signal rather than the state specified by *func*; see *signal*(2). The *pid*, *addr*, and *data* arguments are ignored, and a return value is not defined for this request. Peculiar results will ensue if the parent does not expect to trace the child.

The remainder of the requests can only be used by the parent process. For each, *pid* is the process ID of the child. The child must be in a stopped state before these requests are made.

1, 2 With these requests, the word at location *addr* in the address space of the child is returned to the parent process. If instruction (I) and data (D) space are separated, request **1** returns a word from I space, and request **2** returns a word from D space. If I and D space are not separated, either request **1** or request **2** may be used with equal results. The *data* argument is ignored. These two requests will fail if *addr* is not the start address of a word, in which case a value of –1 is returned to the parent process and the parent's *errno* is set to EIO.

3 With this request, the word at location *addr* in the child's USER area in the system's address space (see **<sys/user.h>**) is returned to the parent process. Addresses in this area are system dependent, but start at zero. The limit can be derived from **<sys/user.h>**. The *data* argument is ignored. This request will fail if *addr* is not the start address of a word or is outside the USER area, in which case a value of –1 is returned to the parent process and the parent's *errno* is set to EIO.

4, 5 With these requests, the value given by the *data* argument is written into the address space of the child at location *addr*. Request 4 writes a word into I space, and request 5 writes a word in D space. Upon successful completion, the value written into the address space of the child is returned to the parent. These two requests will fail if *addr* is not the start address of a word, or if *addr* is a location in a pure procedure space and either another process is executing in that space or the parent process does not have write access for the executable file corresponding to that space. Upon failure a value of -1 is returned to the parent process and the parent's *errno* is set to EIO.

6 With this request, a few entries in the child's USER area can be written. *Data* gives the value that is to be written and *addr* is the location of the entry. The few entries that can be written are dependent on the architecture of the system, but include the user data registers, auxiliary data registers, and status register (the set of registers, or bits in registers, which the user's program could modify).

7 This request causes the child to resume execution. If the *data* argument is 0, all pending signals including the one that caused the child to stop are canceled before it resumes execution. If the *data* argument is a valid signal number, the child resumes execution as if it had incurred that signal, and any other pending signals are canceled. The *addr* argument must be equal to 1 for this request. Upon successful completion, the value of *data* is returned to the parent. This request will fail if *data* is not 0 or a valid signal number, in which case a value of -1 is returned to the parent process and the parent's *errno* is set to EIO.

8 This request causes the child to terminate with the same consequences as *exit*(2).

9 This request causes a flag to be set so that an interrupt will occur upon the completion of one machine instruction, and then executes the same steps as listed above for request **7**. If the processor does not provide a trace bit, this request returns an error. This effectively allows single stepping of the child.

Whether or not the trace bit remains set after this interrupt is a function of the hardware.

To forestall possible fraud, *ptrace* inhibits the set-user-id facility on subsequent *exec*(2) calls. If a traced process calls *exec*, it will stop before executing the first instruction of the new image showing signal **SIGTRAP**.

ERRORS

Ptrace will in general fail if one or more of the following are true:

[EIO] *Request* is an illegal number.

[ESRCH] *Pid* identifies a child that does not exist or has not executed a *ptrace* with request **0**.

HARDWARE DEPENDENCIES

Series 800

Request **6** is not supported and will return -1 with errno set to EIO.

Two other requests are available:

10 With this request, the word at location *addr* in the save_state structure at the base of the per-process kernel stack is returned to the parent process. *Addr* must be word-aligned and less than STACKSIZE*NBPG (see **<sys/param.h>** and **<machine/param.h>**). The save_state structure contains the registers and other information about the process.

11 The save_state structure at the base of the per-process kernel stack is written, as it is read with request **10**. Only a few locations can be written in this way: the general registers, most floating point registers, a few control registers, and certain bits of the interruption processor status word.

SEE ALSO

adb(1), exec(2), signal(2), wait(2).

NAME

read, readv – read input

SYNOPSIS

```
int read (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

#include <sys/types.h>
#include <sys/uio.h>

int readv (fildes, iov, iovcnt)
int fildes;
struct iovec *iov;
int iovcnt;
```

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

Read attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*. *Readv* performs the same action but scatters the input data into the *iovcnt* buffers specified by the elements of the *iovec* array: iov[0], iov[1], ..., iov[*iovcnt* - 1].

For *readv* the iovec structure is defined as:

```
struct iovec {
     caddr_t             iov_base;
     unsigned iov_len;
};
```

Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. *Readv* will always fill one area completely before proceeding to the next area. The *iovec* array may be at most *MAXIOV* long.

On devices capable of seeking, the *read* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *read*, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a device is undefined.

Upon successful completion, *read* returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if 1) the file is associated with a communication line (see *ioctl*(2) and *termio*(7)), or 2) if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached.

When attempting to read from an ordinary file with enforcement-mode file and record locking set (see *chmod*(2)), and the segment of the file to be read has a blocking write lock (i.e. a write lock owned by another process):

> If O_NDELAY is set, the function read will return –1 and errno will be set to EAGAIN.
>
> If O_NDELAY is clear, the function read will sleep until the blocking write lock is removed.

When attempting to read from an empty pipe (or FIFO):

> If O_NDELAY is set, the read will return a 0.
>
> If O_NDELAY is clear, the read will block until data is written to the file or the file is no longer open for writing.

When attempting to read a file associated with a tty that has no data currently available:

If O_NDELAY is set, the read will return a 0.

If O_NDELAY is clear, the read will block until data becomes available.

RETURN VALUE

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. Otherwise, a −1 is returned and *errno* is set to indicate the error.

EXAMPLES

Assuming a process opened a file for reading, the following call to *read*(2) reads *BUFSIZ* bytes from the file into the buffer pointed to by *mybuf*.

```
#include <stdio.h>    /*include this for BUFSIZ definition*/

char mybuf[BUFSIZ];
int nbytes;

                 nbytes = read (myfd, mybuf, BUFSIZ);
```

ERRORS

Read will fail if one of the following conditions is true and errno will be set accordingly:

[EBADF]	*Fildes* is not a valid file descriptor open for reading.
[EINTR]	A signal was caught during the *read* system call.
[EAGAIN]	Enforcement-mode file and record locking was set, O_NDELAY was set, and there was a blocking write lock.
[EDEADLK]	A resource deadlock would occur as a result of this operation (see *lockf*(2) and *fcntl*(2)).
[EFAULT]	*Buf* points outside the allocated address space. The reliable detection of this error will be implementation-dependent.
[ENOLCK]	The system record lock table was full, so the read could not go to sleep until the blocking write lock was removed.

In addition, *readv* may return one of the following errors:

[EFAULT]	*Iov_base* or *iov* points outside of the allocated address space. The reliable detection of this error will be implementation dependent.
[EINVAL]	*Iovcnt* was less then or equal to 0, or greater than *MAXIOV*.
[EINVAL]	The sum of iov len values in the iov array overflowed a 32-bit integer.

WARNINGS

Record locking may or may not be enforced by the system depending on the setting of the file's mode bits (see *lockf*(2)).

The character-special devices, and raw disks in particular, apply constraints on how *read* can be used. See the specific Section (7) entries for details on particular devices.

Check all references to *signal*(2) for appropriateness on systems that support *sigvector*(2). *Sigvector*(2) can affect the behavior described on this page.

HARDWARE DEPENDENCIES

Series 500

In general, a value of *nbyte* greater than 512K is not supported when *fildes* is associated with a device file. There are two exceptions to this:

the device is a terminal or the null device; or

buf points to a local (not global) buffer, and has been locked with *memlck*(2). A local buffer is an array that is declared within the procedure and resides on the stack.

Any request for greater than 512K megabytes on unsupported device files results in errno being set to EINVAL. Requests for less than 512K megabytes could result in errno being set to ENOMEM.

Series 500, Integral PC
Readv is not currently supported.

AUTHOR

Read was developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

creat(2), dup(2), fcntl(2), ioctl(2), lockf(2), open(2), pipe(2), select(2), ustat(2), tty(7).

NAME

reboot – boot the system

SYNOPSIS

#include <sys/reboot.h>
int reboot (howto, device_file, filename)
int howto; char *device_file; char *filename;

MARKETING MODEL

Level C

TECHNICAL MODEL

LM
HP+

DESCRIPTION

Reboot causes the system to be rebooted. *Howto* is a mask of reboot options (see <sys/reboot.h>). Only RB_HALT, RB_AUTOBOOT, RB_NOSYNC, RB_NEWDEVICE, and RB_NEWFILE are recognized options.

The *howto* options are:

RB_AUTOBOOT
: A filesystem sync is performed (unless RB_NOSYNC is set) and the processor is rebooted from the default device and file.

RB_HALT
: The processor is simply halted. A sync of the filesystem will be done unless the RB_NOSYNC flag is set. RB_HALT should be used with caution.

RB_NOSYNC
: A sync of the filesystem is not to be performed.

RB_NEWDEVICE
: The *device_file* argument to the system call is to be used as the filename of the device from which to reboot.

RB_NEWFILE
: The *filename* argument to the system call is to be used as the name of the file to be rebooted.

Device_file specifies the device from which the reboot is to take place. *Device_file* must be a block or character special file name and is used only if the RB_NEWDEVICE option is set.

Filename specifies the name of the file to be rebooted (only used if the RB_NEWFILE option is set). This file will be loaded into memory by the bootstrap and control passed to it.

Only the super-user may *reboot* a machine.

ERRORS

Reboot will fail if the following is true:

[EPERM] The effective user id of the caller is not super-user.

HARDWARE DEPENDENCIES

Series 300

Filename must be one of the files listed by the boot rom at power up.

The default device and file for RB_AUTOBOOT are those from which the system was previously booted.

Series 800

The RB_NEWDEVICE and RB_NEWFILE options and the *device_file* and *filename* parameters are ignored.

The default file and device for RB_AUTOBOOT are /hp-ux on the current root device.

RETURN VALUE

If successful, this call never returns. Otherwise, a −1 is returned and an error is returned in the global variable *errno*.

AUTHOR

Reboot was developed by HP, and the University of California, Berkeley.

SEE ALSO

reboot(1M).

NAME

rmdir – remove a directory file

SYNOPSIS

rmdir(path)
char *path;

REMARKS

Not all systems implement this as a system call; some use a library call to the *rmdir*(1) command to achieve the same effect. The errors documented below will appear in any case, and no error messages will ever be printed.

DESCRIPTION

Rmdir removes a directory file whose name is given by *path*. The directory must be empty (except for files "." and "..") before it can be removed.

RETURN VALUE

A 0 is returned if the remove succeeds. Otherwise a −1 is returned and an error code is stored in the global location *errno*.

ERRORS

The named file is removed unless one or more of the following are true:

[EACCES]	A component of the path prefix denies search permission.
[EACCES]	Write permission is denied on the directory containing the link to be removed.
[EBUSY]	The directory to be removed is the mount point for a mounted file system.
[EFAULT]	*Path* points outside the process's allocated address space. The reliable detection of this error will be implementation dependent.
[ENAMETOOLONG]	The path name exceeds MAXPATHLEN characters.
[ENOENT]	A component of the path name is too long.
[ENOENT]	The named file does not exist.
[ENOTDIR]	A component of the path is not a directory.
[ENOTEMPTY]	The named directory is not empty. It contains files other than "." and "..".
[EROFS]	The directory entry to be removed resides on a read-only file system.

HARDWARE DEPENDENCIES

Series 500:

The directory entries "." and ".." are recognized, but files of the same names do not appear in the *dir* structure.

AUTHOR

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

SEE ALSO

mkdir(2), unlink(2).

NAME

rtprio – change or read realtime priority

SYNOPSIS

#include <sys/rtprio.h>

rtprio (pid, prio)
int pid, prio;

DESCRIPTION

Rtprio is used to set or read the realtime priority of a process. If *pid* is zero, it names the calling process; otherwise it gives the pid of the process. When setting the realtime priority of another process, the real or effective user ID of the calling process must match the real or saved user ID of the process to be modified, or the effective user ID of the calling process must be that of super-user. The calling process must also be a member of a privilege group allowing *rtprio* (see *getprivgrp*(2)) or the effective user ID of the calling process must be super-user. Simply reading real-time priorities requires no special privilege.

Real time scheduling policies differ from the normal timesharing policies in that the realtime priority is used to absolutely order all realtime processes; this priority is not degraded over time. All realtime processes are of higher priority than normal user processes, although some system processes may run at realtime priorities themselves. If there are several eligible processes at the same priority level, they will be run in a round robin fashion as long as no process with higher priority intercedes. A realtime process will receive cpu service until it either voluntarily gives up the cpu or is preempted by a process of equal or higher priority. Interrupts may also preempt a realtime process.

Valid realtime priorities run from zero to 127. Zero is the highest (most important) priority. This realtime priority is inherited across *forks* and *execs*.

Prio specifies the following:

0–127	Set process to this realtime priority.
RTPRIO_NOCHG	Do not change realtime priority. This is used for reading the process realtime priority.
RTPRIO_RTOFF	Set this process to no longer have a realtime priority. It will resume a normal timesharing priority. Any process, regardless of privilege, is allowed to turn off its own realtime priority using a *pid* of zero.

EXAMPLES

The following call to *rtprio* sets the calling process to a real-time priority of 90:

```
rtprio (0, 90);
```

RETURN VALUE

If no error occurs, *rtprio* will return the *pid*'s former (before the call) realtime priority. If the process was not a realtime process, RTPRIO_RTOFF will be returned. If an error does occur, -1 is returned and *errno* is set to one of the values described in the ERRORS section.

ERRORS

[EINVAL]	*Prio* is not RTPRIO_NOCHG, RTPRIO_RTOFF, or in the range of 0 to 127.
[EPERM]	The calling process is not the super–user and neither its real or effective user–id match the real or saved user–id of the process indicated by *pid.*

[EPERM]	The calling process is not a member with a of a group having PRIV_RTPRIO capability and *prio* is not RTPRIO_NOCHG, or RTPRIO_RTOFF with a *pid* of zero.
[ESRCH]	No process can be found corresponding to that specified by *pid.*

HARDWARE DEPENDENCIES

Series 500

Some of the work done by the system on behalf of users is done with daemon processes which have various priorities. Some functions such as copying user space on a fork, virtual memory swapping, and LAN activity are done at a priority lower than any of the *rtprio*(2) priorities.

Other functions, such as terminal I/O, disk I/O, DIL interrupts, signals, *select*(2) wakeups, and system timers, are done at a priority level equivalent to an *rtprio*(2) priority of 64.

If there is a real-time process that is consuming all available CPU time, the system will not be able to accomplish any other system activities that have a lower priority, even if the function is on behalf of the real-time process. In the case of multi-CPU systems, it will take multiple real-time processes to lock out the system.

The user of *rtprio*(2) must decide whether the task requiring real-time priorities needs to have an effective priority greater than or less than the system functions provided.

Integral PC

The normal user may change the real-time priority of any process except that of the scheduler.

AUTHOR

Rtprio was developed by HP.

SEE ALSO

rtprio(1), getprivgrp(2), nice(2), plock(2).

WARNINGS

Normally, compute bound programs should not be run at realtime priorities, because all time sharing work on the cpu would come to a complete halt.

NAME

select – synchronous I/O multiplexing

SYNOPSIS

#include <time.h>

int select(nfds, readfds, writefds, exceptfds, timeout)
int nfds, *readfds, *writefds, *exceptfds;
struct timeval *timeout;

DESCRIPTION

Select examines the file descriptors specified by the bit masks *readfds*, *writefds* and *exceptfds*. The bits from 0 through *nfds*-1 are examined. File descriptor *f* is represented by the bit 1<<f in the masks. More formally, a file descriptor is represented by:

```
fds[(f / BITS_PER_INT)] & (1 << (f % BITS_PER_INT))
```

When *select* completes successfully it returns the three bit masks modified as follows: For each file descriptor less than *nfds*, the corresponding bit in each mask is set if the bit was set upon entry and the file descriptor is ready for reading, writing or has an exceptional condition pending.

If *timeout* is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a zero pointer, the select waits until an event causes one of the masks to be returned with a valid (non-zero) value. To poll, the *timeout* argument should be non-zero, pointing to a zero valued timeval structure. Specific implementations may place limitations on the maximum timeout interval supported. The constant *MAX_ALARM* defined in *<sys/param.h>* specifies the implementation-specific maximum (in seconds). Whenever *timeout* specifies a value greater than this maximum, it is silently rounded down to this maximum. On all implementations, *MAX_ALARM* is guaranteed to be at least 31 days (in seconds). Note that the use of a timeout does not affect any pending timers set up by *alarm*(2) or *setitimer*(2).

Any or all of *readfds*, *writefds*, and *exceptfds* may be given as 0 if no descriptors are of interest.

Ordinary files always select true whenever selecting on reads, writes, and/or exceptions.

EXAMPLES

The following call to *select* checks if any of 4 terminals are ready for reading. *Select* will time out after 5 seconds if no terminals are ready for reading. Note that the code for opening the terminals or reading from the terminals is not shown in this example. Also, note that this example must be modified if the calling process has more than 32 file descriptors open:

```
#define MASK(f)         (1 << (f))
#define NTTYS4

int tty[NTTYS];
int ttymask[NTTYS];
int readmask = 0;
int readfds;
int nfound, i;
struct timeval timeout;

        /* First open each terminal for reading and put the
        * file descriptors into array tty[NTTYS]. The code
        * for opening the terminals is not shown here.
        */

        for (i=0; i < NTTYS; i++) {
                ttymask[i] = MASK(tty[i]);
                readmask |= ttymask[i];
        }

        timeout.tv_sec  = 5;
        timeout.tv_usec = 0;
        readfds = readmask;

        /* select on NTTYS+3 file descriptors if stdin, stdout
        * and stderr are also open
        */
        if ((nfound = select (NTTYS+3, &readfds, 0, 0, &timeout)) == -1)
                perror ("select failed");
        else if (nfound == 0)
                printf ("select timed out \n");
        else for (i=0; i < NTTYS; i++)
                if (ttymask[i] & readfds)
                        /* Read from tty[i].  The code for reading
                        * is not shown here.
                        */
                else printf ("tty[%d] is not ready for reading \n",i);
```

RETURN VALUE

Select returns the number of descriptors contained in the bit masks, or −1 if an error occurred. If the time limit expires then *select* returns 0 and all the masks are cleared.

ERRORS

An error return from *select* indicates:

[EBADF]	One or more of the bit masks specified an invalid descriptor.
[EINTR]	A signal was delivered before any of the selected for events occurred or before the time limit expired.
[EFAULT]	One or more of the pointers was invalid. The reliable detection of this error will be implementation dependent.
[EINVAL]	Invalid timeval passed for timeout.
[EINVAL]	*nfds < 0*

WARNINGS

Check all references to *signal*(2) for appropriateness on systems that support *sigvector*(2). *Sigvector*(2) can affect the behavior described on this page.

The file descriptor masks are always modified on return, even if the call returns as the result of a timeout.

HARDWARE DEPENDENCIES

Series 200, 300

Select(2) supports the following devices and file types:

pipes
fifo special files (named pipes)
All serial interfaces
All ITEs and HP-HIL input devices
pty(7) special files
HP 98643 LAN interface card driver

File types not supporting *select*(2) always return true.

Series 500

Select(2) supports the following devices and file types:

pipes
fifo special files (named pipes)
pty(7) special files
Model 520 Internal Terminal Emulator (ITE)
HP 98700H ITE and HP-HIL input devices
(such as HP 46020A Keyboard and HP 46086A Button Box)
HP 27128A ASI tty driver
HP 27125A LAN interface card driver (LLA)
HP 27130A/B port MUX (with appropriate firmware revision)

Device files that do not support *select*(2) always return true for those conditions the user is selecting on.

Series 800

Select(2) supports the following devices and file types:

pipes
fifo special files (named pipes)
all serial devices
hpib(7) special files
gpio(7) special files
lan(7) special files
pty(7) special files

The convention for device files that do not support select(2) is to always return true for those conditions the user is selecting on.

Consult the individual device manual pages to determine the extent to which any particular driver supports select(2).

AUTHOR

Select was developed by HP and the University of California, Berkeley.

SEE ALSO

fcntl(2), read(2), write(2).

NAME

semctl – semaphore control operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (semid, semnum, cmd, arg)
int semid, semnum, cmd;
union semun {
     int val;
     struct semid_ds *buf;
     ushort *array;
} arg;
```

DESCRIPTION

Semctl provides a variety of semaphore control operations as specified by *cmd*.

The following *cmds* are executed with respect to the semaphore specified by *semid* and *semnum:*

GETVAL	Return the value of semval (see the *glossary).*
SETVAL	Set the value of semval to *arg.val*. When this cmd is successfully executed, the semadj value corresponding to the specified semaphore in all processes is cleared.
GETPID	Return the value of sempid.
GETNCNT	Return the value of semncnt.
GETZCNT	Return the value of semzcnt.

The following *cmds* return and set, respectively, every semval in the set of semaphores.

GETALL	Place semvals into array pointed to by *arg.array*.
SETALL	Set semvals according to the array pointed to by *arg.array*. When this cmd is successfully executed the semadj values corresponding to each specified semaphore in all processes are cleared.

The following *cmds* are also available:

IPC_STAT Place the current value of each member of the data structure associated with *semid* into the structure pointed to by *arg.buf*. The contents of this structure are defined in the *glossary.*

IPC_SET Set the value of the following members of the data structure associated with *semid* to the corresponding value found in the structure pointed to by *arg.buf*:

```
sem_perm.uid
sem_perm.gid
sem_perm.mode /* only low 9 bits */
```

This cmd can only be executed by a process that has an effective user ID equal to either that of super-user or to the value of either **sem_perm.uid** or **sem_perm.cuid** in the data structure associated with *semid*.

IPC_RMID Remove the semaphore identifier specified by *semid* from the system and destroy the set of semaphores and data structure associated with it. This cmd can only be executed by a process that has an effective user ID equal to either that of super-user or to the value of either **sem_perm.uid** or **sem_perm.cuid** in the data structure associated with *semid*.

EXAMPLES

The following call to *semctl* initializes the set of 4 semaphores to the values 0, 1, 0 and 1 respectively. This example assumes the process has a valid semid representing a set of 4 semaphores as shown on the *semget*(2) manual page. For an example of performing "P" and "V" operations on the semaphores below, refer to the *semop*(2) manual page.

```
ushort semarray[4];

        semarray[0] = 0;
        semarray[1] = 1;
        semarray[2] = 0;
        semarray[3] = 1;

        semctl (mysemid, 0, SETALL, semarray);
```

ERRORS

Semctl will fail if one or more of the following are true:

[EINVAL]	*Semid* is not a valid semaphore identifier.
[EINVAL]	*Semnum* is less than zero or greater than or equal **sem_nsems**.
[EINVAL]	*Cmd* is not a valid command.
[EACCES]	Operation permission is denied to the calling process (see the *glossary)*.
[ERANGE]	*Cmd* is **SETVAL** or **SETALL** and the value to which semval is to be set is greater than the system imposed maximum.
[EPERM]	*Cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of super-user and it is not equal to the value of either **sem_perm.uid** or **sem_perm.cuid** in the data structure associated with *semid*.
[EFAULT]	*Arg.buf* or *arg.array* points to an illegal address. The reliable detection of this error will be implementation dependent.

RETURN VALUE

Upon successful completion, the value returned depends on *cmd* as follows:

GETVAL	The value of semval.
GETNCNT	The value of semncnt.
GETZCNT	The value of semzcnt.
GETPID	The value of sempid.
All others	A value of 0.

Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

semget(2), semop(2), stdipc(3C).

NAME

shmget – get shared memory segment

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (key, size, shmflg)
key_t key;
int size, shmflg;

DESCRIPTION

Shmget returns the shared memory identifier associated with *key*.

A shared memory identifier and associated data structure and shared memory segment of size *size* bytes (see *glossary*) are created for *key* if one of the following are true:

Key is equal to **IPC_PRIVATE**.

Key does not already have a shared memory identifier associated with it, and (*shmflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

Shm_perm.cuid, **shm_perm.uid**, **shm_perm.cgid**, and **shm_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of **shm_perm.mode** are set equal to the low-order 9 bits of *shmflg*. **Shm_segsz** is set equal to the value of *size*.

Shm_lpid, **shm_nattch**, **shm_atime**, and **shm_dtime** are set equal to 0.

Shm_ctime is set equal to the current time.

EXAMPLES

The following call to *shmget* returns a unique shmid for the newly created shared memory segment of 4096 bytes:

```
int myshmid;

          myshmid = shmget (IPC_PRIVATE, 4096, 0600);
```

ERRORS

Shmget will fail if one or more of the following are true:

[EINVAL]	*Size* is less than the system-imposed minimum or greater than the system-imposed maximum.
[EACCES]	A shared memory identifier exists for *key* but operation permission (see *glossary*) as specified by the low-order 9 bits of *shmflg* would not be granted.
[EINVAL]	A shared memory identifier exists for *key* but the size of the segment associated with it is less than *size* and *size* is not equal to zero.
[ENOENT]	A shared memory identifier does not exist for *key* and (*shmflg* & **IPC_CREAT**) is "false".
[ENOSPC]	A shared memory identifier is to be created but the system-imposed limit on the maximum number of allowed shared memory identifiers system wide would be exceeded.

[ENOMEM]	A shared memory identifier and associated shared memory segment are to be created but the amount of available physical memory is not sufficient to fill the request.
[EEXIST]	A shared memory identifier exists for *key* but ((*shmflg* & IPC_CREAT) **&&** (*shmflg* & **IPC_EXCL**)) is "true".

HARDWARE DEPENDENCIES

Series 500

Shared memory segments larger than **16384** bytes are paged virtual segments; otherwise they are non-paged virtual segments.

RETURN VALUE

Upon successful completion, a non-negative integer, namely a shared memory identifier is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

shmctl(2), shmop(2), stdipc(3C).

NAME

semop – semaphore operations

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (semid, sops, nsops)
int semid;
struct sembuf *sops;
int nsops;

DESCRIPTION

Semop is used to atomically perform an array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by *semid*. *Sops* is a pointer to the array of semaphore-operation structures. *Nsops* is the number of such structures in the array. The contents of each structure includes the following members:

```
ushort   sem_num;   /* semaphore number */
short    sem_op;    /* semaphore operation */
short    sem_flg;   /* operation flags */
```

Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore specified by *semid* and *sem_num*. Semaphore array operations are atomic, in that none of the semaphore operations will be performed until blocking conditions on all of the semaphores in the array have been removed.

Sem_op specifies one of three semaphore operations as follows:

If *sem_op* is a negative integer, one of the following will occur:

If semval (see *intro*(2)) is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is subtracted from semval. Also, if (*sem_flg* & **SEM_UNDO**) is "true", the absolute value of *sem_op* is added to the calling process's semadj value (see *exit*(2)) for the specified semaphore.

If semval is less than the absolute value of *sem_op* and (*sem_flg* & **IPC_NOWAIT**) is "true", *semop* will return immediately.

If semval is less than the absolute value of *sem_op* and (*sem_flg* & **IPC_NOWAIT**) is "false", *semop* will increment the semncnt associated with the specified semaphore and suspend execution of the calling process until one of the following conditions occur:

Semval becomes greater than or equal to the absolute value of *sem_op*. When this occurs, the value of semncnt associated with the specified semaphore is decremented, the absolute value of *sem_op* is subtracted from semval and, if (*sem_flg* & **SEM_UNDO**) is "true", the absolute value of *sem_op* is added to the calling process's semadj value for the specified semaphore.

The semid for which the calling process is awaiting action is removed from the system (see *semctl*(2)). When this occurs, *errno* is set equal to EIDRM, and a value of -1 is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of semncnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in *signal*(2).

If *sem_op* is a positive integer, the value of *sem_op* is added to semval and, if (*sem_flg* & **SEM_UNDO**) is "true", the value of *sem_op* is subtracted from the calling process's semadj value for the specified semaphore.

If *sem_op* is zero, one of the following will occur:

> If semval is zero, *semop* will proceed to the next semaphore operation specified by *sops,* or return immediately if this is the last operation.
>
> If semval is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "true", *semop* will return immediately.
>
> If semval is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "false", *semop* will increment the semzcnt associated with the specified semaphore and suspend execution of the calling process until one of the following occurs:
>
> > Semval becomes zero, at which time the value of semzcnt associated with the specified semaphore is decremented.
> >
> > The semid for which the calling process is awaiting action is removed from the system. When this occurs, *errno* is set equal to EIDRM, and a value of -1 is returned.
> >
> > The calling process receives a signal that is to be caught. When this occurs, the value of semzcnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in *signal*(2)).

EXAMPLES

The following call to *semop* atomically performs a "P" or "get" operation on the second semaphore in the semaphore set and a "V" or "release" operation on the third semaphore in the set. This example assumes the process has a valid semid which represents a set of 4 semaphores as shown on the *semget*(2) manual page. It also assumes that the semvals of the semaphores in the set have been initialized as shown on the *semctl*(2) manual page.

```
struct sembuf sops[4];

        sops[0].sem_num =  1;
        sops[0].sem_op  = -1; /* P (get) */
        sops[0].sem_flg =  0;
        sops[1].sem_num =  2;
        sops[1].sem_op  =  1; /* V (release) */
        sops[1].sem_flg =  0;

        semop (mysemid, sops, 2);
```

ERRORS

Semop will fail if one or more of the following are true for any of the semaphore operations specified by *sops*:

[EINVAL]	*Semid* is not a valid semaphore identifier.
[EFBIG]	*Sem_num* is less than zero or greater than or equal to the number of semaphores in the set associated with *semid.*
[E2BIG]	*Nsops* is greater than the system-imposed maximum.
[EACCES]	Operation permission is denied to the calling process (see *intro*(2)).
[EAGAIN]	The operation would result in suspension of the calling process but (*sem_flg* & **IPC_NOWAIT**) is "true".

[ENOSPC]	The limit on the number of individual processes requesting an **SEM_UNDO** would be exceeded.
[EINVAL]	The number of individual semaphores for which the calling process requests a **SEM_UNDO** would exceed the limit.
[ERANGE]	An operation would cause a semval to overflow the system-imposed limit.
[ERANGE]	An operation would cause a semadj value to overflow the system-imposed limit.
[EFAULT]	*Sops* points to an illegal address. The reliable detection of this error will be implementation dependent.

Upon successful completion, the value of sempid for each semaphore specified in the array pointed to by *sops* is set equal to the process ID of the calling process. The value of **sem_otime** in the data structure associated with the semaphore identifier will be set to the current time.

RETURN VALUE

If *semop* returns due to the receipt of a signal, a value of −1 is returned to the calling process and *errno* is set to EINTR. If it returns due to the removal of a *semid* from the system, a value of −1 is returned and *errno* is set to EIDRM.

Upon successful completion, a non-negative value is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

WARNING

Check all references to *signal*(2) for appropriateness on systems that support *sigvector*(2). *Sigvector*(2) can affect the behavior described on this page.

SEE ALSO

exec(2), exit(2), fork(2), semctl(2), semget(2), stdipc(3C).

NAME

setgroups – set group access list

SYNOPSIS

#include <sys/param.h>
setgroups(ngroups, gidset)
int ngroups, *gidset;

DESCRIPTION

Setgroups sets the group access list of the current user process according to the array *gidset*. The parameter *ngroups* indicates the number of entries in the array and must be no more than NGROUPS, as defined in <*sys/param.h*>.

Only the superuser may set new groups by adding to the group access list of the current user process; any user may delete groups from it.

RETURN VALUE

A 0 value is returned on success, -1 on error, with an error code stored in *errno*.

ERRORS

The *setgroups* call will fail if:

[EPERM]	The caller is not the superuser and has attempted to set new groups.
[EFAULT]	The address specified for *gidset* is outside the process address space. The reliable detection of this error will be implementation dependent.
[EINVAL]	*Ngroups* is greater than NGROUPS or not positive.
[EINVAL]	An entry in *gidset* is not a valid group ID.

AUTHOR

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

SEE ALSO

getgroups(2), initgroups(3C)

NAME

sethostname – set name of host cpu

SYNOPSIS

sethostname(name, namelen)
char *name;
int namelen;

DESCRIPTION

This call sets the name of the host processor to be *name,* which has a length of *namelen* characters. The maximum value of *namelen* is **UTSLEN** as defined in **<sys/utsname.h>.** This is normally executed when the system is bootstrapped, executed out of the file **/etc/rc.** This system call sets the *nodename* field in the *utsname* structure returned by *uname*(2).

ERRORS

Sethostname will fail and return an error if:

[EPERM]	It is not executed by the super-user.
[EFAULT]	*Name* points to an illegal address. The reliable detection of this error will be implementation dependent.

HARDWARE DEPENDENCIES

Integral PC

Normal users have all super-user capabilities.

AUTHOR

Sethostname was developed by the University of California, Berkeley.

SEE ALSO

hostname(1), uname(1), gethostname(2), uname(2).

NAME

setpgrp, setpgrp2 – set process group ID

SYNOPSIS

int setpgrp ()

int setpgrp2 (pid, pgrp)
int pid, pgrp;

DESCRIPTION

Setpgrp sets the process group ID of the calling process to the process ID of the calling process and returns the new process group ID.

Setpgrp breaks the calling process's terminal affiliation unless it is already the process group leader, see *termio*(7). *Setpgrp* enables the sending of SIGHUP upon process group leader termination, see *exit*(2).

Setpgrp2 sets the process group ID of the process specified by *pid* to be *pgrp*. If *pid* is zero, the process group ID of the current process will be affected.

The following condition must be met for *setpgrp2* to succeed; otherwise, the error [EINVAL] is returned:

> The value of *pgrp* must be in the range of valid process group ID values, or it must be zero ("no process group").

In addition, one or more of the following conditions must be met for *setpgrp2* to succeed; otherwise, the error [EPERM] is returned:

> The effective user ID of the current process is super-user.
>
> The affected process is a descendant of the current process.
>
> The real or effective user ID of the current process matches the real or saved user ID of the affected process.

In addition, one or more of the following conditions must be met for *setpgrp2* to succeed, otherwise, the error [EPERM] is returned:

> The effective user ID of the current process is super-user.
>
> The value of *pgrp* matches the saved process group ID of the current process.
>
> All processes with a process ID or process group ID that is the same as *pgrp* have the same real or saved user ID as the real or effective user ID of the current process, or are descendants of the current process.

Setpgrp2 does not affect the process' terminal affiliation, but does affect whether the process is in the distinguished process group of the terminal, see *termio*(7). *Setpgrp2* disables the sending of SIGHUP upon process group leader termination, see *exit*(2).

ERRORS

Setpgrp2 will fail and no change will occur if any of the following are true:

[ESRCH]	No process can be found corresponding to that specified by *pid*.
[EPERM]	The effective user ID of the current process is not super-user; and the real or effective user ID of the current process does not match the real or saved user ID of the specified process; and the specified processes are not descendants of the current process.
[EPERM]	The effective user ID of the current process is not super-user; and the value *pgrp* does not match the saved process group ID of calling process; and a process exists that is not a descendant of the calling process and whose process ID or process group ID match *pgrp*, while neither the real or saved user ID of this

process match either the real or effective user ID of the calling process.

[EINVAL] The value for *pgrp* is outside the range of valid process group ID values and is non-zero.

RETURN VALUE

Setpgrp returns the value of the new process group ID.

Upon successful completion, *setpgrp2* returns zero. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

HARDWARE DEPENDENCIES

Series 200, 300, 500

Setpgrp2 is not supported.

AUTHOR

Setpgrp and *setpgrp2* were developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

exec(2), exit(2), fork(2), getpid(2), kill(2), signal(2), termio(7).

NAME

setresuid, setresgid – set real, effective, and saved user and group IDs

SYNOPSIS

int setresuid (ruid, euid, suid)
int ruid, euid, suid;

int setresgid (rgid, egid, sgid)
int rgid, egid, sgid;

DESCRIPTION

Setresuid sets the real, effective and/or saved user ID of the calling process.

If the current real, effective or saved user ID is equal to the super-user's user ID, *setresuid* sets the real, effective and saved user IDs to *ruid*, *euid* and *suid*, respectively. Otherwise, *setresuid* will only set the real, effective and saved user IDs if *ruid*, *euid* and *suid* each match at least one of the current real, effective or saved user IDs.

If *ruid*, *euid* or *suid* is **-1**, *setresuid* will leave the current real, effective or saved user ID unchanged.

Setresgid sets the real, effective and/or saved group ID of the calling process.

If the current real, effective or saved user ID is equal to the super-user's user ID, *setresgid* sets the real, effective and saved group IDs to *rgid*, *egid* and *sgid*, respectively. Otherwise, *setresgid* will only set the real, effective and saved group IDs if *rgid*, *egid* and *sgid* each match at least one of the current real, effective or saved group IDs.

If *rgid*, *egid* or *sgid* is **-1**, *setresgid* will leave the current real, effective or saved group ID unchanged.

ERRORS

Setresuid and *setresgid* will fail and return **-1** if:

[EINVAL]	*Ruid*, *euid* or *suid* (*rgid*, *egid* or *sgid*) is not a valid user (group) ID.
[EPERM]	None of the conditions above are met.

RETURN VALUE

Upon successful completion, a value of **0** is returned. Otherwise, a value of **-1** is returned and *errno* is set to indicate the error.

HARDWARE DEPENDENCIES

Integral Personal Computer:

For super-user capabilities described above, it is not necessary to be super-user.

AUTHOR

Setresuid and *setresgid* were developed by HP.

SEE ALSO

exec(2), getuid(2), setuid(2).

NAME

setuid, setgid – set user and group IDs

SYNOPSIS

int setuid (uid)
int uid;

int setgid (gid)
int gid;

DESCRIPTION

Setuid sets the real and/or effective user ID of the calling process. The real, effective, and saved user IDs are called *"ruid"*, *"euid"*, and *"suid"*, respectively. The super user's user ID is zero.

If *uid* and *suid* are both zero then *setuid* sets *ruid* and *euid* to zero.

Otherwise, if *uid* is not zero and is equal to *ruid* then *setuid* sets *euid* to *ruid.*

Otherwise, if *uid* is not zero and is equal to *euid* then *setuid* sets *ruid* to *euid.*

Otherwise, if *uid* is not zero and is equal to *suid* then *setuid* sets *euid* to *suid.*

Otherwise, if *euid* is zero then *setuid* sets *ruid* and *euid* to *uid.*

Setgid sets the real, effective, and/or saved group ID of the calling process (*"rgid"*, *"egid"*, and *"sgid"*, respectively).

If *gid* is equal to *rgid* then *setgid* sets *egid* to *rgid.*

Otherwise, if *gid* is equal to *egid* then *setgid* sets *rgid* to *egid.*

Otherwise, if *gid* is equal to *sgid* then *setgid* sets *egid* to *sgid.*

Otherwise, if *euid* is zero then *setgid* sets *rgid*, *egid*, and *sgid* to *gid.*

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

Setuid and *setgid* will fail and return −1 if:

[EPERM]	None of the conditions above are met.
[EINVAL]	*Uid* (*gid*) is not a valid user (group) ID.

HARDWARE DEPENDENCIES

Integral PC

Saved user IDs or saved group IDs are currently not supported.

Normal users have all super-user capabilities.

SEE ALSO

exec(2), getuid(2).

NAME

shmctl – shared memory control operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (shmid, cmd, buf)
int shmid, cmd;
struct shmid_ds *buf;
```

DESCRIPTION

Shmctl provides a variety of shared memory control operations as specified by *cmd*. The following *cmd*s are available:

IPC_STAT Place the current value of each member of the data structure associated with *shmid* into the structure pointed to by *buf*. The contents of this structure are defined in the *glossary*. {READ}

IPC_SET Set the value of the following members of the data structure associated with *shmid* to the corresponding value found in the structure pointed to by *buf*:
shm_perm.uid
shm_perm.gid
shm_perm.mode /* only low 9 bits */

This *cmd* can only be executed by a process that has an effective user ID equal to either that of super-user or to the value of either **shm_perm.uid** or **shm_perm.cuid** in the data structure associated with *shmid*.

IPC_RMID
Remove the shared memory identifier specified by *shmid* from the system and destroy the shared memory segment and data structure associated with it. If the segment is attached to one or more processes, then the segment key is changed to IPC_PRIVATE and the segment is marked removed. The segment will disappear when the last attached process detaches it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of super-user or to the value of either **shm_perm.uid** or **shm_perm.cuid** in the data structure associated with *shmid*.

SHM_LOCK
Lock the shared memory segment specified by *shmid* in memory. This *cmd* can only be executed by a process that either has an effective user ID equal to super-user or has an effective user ID equal to the value of either **shm_perm.uid** or **shm_perm.cuid** in the data structure associated with *shmid* and has PRIV_MLOCK privilege (see *setprivgrp* on *getprivgrp*(2)).

SHM_UNLOCK
Unlock the shared memory segment specified by *shmid*. This *cmd* can only be executed by a process that either has an effective user ID equal to super-user or has an effective user ID equal to the value of either **shm_perm.uid** or **shm_perm.cuid** in the data structure associated with *shmid* and has PRIV_MLOCK privilege (see *setprivgrp* on *getprivgrp*(2)).

EXAMPLES

The following call to *shmctl* locks in memory the shared memory segment represented by *myshmid*. This example assumes the process has a valid shmid, which can be obtained by calling *shmget*(2).

```
shmctl (myshmid, SHM_LOCK, 0);
```

The following call to *shmctl* removes the shared memory segment represented by *myshmid*. This example assumes the process has a valid shmid, which can be obtained by calling *shmget*(2).

```
shmctl (myshmid, IPC_RMID, 0);
```

ERRORS

Shmctl will fail if one or more of the following are true:

[EINVAL]	*Shmid* is not a valid shared memory identifier.
[EINVAL]	*Cmd* is not a valid command.
[EACCES]	*Cmd* is equal to **IPC_STAT** and {READ} operation permission is denied to the calling process (see *glossary*).
[EPERM]	*Cmd* is equal to **IPC_RMID, IPC_SET, SHM_LOCK,** or **SHM_UNLOCK** and the effective user ID of the calling process is not equal to that of super-user and it is not equal to the value of either **shm_perm.uid** or **shm_perm.cuid** in the data structure associated with *shmid*.
[EPERM]	*Cmd* is equal to **SHM_LOCK** or **SHM_UNLOCK** and the effective user ID of the calling process is not equal to that of super-user and the calling process does not have PRIV_MLOCK privilege (see *setprivgrp* on *getprivgrp*(2)).
[EINVAL]	*Cmd* is equal to **SHM_UNLOCK** and the shared-memory segment specified by *shmid* is not locked in memory.
[EFAULT]	*Buf* points to an illegal address. The reliable detection of this error will be implementation dependent.
[ENOMEM]	*Cmd* is equal to SHM_LOCK and there is not sufficient lockable memory to fill the request.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

HARDWARE DEPENDENCIES

Series 200, 300

[EACCESS] *Shmid* is the id of a shared memory segment currently being used by the system to implement other features (see *graphics*(7) and *iomap*(7)).

AUTHOR

Shmctl was developed by AT&T and HP.

SEE ALSO

shmget(2), shmop(2), stdipc(3C).

NAME

semget – get set of semaphores

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (key, nsems, semflg)
key_t key;
int nsems, semflg;
```

DESCRIPTION

Semget returns the semaphore identifier associated with *key*.

A semaphore identifier and associated data structure and set containing *nsems* semaphores are created for *key* if one of the following are true:

Key is equal to **IPC_PRIVATE**.

Key does not already have a semaphore identifier associated with it, and (*semflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new semaphore identifier is initialized as follows:

Sem_perm.cuid, **sem_perm.uid**, **sem_perm.cgid**, and **sem_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of **sem_perm.mode** are set equal to the low-order 9 bits of *semflg*.

Sem_nsems is set equal to the value of *nsems*.

Sem_otime is set equal to 0 and **sem_ctime** is set equal to the current time.

EXAMPLES

The following call to *semget* returns a semid associated with the key returned by ftok("myfile", 'A'). If a semid associated with the key does not exist, a new semid, set of 4 semaphores and associated data structure will be created. If a semid for the key already exists, the semid is simply returned.

```
        int semid;

                mysemid = semget (ftok("myfile",'A'), 4, IPC_CREAT | 0600);
```

ERRORS

Semget will fail if one or more of the following are true:

[EINVAL]	*Nsems* is either less than or equal to zero or greater than the system-imposed limit.
[EACCES]	A semaphore identifier exists for *key*, but operation permission as specified by the low-order 9 bits of *semflg* would not be granted.
[EINVAL]	A semaphore identifier exists for *key*, but the number of semaphores in the set associated with it is less than *nsems* and *nsems* is not equal to zero.
[ENOENT]	A semaphore identifier does not exist for *key* and (*semflg* & **IPC_CREAT**) is "false".
[ENOSPC]	A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphore identifiers system wide would be exceeded.

[ENOSPC]	A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores system wide would be exceeded.
[EEXIST]	A semaphore identifier exists for *key* but ((*semflg* & IPC_CREAT) **&** (*semflg* & **IPC_EXCL**)) is "true".

RETURN VALUE

Upon successful completion, a non-negative integer, namely a semaphore identifier, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

semctl(2), semop(2), stdipc(3C).

NAME

shmop – shared memory operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat (shmid, shmaddr, shmflg)
int shmid;
char *shmaddr
int shmflg;

int shmdt (shmaddr)
char *shmaddr
```

DESCRIPTION

Shmat attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the data segment of the calling process. If the shared memory segment has not already been attached *shmaddr* must be specified as zero, and the segment will be attached at a location selected by the operating system. That location will be the same in all processes accessing that shared memory object. If the shared memory segment has already been attached a non-zero value of *shmaddr* will be accepted as long as the specified address is the same as the current attach address of the segment. Some implementations may permit the specification of a non-zero value as a machine dependent extension, as discussed in HARDWARE DEPENDENCIES below. Systems which do this do not necessarily guarantee that a given shared memory object will appear at the same address in all processes which access it, unless the user specifies an address.

The segment is attached for reading if (*shmflg* & **SHM_RDONLY**) is "true" {READ}, otherwise it is attached for reading and writing {READ/WRITE}. It is not possible to attach a segment for write only.

EXAMPLES

The following call to shmat() attaches the shared memory segment to the process. This example assumes the process has a valid shmid, which can be obtained by calling *shmget*(2).

```
char *shmptr;

          shmptr = (char *) shmat(myshmid, 0, 0);
```

The following call to shmdt() then detaches the shared memory segment.

```
shmdt (shmptr);
```

ERRORS

Shmat will fail and not attach the shared memory segment if one or more of the following are true:

[EINVAL]	*Shmid* is not a valid shared memory identifier.
[EACCES]	Operation permission is denied to the calling process.
[ENOMEM]	The available data space is not large enough to accommodate the shared memory segment.
[EINVAL]	*Shmaddr* is not zero and the machine does not permit non-zero values or *shmaddr* is not equal to the current attach location for the shared memory segment.

[EMFILE] The number of shared memory segments attached to the calling process would exceed the system-imposed limit.

Shmdt detaches from the calling process's data segment the shared memory segment located at the address specified by *shmaddr*.

Shmdt will fail and return -1 if the following is true.

[EINVAL] *Shmdt* will fail and not detach the shared memory segment if *shmaddr* is not the data segment start address of a shared memory segment.

HARDWARE DEPENDENCIES

Series 500

Shmaddr must be zero in all cases for *shmat*. Otherwise, an error is generated. In addition, **SHM_RDONLY** is not supported, and if it is set in *shmflg,* an error is generated.

[EINVAL] *Shmflg* has **SHM_RDONLY** set.

Series 200, 300

Shmaddr may be non-zero, and if it is, the segment is attached at the address specified by one of the following criteria:

If *shmaddr* is equal to zero, the segment is attached at the first available address as selected by the system. The selected value will vary for each process accessing that shared memory object.

If *shmaddr* is not equal to zero and (*shmflg* & **SHM_RND)** is "true", the segment is attached at the address given by (*shmaddr* - (*shmaddr* modulus **SHMLBA**)).

If *shmaddr* is not equal to zero and (*shmflg* & **SHM_RND)** is "false", the segment is attached at the address given by *shmaddr*.

This form of *shmat* will fail and not attach the shared memory segment if one or more of the following are true:

[EACCES] *Shmid* is the id of a shared memory segment currently being used by the system to implement other features (see *graphics*(7) and *iomap*(7)).

[EINVAL] *Shmaddr* is not equal to zero, and the value of (*shmaddr* - (*shmaddr* modulus **SHMLBA**)) is an illegal address.

[EINVAL] *Shmaddr* is not equal to zero, (*shmflg* & **SHM_RND)** is "false", and the value of *shmaddr* is an illegal address.

[ENOMEM] The calling process is locked (see *plock*(2)) and there is not sufficient lockable memory to support the process-related data structure overhead.

Series 800

Shmat will fail and return **−1** if the following is true:

[EINVAL] The calling process is already attached to shmid.

RETURN VALUES

Upon successful completion, the return value is as follows:

Shmat returns the data segment start address of the attached shared memory segment.

Shmdt returns a value of 0.

Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), exit(2), fork(2), shmctl(2), shmget(2), stdipc(3C).

NAME

sigblock – block signals

SYNOPSIS

long sigblock(mask);
long mask;

DESCRIPTION

Sigblock causes the signals specified in *mask* to be added to the set of signals currently being blocked from delivery. Signal *i* is blocked if the *i*-th bit in *mask* is a 1 (that is, if (mask & (1L << (i-1))) != 0).

It is not possible to block those signals which cannot be ignored, as documented in *signal*(2); this restriction is silently imposed by the system.

Sigsetmask(2) can be used to set the mask absolutely.

EXAMPLES

The following call to *sigblock* adds the SIGUSR1 and SIGUSR2 signals to the mask of signals currently blocked for the process:

```
#define MASK(s)        (1L << ((s)-1))

long oldmask;

          oldmask = sigblock (MASK (SIGUSR1) | MASK (SIGUSR2));
```

RETURN VALUE

The previous set of masked signals is returned.

AUTHOR

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

SEE ALSO

kill(2), sigsetmask(2), sigvector(2).

NAME

signal – specify what to do upon receipt of a signal

SYNOPSIS

```
#include <signal.h>

int (*signal (sig, func))()
int sig;
int (*func)();

func(sig [, code, scp ] )
int sig, code;
struct sigcontext *scp;
```

DESCRIPTION

Signal allows the calling process to choose one of three ways in which it is possible to handle the receipt of a specific signal. *Sig* specifies the signal and *func* specifies the choice.

Sig can be assigned any one of the following except SIGKILL or SIGSTOP:

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03*	quit
SIGILL	04*●	illegal instruction
SIGTRAP	05*●	trace trap
SIGIOT	06*	software generated (sent by *abort*(3C))
SIGEMT	07*	software generated
SIGFPE	08*	floating point exception
SIGKILL	09‡+%	kill
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user defined signal 1
SIGUSR2	17	user defined signal 2
SIGCLD	18†	death of a child (see **WARNING** below)
SIGPWR	19●†	power fail (see **WARNING** below)
SIGVTALRM	20	virtual timer alarm; see *getitimer*
SIGPROF	21	profiling timer alarm; see *getitimer*
SIGIO	22†	Asynchronous IO signal; see *select*
SIGWINDOW	23†	A window change or mouse signal; see the windowing package
SIGSTOP	24‡+#	stop
SIGTSTP	25#	stop signal generated from keyboard
SIGCONT	26†%	continue after stop
SIGTTIN	27#	background read attempted from control terminal
SIGTTOU	28#	background write attempted to control terminal

*	Indicates that a core dump can be generated.
†	Indicates that the action on SIG_DFL is to ignore the signal, rather than terminate the process.
#	Indicates that the action on SIG_DFL is to stop rather than terminate the process.

%	Indicates that the signal will not be held off by a stopped process.
•	Indicates that the signal is not reset when it is caught by *signal*.
‡	Indicates that the signal cannot be ignored.
+	Indicates that the signal cannot be caught.

See below for details.

Func is assigned one of three values: SIG_DFL, SIG_IGN, or a *function address*. The actions prescribed by these values are as follows:

SIG_DFL – (usually) terminate process upon receipt of a signal.
For those signals not flagged with a dagger (†) or a pound (#) above, upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(2). In addition a "core image" will be made in the current working directory of the receiving process if *sig* is one for which an asterisk appears in the above list *and* the following conditions are met:

The effective user ID and the real group ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of 0666 modified by the file creation mask, see *umask*(2)

a file owner ID that is the same as the effective user ID of the receiving process

a file group ID that is the same as the effective group ID of the receiving process

SIG_DFL – (#) stop process upon receipt of a signal.
For those signals flagged with a pound (#) above, upon receipt of the signal *sig*, the receiving process is to be stopped. While a process is stopped, any additional signals (except those marked with a percent (%) above that are processed immediately) that are sent to the process will be held off until the process is restarted. When the process is restarted, pending signals will be processed. When a process whose parent is the initialization process (proc1) stops as the result of receiving the SIGTSTP, SIGTTIN, or SIGTTOU signals, it is sent the SIGKILL signal, which causes the process to terminate.

SIG_DFL – (†) no action upon receipt of a signal.
For those signals flagged with a dagger (†) above, neither terminate nor stop action is taken.

SIG_IGN – ignore signal.
The signal *sig* is to be ignored.

Note: the signals SIGKILL and SIGSTOP cannot be ignored.

function address – catch signal.
Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function pointed to by *func*. The signal number *sig* will be passed as the first parameter to the signal-catching function. The HP-UX kernel will also pass two additional (optional) parameters to signal handler routines. The complete parameter list for *func* is:

sig	signal number.
code	a word of information usually provided by the hardware.

scp	a pointer to the machine dependent structure *sigcontext* defined in the include file **signal.h**.

Depending on the value of *sig*, *code* can be zero and/or *scp* can be NULL. The meanings of *code* and *scp* and the conditions upon which they are other than zero or NULL are implementation dependent. It is permissible for *code* to always be zero, and *scp* to always be NULL.

The pointer *scp* will only be valid during the context of the signal handler.

The optional parameters can be omitted from the handler parameter list, in which case the handler is exactly compatible with System V UNIX.

Truly portable software should not use the optional parameters in signal-catching routines.

Before entering the signal-catching function, the value of *func* for the caught signal will be set to SIG_DFL unless the signal is one of those flagged with a bullet (●) above.

Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted.

When a signal that is to be caught occurs during the execution of calls such as *read*, a *write*, an *open*, or an *ioctl* system call on a slow device (like a terminal; but not a file), during a *pause* system call, or during a *wait* system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function will be executed and then the interrupted system call can return a −1 to the calling process with *errno* set to EINTR.

Note: The signals SIGKILL and SIGSTOP cannot be caught.

SIGKILL may be sent by the system in the event of an unsuccessful *exec*, if the original program has already been deleted. When *signal* is called with *func* equal to SIG_IGN and a signal *sig* is pending, the pending signal is cleared.

EXAMPLES

The following call to *signal* sets up a signal handler for the SIGINT signal:

```
    int myhandler();

              signal (SIGINT, myhandler);
```

RETURN VALUE

Upon successful completion, *signal* returns the previous value of *func* for the specified signal *sig*. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

Signal will fail if one or more of the following are true:

[EINVAL]	*Sig* is an illegal signal number, or is equal to SIGKILL or SIGSTOP.

WARNINGS

Two other signals that behave differently than the signals described above exist in this release of the system; they are:

SIGCLD	18†	death of a child
SIGPWR	19●†	power fail

There is no guarantee that, in future releases of the HP-UX system, these signals will continue to behave as described below; they are included only for compatibility with other versions of the UNIX system. Their use in new programs is strongly discouraged.

For these signals, *func* is assigned one of three values: SIG_DFL, SIG_IGN, or a *function address*. The actions prescribed by these values are as follows:

SIG_DFL - - ignore signal.
: The signal is to be ignored.

SIG_IGN - - ignore signal.
: The signal is to be ignored. Also, if *sig* is SIGCLD, the calling process's child processes will not create zombie processes when they terminate, see *exit*(2).

function address - - catch signal.
: If the signal is SIGPWR, the action to be taken is the same as that described above for *func* equal to *function address*. The same is true if the signal is SIGCLD. In addition, if *signal* is called to catch SIGCLD in a process that currently has terminated (zombie) children, a SIGCLD signal is delivered to the process immediately. Thus if the signal-catching function re-installs itself, the apparent effect is that any SIGCLD signals received due to the death of children while the function is executing are queued and the signal-catching function is continually reentered until the queue is empty. Note that the function must re-install itself after it has called *wait*(2) or *wait3*(2). Otherwise the presence of the child that caused the original signal will cause another signal immediately, resulting in infinite recursion.

The SIGCLD affects two other system calls (*wait*(2), and *exit*(2)) in the following ways:

wait
: If the *func* value of SIGCLD is set to SIG_IGN and a *wait* is executed, the *wait* will block until all of the calling process's child processes terminate; it will then return a value of **−1** with *errno* set to ECHILD.

exit
: If in the exiting process's parent process the *func* value of SIGCLD is set to SIG_IGN, the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the preceding processes. A process that can be piped into in this manner (and thus become the parent of other processes) should take care not to set SIGCLD to be caught.

Some implementations do not generate SIGPWR. For systems without non-volatile memory, it is not useful. If SIGPWR is generated, it occurs when power is restored and the system has done all necessary re-initialization. Processes will re-start by responding to SIGPWR.

HARDWARE DEPENDENCIES

Series 200, 300, 500
: The SIGSTOP, SIGTSTP, SIGCONT, SIGTTIN, SIGTTOU signals are not supported.

Series 200, Series 300
: The signal SIGPWR is not currently generated.

The *code* word is always zero for all signals except signal 4 (SIGILL) and signal 8 (SIGFPE). For SIGILL, *code* has the following values:

0	illegal instruction;
6	check instruction;
7	TRAPV;
8	privilege violation.

For SIGFPE, *code* has the following values:

0	floating point exception;
5	divide-by-zero.

Refer to the MC68000 processor documentation provided with your system for more detailed information about the meaning of these errors.

Series 500

The SIGEMT signal means "out of memory," and is generated by the HP-UX Operating System. When sent by the system, this signal is always fatal to the process, and cannot be caught or ignored.

SIGIOT can be sent if an invalid string operation is attempted, or if a bounds range check trap is encountered.

The signal SIGBUS is not currently generated by the operating system.

The signal handler parameter *code* contains the trap number provided by the hardware in the event a trap occurs in the user's program; see *trapno*(2) for a list of these trap numbers. Otherwise, *code* will be zero.

The structure pointer *scp* is defined when a trap occurs in the user's program, and points to the structure *sigcontext* defined in **signal.h**. If no trap occurs, *scp* will be NULL.

A zero value is returned on floating point underflow. Floating point overflow, divide-by-zero, integer divide-by-zero, and illegal floating point operation exceptions result in the signal SIGFPE being sent to the process. An undefined value is returned as the result of the operation if the signal SIGFPE is ignored or caught.

SIGFPE is not sent on integer overflow. Instead, a wrapped integer result is returned.

Series 800

The structure pointer *scp* is always defined.

The *code* word is always zero for all signals except signal 4 (SIGILL) and signal 8 (SIGFPE). For SIGILL, *code* has the following values:

8	illegal instruction trap;
9	break instruction trap;
10	privileged operation trap;
11	privileged register trap.

For SIGFPE, *code* has the following values:

12	overflow trap;
13	conditional trap;
14	assist exception trap;
22	assist emulation trap.

Refer to the Series 800 processor documentation provided with your system for more detailed information about the meaning of these errors.

Integral PC

The Integral PC implements the signal SIGMOUSE with a value of 20.

AUTHOR

Signal was developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

kill(1), kill(2), lseek(2), pause(2), wait(2), abort(3C), setjmp(3C).

NAME

sigpause – atomically release blocked signals and wait for interrupt

SYNOPSIS

long sigpause(sigmask)
long sigmask;

DESCRIPTION

Sigpause blocks signals according to the value of *sigmask* in the same manner as *sigsetmask*(2), then atomically waits for an unmasked signal to arrive. On return *sigpause* restores the current signal mask to the value that existed before the *sigpause* call. When no signals are to be blocked, a value of 0L is used for *sigmask.*

In normal usage, a signal is blocked using *sigblock*(2). To begin a critical section variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses, awaiting work by using *sigpause* with the mask returned by *sigblock.*

EXAMPLES

The following call to *sigpause* waits until the calling process receives a signal:

```
sigpause (0L);
```

The following example blocks the SIGIO signal until the *sigpause* is called. When a signal is received at the *sigpause* statement, the signal mask is restored to its value before *sigpause* was called:

```
#define MASK(s)        (1L << ((s)-1))

long savemask;

          savemask = sigblock (MASK (SIGIO));

          /* critical section */

          sigpause (savemask);
```

RETURN VALUE

Sigpause will terminate when it is interrupted by a signal. When *sigpause* terminates, it will return -1 and set *errno* to **EINTR.**

WARNINGS

Check all references to *signal*(2) for appropriateness on systems that support *sigvector*(2). *Sigvector*(2) can affect the behavior described on this page.

AUTHOR

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

SEE ALSO

sigblock(2), sigsetmask(2), sigvector(2).

NAME

sigsetmask – set current signal mask

SYNOPSIS

long sigsetmask(mask);
long mask;

DESCRIPTION

Sigsetmask sets the current signal mask (those signals which are blocked from delivery). Signal *i* is blocked if the *i*-th bit in *mask* is a 1 (that is, if (mask & (1L << (i-1))) != 0).

It is not possible to mask those signals which cannot be ignored, as documented in *signal*(2); this restriction is silently imposed by the system.

Sigblock(2) can be used to add elements to the set of blocked signals.

EXAMPLES

The following call to *sigsetmask* causes only the SIGUSR1 and SIGUSR2 signals to be blocked:

```
#define MASK(s)        (1 << ((s)-1))

long oldmask;

        oldmask = sigsetmask (MASK (SIGUSR1) | MASK (SIGUSR2));
```

RETURN VALUE

The previous set of masked signals is returned.

AUTHOR

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

SEE ALSO

kill(2), sigblock(2), sigpause(2), sigvector(2).

NAME

sigspace – assure sufficient signal stack space

SYNOPSIS

```
#include <signal.h>

long sigspace(ss);
long ss;
```

DESCRIPTION

Sigspace allows users to define additional space for stack use which is guaranteed to be available if signals are to be processed. If *ss* is positive, it specifies a space, in bytes, which the system guarantees will be available when processing a signal. A zero value removes any guarantee of space and any negative value leaves the guarantee unchanged, and may be used to interrogate the current guaranteed value. When a signal's action indicates its handler should use the guaranteed space (specified with a *sigvector*(2) call), the system checks to see if the process is currently using that space. If the process is not currently using that space the system arranges for that space to be available for the duration of the signal handler's execution. If that space has already been made available (due to a previous signal) no change is made. The normal stack discipline is resumed when the signal causing the use of the guaranteed space is exited.

The guaranteed space is inherited by child processes after a *fork* but the guarantee of space is removed after an *exec*.

NOTES

The guaranteed space may not be increased in size automatically, as is done for the normal stack. If the stack overflows the guaranteed space unpredictable results may occur.

Guaranteeing space for a stack may cause interference with other memory allocation routines, in an implementation dependent manner.

During normal execution of the program the system checks for possible overflow of the stack. Guaranteeing space may cause the space available for normal execution to be reduced.

Leaving the context of a service routine in an abnormal way, such as by *longjmp* on *setjmp*(3C), may remove the guarantee that the ordinary execution of the program will not extend into the guaranteed space. It may also cause the program to forever lose its ability to automatically increase the stack size, and the program may then be limited to the guaranteed space.

RETURN VALUE

Upon successful completion, a the size of the old guaranteed space is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

Sigspace will fail and the guaranteed amount of space will remain unchanged if one of the following occurs.

[ENOMEM]	Enough space cannot be guaranteed because of either hardware limitations or because some software imposed limit would be exceeded.

HARDWARE DEPENDENCIES

Series 500:

Sigspace is ignored (as a no-op) by Series 500. The return value is always zero.

Series 200, 300:

The guaranteed space is allocated with *malloc*(3C). This call may thus interfere with other heap management mechanisms.

The kernel overhead taken in the reserved space is 148 bytes on Series 200 computers and 440 bytes on Series 300. This overhead must be included in the requested amount. These values are subject to change in future releases.

BUGS

Methods for calculating the required size are not yet well developed.

AUTHOR

Sigspace was developed by the Hewlett-Packard Company.

SEE ALSO

sigvector(2), setjmp(3C).

NAME

sigvector – software signal facilities

SYNOPSIS

#include <signal.h>

sigvector(sig, vec, ovec)
int sig;
struct sigvec *vec, *ovec;

DESCRIPTION

The system defines a set of signals that can be delivered to a process. The set of signals is defined in *signal*(2), along with the meaning and side-effects of each signal. This manual page, along with those for *sigblock*(2), *sigsetmask*(2), *sigpause*(2), and *sigspace*(2) define an alternate mechanism for handling these signals that assures the delivery of signals and integrity of signal handling procedures. The facilities described here should not be used in the same program as *signal*(2).

With this interface, signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process can specify a handler to which a signal is delivered, or specify that a signal is to be blocked or ignored. A process can also specify that a default action is to be taken by the system when a signal occurs. It is possible to assure a minimum amount of stack space for processing signals using the *sigspace*(2) call.

All signals have the same priority. Signal routines execute with the signal that caused their invocation blocked, but other signals can yet occur. A global signal mask defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally 0). It can be changed with a *sigblock*(2), *sigsetmask*(2), or *sigpause*(2) call, or when a signal is delivered to the process.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently blocked by the process, it is delivered to the process. When a signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described below), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally, the process will resume execution in the context from before the signal's delivery. If the process wishes to resume in a different context, it must arrange to restore the previous context itself.

When a signal is delivered to a process, a new signal mask is installed for the duration of the process' signal handler (or until a *sigblock* or *sigsetmask* call is made). This mask is formed by taking the current signal mask, adding the signal to be delivered, and *or*'ing in the signal mask associated with the handler to be invoked. When the user's signal handler returns normally, the original mask is restored.

Sigvector assigns a handler for a specific signal. *Vec* and *ovec* are pointers to *sigvec* structures that include the following elements:

```
int     (*sv_handler)();
long    sv_mask;
long    sv_flags;
```

If *vec* is non-zero, it specifies a handler routine, a mask to be used when delivering the specified signal, and a set of flags that can modify the delivery of the signal. If *ovec* is non-zero, the previous handling information for the signal is returned to the user. If *vec* is zero, signal handling is unchanged: thus, the call can be used to enquire about the current handling of a given signal. If *vec* and *ovec* point to the same structure, the value of *vec* is read prior to being overwritten.

The *sv_flags* field can be used to modify the receipt of signals. The following flag bits are defined:

SV_ONSTACK	Use the *sigspace* allocated space
SV_BSDSIG	Use the Berkeley signal semantics

If SV_ONSTACK is set, then the system will use, or permit the use of, the space reserved for signal processing in the *sigspace*(2) system call.

If SV_BSDSIG is set, then the signal will be given the Berkeley semantics. The following signals are affected by this flag.

SIGCLD In addition to being sent when a child process dies, the signal will also be sent when any child's status changes from running to stopped. This would normally be used by a program such as *csh* when maintaining process groups under Berkeley Job Control.

Once a signal handler is installed, it remains installed until another *sigvector* call is made, or an *exec*(2) is performed. The default action for a signal can be reinstated by setting *sv_handler* to SIG_DFL; this default is usually termination. If *sv_handler* is SIG_IGN the signal is usually subsequently ignored, and pending instances of the signal are discarded. The exact meaning of SIG_DFL and SIG_IGN for each signal is discussed in *signal*(2). Unlike *signal*(2), there is no category of "reset when caught" signals.

Certain system calls can be interrupted by a signal, the remainder will complete before the signal is serviced. The *scp* pointer described in *signal*(2) is always non-null if *sigvector* is supported. *Scp* points to a machine-dependent *sigcontext* structure. All implementations of this structure include the fields:

```
int     sc_syscall;
char    sc_syscall_action;
```

The value SYS_NOTSYSCALL for the *sc_syscall* field indicates that the signal is not interrupting a system call; any other value indicates which system call it is interrupting. If a signal that is being caught occurs during one of the interruptable calls, the signal handler is immediately invoked. If the signal handler is exited in a normal way, the value of the *sc_syscall_action* field is inspected; if it is SIG_RETURN the system call is aborted and the interrupted program continues past the call with the result of the interrupted call being -1 and *errno* set to EINTR. If the value of the *sc_syscall_action* field is SIG_RESTART, the call is restarted. A call is restarted if, in the case of a read or write, it had transferred no data. If some data had been transferred, the operation is considered to have completed with a partial transfer, and the *sc_syscall* value is SYS_NOTSYSCALL. Other values are undefined and reserved for future use.

Exiting the handler abnormally (such as with *longjmp* on *setjmp*(3C)) will abort the call, and the user is responsible for the context of further execution. The value of *scp->sc_syscall_action* is ignored when the value of *scp->sc_syscall* is SYS_NOTSYSCALL. *Scp->sc_syscall_action* is always initialized to SIG_RETURN before invocation of a signal handler. When an interruptable call is interrupted by multiple signals, if any signal handler returns a value of SIG_RETURN in *scp->sc_syscall_action,* all subsequent signal handlers are passed a value of SYS_NOTSYSCALL in *scp->sc_syscall.*

The interruptable system calls, and corresponding values for *scp->sc_syscall* are listed below.

Call	sc_syscall value
------	----------------
read (slow devices)	SYS_READ
readv (slow devices)	SYS_READV
write (slow devices)	SYS_WRITE
writev (slow devices)	SYS_WRITEV
open (slow devices)	SYS_OPEN
ioctl (slow requests)	SYS_IOCTL
wait	SYS_WAIT
select	SYS_SELECT
pause	SYS_PAUSE
sigpause	SYS_SIGPAUSE
semop	SYS_SEMOP
msgsnd	SYS_MSGSND
msgrcv	SYS_MSGRCV

Note that *read*, *write* or *ioctl* on fast devices (disks) is not interruptable, but IO to a slow device (teletype) is. Additional system calls, for example those used for networking, can also be interruptable on some implementations. In these cases additional values can be specified for *scp->sc_syscall.* Programs that look at the values of *scp->sc_syscall* should always compare them to these symbolic constants; the numerical values represented by these constants may vary among implementations.

After a *fork*(2) or *vfork*(2) the child inherits all signals, the signal mask, and the reserved signal stack space.

Exec(2) resets all caught signals to the default action, ignored signals remain ignored, the signal mask remains the same, and the reserved signal stack space is released.

NOTES

The mask specified in *vec* is not allowed to block those signals that cannot be ignored, as defined in *signal*(2). This is enforced silently by the system.

If *sigvector* is called to catch SIGCLD in a process that currently has terminated (zombie) children, a SIGCLD signal is delivered to the calling process immediately, or as soon as SIGCLD is unblocked if it is currently blocked. Thus, in a process that spawns multiple children and catches SIGCLD, it is sometimes advisable to re-install the handler for SIGCLD after each invocation in case there are multiple zombies present. This is true even though the handling of the signal is not reset by the system as with *signal*(2), because deaths of multiple processes while SIGCLD is blocked in the handler will only result in delivery of a single signal. Note that the function must re-install itself after it has called *wait*(2) or *wait3*(2). Otherwise the presence of the child that caused the original signal will always cause another signal to be delivered.

RETURN VALUE

A **0** value indicated that the call succeeded. A **−1** return value indicates an error occurred and *errno* is set to indicate the reason.

ERRORS

Sigvector will fail and no new signal handler will be installed if one of the following occurs:

[EFAULT] Either *vec* or *ovec* points to memory that is not a valid part of the process address space. The reliable detection of this error will be implementation dependent.

[EINVAL] *Sig* is not a valid signal number.

[EINVAL] An attempt is made to ignore or supply a handler for a signal that cannot be caught or ignored, see *signal*(2).

WARNINGS

Restarting a *select*(2) call can sometimes cause unexpected results. If the *select* call has a timeout specified, the timeout is restarted with the call, ignoring any portion that had elapsed prior to interruption by the signal. Normally this simply extends the timeout and is not a problem. However, if a handler repeatedly catches signals and the timeout specified to select is longer than the time between those signals, restarting the *select* call effectively renders the timeout infinite.

HARDWARE DEPENDENCIES

Integral PC

Interruptible and restartable kernel calls are not supported. SELECT, READY, and WRITEV are not supported.

Series 200, 300, 500

The SV_BSDSIG flag is not supported.

AUTHOR

Sigvector was developed by HP, and the University of California, Berkeley.

SEE ALSO

kill(1), kill(2), ptrace(2), sigblock(2), signal(2), sigpause(2), sigsetmask(2), sigspace(2), setjmp(3C), termio(7).

NAME

stat, fstat – get file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int stat (path, buf)
char *path;
struct stat *buf;

int fstat (fildes, buf)
int fildes;
struct stat *buf;
```

DESCRIPTION

Path points to a path name naming a file. Read, write, or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable. *Stat* obtains information about the named file.

Similarly, *fstat* obtains information about an open file known by the file descriptor *fildes*, obtained from a successful *open*, *creat*, *dup*, *fcntl*, or *pipe* system call.

Buf is a pointer to a *stat* structure into which information is placed concerning the file.

The contents of the structure pointed to by *buf* include the following members:

```
dev_t   st_dev;     /* ID of device containing a */
                    /* directory entry for this file */
ino_t   st_ino;     /* Inode number */
ushort  st_mode;    /* File mode; see mknod(2) */
short   st_nlink;   /* Number of links */
ushort  st_uid;     /* User ID of file owner */
ushort  st_gid;     /* Group ID of file group */
dev_t   st_rdev;    /* Device ID; this entry defined */
                    /* only for char or blk spec files */
off_t   st_size;    /* File size (bytes) */
time_t  st_atime;   /* Time of last access */
time_t  st_mtime;   /* Last modification time */
time_t  st_ctime;   /* Last file status change time */
                    /* Measured in secs since */
                    /* 00:00:00 GMT, Jan 1, 1970 */
uint    st_remote:1;/* Set if file is remote */
dev_t   st_netdev;  /* ID of device containing */
                    /* network special file */
ino_t   st_netino;  /* Inode number of network special file */
```

st_atime Time when file data was last accessed. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *read*(2), *readv* (on *read*(2)), and *utime*(2).

st_mtime

Time when data was last modified. Changed by the following system calls: *creat*(2), *truncate*(2), *ftruncate* (on *truncate*(2)), *mknod*(2), *pipe*(2), *prealloc*(2), *utime*(2), *write*(2), and *writev* (on *write*(2)). Changed also by *close*(2) when the file is a named pipe (FIFO special) and the reference counts are zero.

st_ctime Time when file status was last changed. Changed by the following system calls: *chmod*(2), *chown*(2), *creat*(2), *fchmod*(2), *fchown*(2), *truncate*(2), *ftruncate* (on *truncate*(2)), *link*(2), *mknod*(2), *pipe*(2), *prealloc*(2), *unlink*(2), *utime*(2), *write*(2), and *writev* (on *write*(2)).

st_remote
A zero value indicates that the file is on the local node; non-zero indicates that the file is on a remote node, and accessed through remote file access (RFA). Not all HP-UX systems support RFA; *st_remote* is always zero on those systems which do not.

st_netdev, st_netino
All remote file access takes place through a special file in the local file system known as a network special file. Each network special file identifies a particular remote node. When *st_remote* is non-zero, *st_netdev* and *st_netino* identify the appropriate network special file; otherwise these fields are zero.

The *touch*(1) command can be used to explicitly control the times of a file.

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

Stat will fail if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist (for example, *path* is null or a component of *path* does not exist).
[EACCES]	Search permission is denied for a component of the path prefix.
[EFAULT]	*Buf* or *path* points to an invalid address. The reliable detection of this error will be implementation dependent.

Fstat will fail if one or more of the following are true:

[EBADF]	*Fildes* is not a valid open file descriptor.
[EFAULT]	*Buf* points to an invalid address. The reliable detection of this error will be implementation dependent.

HARDWARE DEPENDENCIES

Series 500

Besides the definition given above, **st_size** also has meaning in the case of special files which refer to disks. In such a case, **st_size** either returns the total physical size (in bytes) of the mass storage volume, when appropriate, or -1 otherwise. This is a property of the physical device, not any directory structure imposed upon it.

The fields st_netdev and st_netino are not supported.

SEE ALSO

touch(1), chmod(2), chown(2), creat(2), link(2), mknod(2), pipe(2), read(2), time(2), truncate(2), unlink(2), utime(2), write(2), stat(5).

NAME
stime – set time and date

SYNOPSIS
int stime (tp)
long *tp;

DESCRIPTION
Stime sets the system's idea of the time and date. *Tp* points to the value of time as measured in seconds from 00:00:00 GMT January 1, 1970.

RETURN VALUE
Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS
[EPERM]	*Stime* will fail if the effective user ID of the calling process is not super-user.

HARDWARE DEPENDENCIES
Integral PC
Normal users have all super-user capabilities.

SEE ALSO
date(1), gettimeofday(2), time(2).

NAME

stty, gtty – control device

SYNOPSIS

```
#include <sgtty.h>

stty(fildes,argp)
int fildes;
struct sgttyb *argp;

gtty(fildes,argp)
int fildes;
struct sgttyb *argp;
```

REMARKS

These system calls are preserved for backward compatibility with Bell Version 6. They provide as close an approximation as possible to the old Version 6 functions. All new code should use the TCSETA/TCGETA *ioctl* calls described in *termio*(7).

DESCRIPTION

For certain status setting and status inquiries about terminal devices, the functions *stty* and *gtty* are equivalent to

```
ioctl(fildes, TIOCSETP, argp)
ioctl(fildes, TIOCGETP, argp)
```

respectively; see *sttyV6*(7) and *termio*(7).

RETURNS

Zero is returned if the call was successful; -1 if the file descriptor does not refer to the kind of file for which it was intended.

SEE ALSO

stty(1), exec(2), tty(7), sttyV6(7), termio(7).

NAME

swapon – add a swap device for interleaved paging/swapping

SYNOPSIS

swapon (special)
char *special;

DESCRIPTION

Swapon makes the block device *special* available to the system for allocation for paging and swapping. The names of potentially available devices are known to the system and defined at system configuration time. See the *System Administrator's Manual* for information on how the size of the swap area is calculated.

Swapon may be invoked only by the super-user.

RETURNS

Swapon will fail if one or more of the following are true:

[ENOTBLK]	*Special* is not the name of a block special file.
[ENXIO]	The device associated with *special* could not be opened.
[EBUSY]	The device associated with *special* is already in use.
[ENODEV]	The device associated with *special* does not exist.
[EPERM]	The effective user ID is not super-user.

WARNINGS

There is no way to stop swapping on a disk so that the pack may be dismounted.

HARDWARE DEPENDENCIES

The HFS file system is implemented on Series 200 beginning with HP-UX Release 5.0, and on Series 300 and Series 800.

AUTHOR

Swapon was developed by the University of California, Berkeley.

SEE ALSO

swapon(1M).

NAME

sync – update super-block

SYNOPSIS

void sync ( )

DESCRIPTION

Sync causes all information in memory that should be on disk to be written out. This includes modified super blocks, modified inodes, and delayed block I/O.

It should be used by programs which examine a file system, for example *fsck*, *df*, etc. It is mandatory before a shutdown.

The writing, although scheduled, is not necessarily complete upon return from *sync*.

In some HP-UX systems, *sync* may be reduced to a no-op. This is permissible on a system which does not cache buffers, or in a system that in some way ensures that the disks are always in a consistent state.

SEE ALSO

sync(1M).

NAME

time – get time

SYNOPSIS

long time ((long *) 0)

long time (tloc)
long *tloc;

DESCRIPTION

Time returns the value of time in seconds since 00:00:00 GMT, January 1, 1970.

If *tloc* (taken as an integer) is non-zero, the return value is also stored in the location to which *tloc* points.

ERRORS

[EFAULT] *Time* will fail if *tloc* points to an illegal address. The reliable detection of this error will be implementation dependent.

RETURN VALUE

Upon successful completion, *time* returns the value of time. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

date(1), gettimeofday(2), stime(2).

NAME

times – get process and child process times

SYNOPSIS

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/times.h>

long times (buffer)
struct tms *buffer;
```

DESCRIPTION

Times fills the structure pointed to by *buffer* with time-accounting information. The structure defined in **sys/times.h** is as follows:

```
struct tms {
    time_t  tms_utime;    /* user time */
    time_t  tms_stime;    /* system time */
    time_t  tms_cutime;   /* user time, children */
    time_t  tms_cstime;   /* system time, children */
};
```

This information comes from the calling process and each of its terminated child processes for which it has executed a *wait*. The times are in units of 1/HZ seconds, where HZ is processor dependent (see **<sys/param.h>**).

Tms_utime is the CPU time used while executing instructions in the user space of the calling process.

Tms_stime is the CPU time used by the system on behalf of the calling process.

Tms_cutime is the sum of the *tms_utime*s and *tms_cutime*s of the child processes.

Tms_cstime is the sum of the *tms_stime*s and *tms_cstime*s of the child processes.

ERRORS

[EFAULT]	*Times* will fail if *buffer* points to an illegal address. The reliable detection of this error will be implementation dependent.

HARDWARE DEPENDENCIES

Series 500:

For computers with multiple CPU's, the child CPU times listed can be greater than the actual elapsed real time, since the CPU time is counted on a per-CPU basis. Thus, if all three CPUs are executing, the CPU time is the sum of the three execution times of the CPUs.

RETURN VALUE

Upon successful completion, *times* returns the elapsed real time, in units of 1/HZ of a second, since an arbitrary point in the past (e.g., system start-up time). This point does not change from one invocation of *times* to another. If *times* fails, a −1 is returned and *errno* is set to indicate the error.

SEE ALSO

time(1), gettimeofday(2), exec(2), fork(2), time(2), wait(2).

BUGS

Not all CPU time expended by system processes on behalf of a user process is counted in the system CPU time for that process.

NAME

trapno – hardware trap numbers

Remarks:

The following description of hardware trap numbers is valid for the Series 500 only.

DESCRIPTION

The following trap numbers refer to hardware traps occurring on the HP 9000 Series 500 computers. *Trapno* values are reported by the *err*(1) command, and are passed to signal handlers (see *signal*(2)) when hardware traps cause signals to be sent to the current process.

The *trapno* value, trap name, and description are listed below for each possible trap condition. By convention, trap numbers are shown in octal.

VALUE	NAME: DESCRIPTION
01	Bounds Violation: An address is outside the limits for the program, stack, or global data segments. [2]
02	Check Trap: A user value is outside a prescribed range. [1]
03	Breakpoint Trap: Debugging trap. [1]
04	Machine Instruction Trap: Used by the operating system.
05	String Trap: Illegal string operation or data. [2]
06	Unused.
07	Unused.
010	Reset: Used by the operating system.
011	Page Table Violation: The page table entry referenced is beyond the current length of the page table. [2]
012	Inconsistent Registers: An attempt was made to set up an inconsistent set of registers describing the global data segment, stack segment, or program segment. [2]
013	External Data Segment Bounds Violation: An address is outside the limits of an external data segment. [2]
014	System Error: Used by the operating system.
015	External Data Segment Pointer Violation: Illegal data segment pointer; probably a pointer between 0 and 524287 decimal. [2]
016	Pointer Conversion Violation: An attempt was made to form a data segment pointer with an offset which is too large for the type of pointer being used. [2]
017	External Program Pointer Violation: Illegal procedure pointer. [2]
020	Unimplemented Instruction: Attempt to execute an undefined instruction. [1]
021	STT Violation: Illegal procedure pointer. [2]
022	CST Violation: Illegal procedure pointer. [2]
023	DST Violation: Illegal segment number in an external data segment pointer. [2]
024	Stack Overflow: The operating system normally handles this trap by extending the stack segment.
025	Stack Underflow: An attempt to pop a word from the local stack when the local stack is empty. [2]
026	Privileged Mode Violation: An attempt to execute a privileged instruction or return to a privileged procedure while in unprivileged mode. [2]

027	Privileged Mode Data Violation: An attempt to reference a privileged data segment while in unprivileged mode. [2]
030	Unexpected Pointer Type: An instruction has encountered a pointer type which it cannot handle. [2]
031	User Traps: Integer divide by zero. [1]
032	Illegal Decimal Number: A decimal math instruction has been supplied an illegal operand. [2]
033	Exponent Size Trap: Exponent too large during a number conversion instruction. [2]
034	Floating Point Operand Trap: Attempt to operate on illegal numbers, divide by zero, or convert a 64-bit number to a 32-bit number which cannot accommodate the exponent. [1]
035	Floating Point Result Trap: Floating point overflow; also caused by an explicit request to trap on an inexact result. [1]
036	Unexpected External Data Segment Type: A paged external data segment was encountered when an unpaged segment was expected, or vice versa. [2]
037	Absent Code Segment: Handled by the operating system.
040	Absent Page: Handled by the operating system.
041	Uncallable Procedure: Attempt to call an uncallable privileged procedure while in unprivileged mode. [2]
042	Absent Data Segment: Handled by the operating system.
043	Absent Page Table: Handled by the operating system.
044	Start-of-Line: Debugging trap. [1]
045	Variable Trace: Debugging trap. [1]
046	Start-of-Procedure: Debugging trap. [1]
047	End-of-Procedure: Debugging trap. [1]
050	Start-of-Subroutine: Debugging trap. [1]
051	End-of-Subroutine: Debugging trap. [1]
052	Code Segment Violation: Attempt to modify a code segment. [2]
053	Branch Violation: Illegal branch instruction. [2]
054	Message Trap: Used internally by the operating system.
055	Instruction Sequencing Bounds Violation: Program destination is out of bounds; probably a stack marker has been incorrectly modified.
056	Start-of-Line-Check Trap: Debugging trap. [1]
057	Data Segment Write Violation: Attempt to modify a write-protected data segment. [2]
060	System semaphore trap on up; relative pointer. [1]
061	System semaphore trap on up; absolute pointer. [1]
062	System semaphore trap on down; relative pointer. [1]
063	System semaphore trap on down; absolute pointer. [1]
064	Invalid internal math transformation. [1]

The footnotes are as follows:

[1]:	If the program returns from the trap (signal) handler, execution will resume with the next instruction.
[2]:	If the program returns from the trap (signal) handler, execution will resume at the current instruction.

SEE ALSO

err(1), signal(2).

WARNING

Trapno is intended for diagnostic purposes only. Values and meanings may change in future releases of HP-UX.

NAME

truncate, ftruncate – truncate a file to a specified length

SYNOPSIS

truncate(path, length)
char *path;
unsigned long length;

ftruncate(fd, length)
int fd;
unsigned long length;

DESCRIPTION

Truncate causes the file named by *path* or referenced by *fd* to be truncated to at most *length* bytes in size. If the file previously was larger than this size, the extra data is lost. With *ftruncate*, the file must be open for writing; for *truncate* the user must have write permission for the file.

RETURN VALUES

A value of 0 is returned if the call succeeds. If the call fails a -1 is returned, and the global variable *errno* specifies the error.

ERRORS

Truncate succeeds unless:

[ENOENT]	The pathname was too long.
[ENOTDIR]	A component of the path prefix of *path* is not a directory.
[ENOENT]	A component of the path name is too long.
[EACCES]	A component of the path prefix denies search permission.
[EACCES]	Write permission is denied on the file.
[EISDIR]	The named file is a directory.
[EROFS]	The named file resides on a read-only file system.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed.
[EFAULT]	*Path* points outside the process's allocated address space. The reliable detection of this error will be implementation dependent.
[ENAMETOOLONG]	The path name exceeds MAXPATHLEN characters.

Ftruncate succeeds unless:

[EBADF]	The *fd* is not a valid descriptor.
[EINVAL]	The *fd* references a file that was opened without write permission.

AUTHOR

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

SEE ALSO

open(2).

BUGS

Partial blocks discarded as the result of truncation are not zero filled; this can result in holes in files which do not read as zero.

NAME

ulimit – get and set user limits

SYNOPSIS

long ulimit (cmd, newlimit)
int cmd;
long newlimit;

DESCRIPTION

This function provides for control over process limits. The *cmd* values available are:

1 Get the file size limit of the process. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.

2 Set the file size limit of the process to the value of *newlimit*. Any process may decrease this limit, but only a process with an effective user ID of super-user may increase the limit. Note that the limit must be specified in units of 512-byte blocks.

3 Get the maximum possible break value. See *brk*(2). Depending on system resources such as swap space, this maximum may not be attainable at a given time.

ERRORS

Ulimit will fail if one or more of the following conditions is true.

[EINVAL]	*cmd* is not in the correct range.
[EPERM]	*Ulimit* will fail and the limit will be unchanged if a process with an effective user ID other than super-user attempts to increase its file size limit.

RETURN VALUE

Upon successful completion, a non-negative value is returned. Errors return a -1, with *errno* set appropriately.

SEE ALSO

brk(2), write(2).

NAME

umask – set and get file creation mask

SYNOPSIS

int umask (cmask)
int cmask;

DESCRIPTION

Umask sets the process's file mode creation mask to *cmask* and returns the previous value of the mask. Only the low-order 9 bits of *cmask* and the file mode creation mask are used.

The bits that are set in *cmask* specify which permission bits to turn off in the mode of the created file. For example, suppose a value of 007 is specified for *cmask*. Then, if a file is normally created with permissions of 0777, its mode after creation would be 0770.

RETURN VALUE

The previous value of the file mode creation mask is returned.

SEE ALSO

mkdir(1), mknod(1M), sh(1), chmod(2), creat(2), mknod(2), open(2).

NAME

umount – unmount a file system

SYNOPSIS

int umount (spec)
char *spec;

DESCRIPTION

Umount requests that a previously mounted file system contained on the block special device identified by *spec* be unmounted. *Spec* is a pointer to a path name. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

Umount may be invoked only by the super-user.

ERRORS

Umount will fail if one or more of the following are true:

[EPERM]	The process's effective user ID is not super-user.
[ENOENT]	*Spec* does not exist.
[ENOTBLK]	*Spec* is not a block special device.
[EINVAL]	*Spec* is not mounted.
[EBUSY]	A file on *spec* is busy.
[EFAULT]	*Spec* points outside the process's allocated address space. The reliable detection of this error will be implementation dependent.
[ENXIO]	The device associated with *spec* does not exist.
[ENOTDIR]	A component of *spec* is not a directory.
[ENOENT]	*Spec* is null.
[ENAMETOOLONG]	*Spec* exceeds MAXPATHLEN characters.

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

HARDWARE DEPENDENCIES

Integral Personal Computer:

For superuser capabilities described above, it is not necessary to be superuser.

SEE ALSO

mount(1M), mount(2).

BUGS

If *umount* is called from the program level (i.e. not from the *mount*(1M) level), the table of mounted devices contained in /etc/mnttab is not updated.

NAME

uname – get name of current HP-UX system

SYNOPSIS

#include <sys/utsname.h>

int uname (name)
struct utsname *name;

DESCRIPTION

Uname stores information identifying the current HP-UX system in the structure pointed to by *name*.

Uname uses the structure defined in **<sys/utsname.h>** whose members are:

```
#define UTSLEN    9
#define SNLEN     15

char              sysname[UTSLEN];
char              nodename[UTSLEN];
char              release[UTSLEN];
char              version[UTSLEN];
char              machine[UTSLEN];
char              idnumber[SNLEN];
```

Uname returns a null-terminated string in each field. *Sysname* contains "HP-UX". Similarly, *nodename* contains the name that the system is known by on a communications network and is accessible via *hostname*(1), *sethostname*(2), and *gethostname*(2). *Release* contains the release number of the operating system, e.g. "1.0" or "3.0.1". *Version* contains additional information about the operating system. The first character of the version field is set to "A" for single user systems, "B" for 16-user systems, "C" for 32-user systems, and "D" for 64-user systems. (Note that the contents of the version field may change on future releases as AT&T license agreement restrictions change.) *Machine* contains a standard name that identifies the hardware on which the *idnumber* contains an identification number which is unique within that class of hardware, possibly a hardware or software serial number. This field may return the null string to indicate the lack of an identification number.

ERRORS

[EFAULT]	*Uname* will fail if *name* points to an invalid address. The reliable detection of this error will be implementation dependent.

RETURN VALUE

Upon successful completion, a non-negative value is returned. Otherwise, –1 is returned and *errno* is set to indicate the error.

AUTHOR

Uname was developed by AT&T Bell Laboratories and the Hewlett-Packard Company.

SEE ALSO

hostname(1), uname(1), gethostname(2), sethostname(2).

NAME

unlink – remove directory entry; delete file

SYNOPSIS

int unlink (path)
char *path;

DESCRIPTION

Unlink removes the directory entry named by the path name pointed to by *path*.

ERRORS

The named file is unlinked unless one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist (for example, *path* is null or a component of *path* does not exist).
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	Write permission is denied on the directory containing the link to be removed.
[EPERM]	The named file is a directory and the effective user ID of the process is not super-user.
[EBUSY]	The entry to be unlinked is the mount point for a mounted file system.
[ETXTBSY]	The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed.
[EROFS]	The directory entry to be unlinked is part of a read-only file system.
[EFAULT]	*Path* points outside the process's allocated address space. The reliable detection of this error will be implementation dependent.
[ENAMETOOLONG]	The named file exceeds MAXPATHLEN characters.

When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, the removal is postponed until all references to the file have been closed.

HARDWARE DEPENDENCIES

Series 500

The last link to a directory cannot be unlinked if the directory is not empty.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

rm(1), close(2), link(2), open(2).

NAME

ustat – get file system statistics

SYNOPSIS

```
#include <sys/types.h>
#include <ustat.h>

int ustat (dev, buf)
dev_t dev;
struct ustat *buf;
```

DESCRIPTION

Ustat returns information about a mounted file system. *Dev* is a device number identifying a device containing a mounted file system. *Buf* is a pointer to a *ustat* structure (defined in **ustat.h**) that includes the following elements:

```
daddr_t  f_tfree;      /* Total free blocks */
ino_t    f_tinode;     /* Number of free inodes */
char     f_fname[6];   /* Filsys name */
char     f_fpack[6];   /* Filsys pack name */
int      f_blksize;    /* Block size */
```

ERRORS

Ustat will fail if one or more of the following are true:

[EINVAL]	*Dev* is not the device number of a device containing a mounted file system.
[EFAULT]	*Buf* points outside the process's allocated address space. The reliable detection of this error will be implementation dependent.

HARDWARE DEPENDENCIES

Series 200, 300:

f_tfree and f_blksize are reported in fragment size units.

Series 500:

In the above structure,f_fname[6] is the driver name, not the file system name.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

AUTHOR

Ustat was developed by AT&T Bell Laboratories and the Hewlett-Packard Company.

SEE ALSO

touch(1), stat(2), fs(4).

NAME

utime - set file access and modification times

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

int utime (path, times)
char *path;
struct utimbuf *times;
```

DESCRIPTION

Path points to a path name naming a file. *Utime* sets the access and modification times of the named file.

If *times* is **NULL**, the access and modification times of the file are set to the current time. A process must be the owner of the file or have write permission to use *utime* in this manner.

If *times* is not **NULL**, *times* is interpreted as a pointer to a *utimbuf* structure and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or the super-user may use *utime* this way.

The times in the following structure, found in unistd.h, are measured in seconds since 00:00:00 GMT, Jan. 1, 1970.

```
struct utimbuf {
    time_t  actime;    /* access time */
    time_t  modtime;  /* modification time */
};
```

ERRORS

Utime will fail if one or more of the following are true:

[ENOENT] The named file does not exist.

[ENOTDIR] A component of the path prefix is not a directory.

[EACCES] Search permission is denied by a component of the path prefix.

[EPERM] The effective user ID is not super-user and not the owner of the file and *times* is not **NULL**.

[EACCES] The effective user ID is not super-user and not the owner of the file and *times* is **NULL** and write access is denied.

[EROFS] The file system containing the file is mounted read-only.

[EFAULT] *Times* is not **NULL** and points outside the process's allocated address space. The reliable detection of this error will be implementation dependent.

[EFAULT] *Path* points outside the process's allocated address space. The reliable detection of this error will be implementation dependent.

[ENAMETOOLONG] The named file exceeds MAXPATHLEN characters.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

HARDWARE DEPENDENCIES

Integral Personal Computer:

For superuser capabilities described above, it is not necessary to be superuser.

SEE ALSO

touch(1), stat(2).

NAME

vfork – spawn new process in a virtual memory efficient way

SYNOPSIS

int vfork()

REMARKS

Vfork is provided as a higher performance version of *fork* on those systems which choose to provide it and for which there is a performance advantage.

Vfork differs from *fork* only in that the child process may share code and data with the calling process (parent process). This speeds the cloning activity significantly at a risk to the integrity of the parent process if *vfork* is misused.

The use of *vfork* for any purpose except as a prelude to an immediate *exec* or *exit* is not supported. Any program which relies upon the differences between *fork* and *vfork* is not portable across HP-UX systems.

All implementations of HP-UX must provide the entry *vfork*, but it is permissible for them to treat it identically to *fork*. Some implementations may not choose to distinguish the two because their implementation of fork is as efficient as possible, and others may not wish to carry the added overhead of two similar calls.

DESCRIPTION

Vfork can be used to create new processes without fully copying the address space of the old process. If a forked process is simply going to do an *exec*(2), the data space copied from the parent to the child by *fork*(2) is not used. This is particularly inefficient in a paged environment. *Vfork* is useful in this case. Depending upon the size of the parent's data space, it can give a significant performance improvement over *fork*.

Vfork differs from *fork* in that the child borrows the parent's memory and thread of control until a call to *exec* or an exit (either by a call to *exit*(2) or abnormally.) The parent process is suspended while the child is using its resources.

Vfork returns 0 in the child's context and (later) the pid of the child in the parent's context.

Vfork can normally be used just like *fork*. It does not work, however, to return while running in the child's context from the procedure which called *vfork* since the eventual return from *vfork* would then return to a no longer existent stack frame. Be careful, also, to call *_exit* rather than *exit* if you cannot *exec*, since *exit* will flush and close standard I/O channels, and thereby mess up the parent process's standard I/O data structures. (Even with *fork* it is wrong to call *exit* since buffered data would then be flushed twice.)

The [vfork,exec] window begins at the *vfork* call and ends when the child completes its *exec* call.

RETURN VALUE

Upon successful completion, *vfork* returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of –1 is returned to the parent, no child process is created, and *errno* is set to indicate the error.

ERRORS

Vfork will fail and no child process will be created if one or more of the following are true:

[EAGAIN]	The system-wide limit on the total number of processes under execution would be exceeded.
[EAGAIN]	The system-imposed limit on the total number of processes under execution by a single user would be exceeded.

HARDWARE DEPENDENCIES

Series 200, 300, 800

A call to *signal*(2) in the [vfork,exec] window which is used to catch a signal can affect handling of the signal by the parent. This is not true if the signal is set SIG_DFL or SIG_IGN, or if *sigvector*(2) is used.

Series 500

Shared memory segments generated with the *EMS* intrinsics will be inherited over *vfork*. Private memory segments will not be copied over *vfork*.

Vfork will also fail in the following cases:

[ENOMEM]	There is not enough physical memory to create the new process.
[EAGAIN]	The child process attempts to do a second *vfork* or a *fork* while in the [vfork,exec] window.

The parent and child processes share the same stack space within the [vfork,exec] window. If the size of the stack has been changed within this window by the child process (return from or call to a function, for example), it is likely that the parent and child processes will be killed with signal SIGSEGV.

Series 500, 800

Process times for the parent and child processes within the [vfork,exec] window may be inaccurate.

Series 800

The parent and child processes share the same stack space within the [vfork,exec] window. If the size of the stack has been changed within this window by the child process (return from or call to a function, for example), it is likely that the parent and child processes will be killed with signal SIGSEGV or SIGBUS.

Integral PC

Vforked children have a unique 2K-byte stack allocated to them. Any stack space used beyond this 2K limit is shared between the child and the parent. *Vfork* does not work with shared text programs. Also, to access *vfork* on the Integral PC, one must link in **/usr/lib/librt.a** with the program at compile or load time.

AUTHOR

Vfork was developed by the University of California, Berkeley.

SEE ALSO

exec(2), exit(2), fork(2), wait(2).

NAME

vsadv – advise system about backing store usage

SYNOPSIS

#include <sys/ems.h>

vsadv (index);
int index;

DESCRIPTION

This call requests that all future backing store space allocated for this process be placed on the backing store device specified by *index* (see *vson*(2)). It may be used to tune an application to the local system environment. This request remains in effect until the next call to *vsadv* by this process. An *index* of -1 will set backing store allocation back to the default system policy.

This call is advisory in nature and will never cause subsequent program failures (e.g. if the device has no room, the system will simply override the request and use another device).

This characteristic is inherited across *fork*(2) and *exec*(2).

This call may be reduced to a no-op.

HARDWARE DEPENDENCIES

Implemented on Series 500 only.

AUTHOR

Vsadv was developed by the Hewlett-Packard Company.

SEE ALSO

ems(2), vson(2).

NAME

vson, vsoff – advise OS about backing store devices

SYNOPSIS

```
#include <sys/ems.h>

int vson(pathname, size, q);
int size, q;
char *pathname;

int vsoff(index, force);
int index, force;
```

DESCRIPTION

Vson is used to make the block special file *pathname* available for use by the system as a backing store device for whatever form of backing store is needed by the system. The call returns an id by which the backing store device may be referenced in subsequent *vsoff* or *vsadv*(2) calls. Multiple *vson* calls for the same device will return the same id (here "same device" means identical devno - major and minor - and not necessarily the same file name).

Pathname specifies a block special device file, which may or may not contain a mounted file system, and which must be a CS-80 device. If device does not contain a file system (i.e. an "empty" disk), *size* specifies the available backing storage space (in blocks) to be made available (the storage space is assumed to start at block 0 in this case). If *size* is set to −1 and the device does not contain a file system, the whole block special device will be used for backing store.

Q is a quality (i.e. performance) factor for the device. It is used by the system in load balancing decisions. Higher values suggest secondary choices for backing store devices. There is no inherent significance to the value of *q* other than its value relative to the *q* factor of the other devices in the list. This parameter may be ignored on some implementations.

Vsoff is used to remove a device from the list of backing store devices available to the system. *Index* is the value returned by *vson* when the device was added to the list.

If *force* is not set (i.e. is 0) the system attempts to "gracefully" eliminate backing store usage of device by migrating backing store space onto other devices. If *force* is set (if, for instance, the device has failed) no attempt is made to salvage images stored on the disk. Processes with images on the device will, in all probability, be rather ungracefully terminated in the near future (i.e. when the images are required).

Only the super-user may add or remove backing store devices. A normal user may call *vson* to get the id for a device already known to the system as a backing store device (for subsequent use in a *vsadv*(2) call).

RETURN VALUES

Upon successful completion, *vson* returns the index for the device and *vsoff* returns 0. If there is an error, a value of −1 is returned and *errno* is set to indicate the error.

AUTHOR

Vson was developed by the Hewlett-Packard Company.

SEE ALSO

ems(2), memallc(2), swapon(2), vsadv(2)

NAME

wait – wait for child process to stop or terminate

SYNOPSIS

int wait (stat_loc)
int *stat_loc;

int wait ((int *)0)

#include <sys/wait.h>
int wait3 (stat_loc, options, (int *)0)
int *stat_loc;
int options;

DESCRIPTION

Wait suspends the calling process until one of the immediate children terminates or until a child that is being traced stops, because it has hit a break point. The *wait* system call will return prematurely if a signal is received. If a child process stopped or terminated prior to the call on *wait*, return is immediate.

If *stat_loc* (taken as an integer) is non-zero, 16 bits of information called status are stored in the low order 16 bits of the location pointed to by *stat_loc*. *Status* can be used to differentiate between stopped and terminated child processes. If the child process is terminated, status identifies the cause of termination and passes useful information to the parent. This is accomplished in the following manner:

> If the child process stopped, the high order 8 bits of status will contain the number of the signal that caused the process to stop and the low order 8 bits will be set equal to 0177.
>
> If the child process terminated due to an *exit* or *_exit* call, the low order 8 bits of status will be zero and the high order 8 bits will contain the low order 8 bits of the argument that the child process passed to *exit*; see *exit*(2).
>
> If the child process terminated due to a signal, the high order 8 bits of status will be zero and the low order 8 bits will contain the number of the signal that caused the termination. In addition, if the low order seventh bit (i.e., bit 0200) is set, a "core image" will have been produced; see *signal*(2).

If the *wait3* variant is used, then there are two options available for modifying the behavior of the system call. They may be combined by *or*ing them together. The first is WNOHANG which prevents *wait3* from suspending the calling process even if there are children to wait for. In this case, a value of zero is returned indicating there are no children which have stopped or died. If the second option WUNTRACED is set, then in addition to traced children which are stopped, *wait3* will also return information when children of the current process are stopped but not traced (with *ptrace(2))* because they received a SIGTTIN, SIGTTOU, SIGTSTP, or SIGSTOP signal.

The third parameter to *wait3* is currently unused and must always be a null pointer.

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means the initialization process inherits the child processes.

ERRORS

Wait will fail if one or more of the following are true:

[ECHILD]	The calling process has no existing unwaited-for child processes. In this case, *wait* returns immediately.
[EFAULT]	*Stat_loc* points to an illegal address. The reliable detection of this error will be implementation dependent.

[EINVAL] *Wait3* was passed a non-null pointer value for its third argument.

RETURN VALUE

If *wait* returns due to the receipt of a signal, a value of -1 is returned to the calling process and *errno* is set to EINTR. If *wait* returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. If *wait3* is called, the WNOHANG option is used, and there are no stopped or terminated children, then a value of zero is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

WARNINGS

The behavior of *wait* is affected by setting the SIGCLD signal to SIG_IGN. Check all references to *signal*(2) for appropriateness on systems that support *sigvector*(2). *Sigvector*(2) can affect that behavior described on this page. See *WARNINGS* on *signal*(2).

HARDWARE DEPENDENCIES

Series 200, 300, 500

Wait3 is not supported.

AUTHOR

Wait and *wait3* were developed by the Hewlett-Packard Company, AT&T Bell Laboratories, and the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

Exit conditions (**$?**) in sh(1), exec(2), exit(2), fork(2), pause(2), ptrace(2), signal(2).

NAME

write, writev – write on a file

SYNOPSIS

```
int write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

#include <sys/types.h>
#include <sys/uio.h>

int writev (fildes, iov, iovcnt)
int fildes;
struct iovec *iov;
int iovcnt;
```

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

Write attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes*. *Writev* performs the same action, but gathers the output data from the *iovlen* buffers specified by the elements of the *iovec* array: iov[0], iov[1], ..., iov[*iovcnt* - 1].

For *writev* the iovec structure is defined as:

```
struct iovec {
     caddr_t    iov_base;
     int        iov_len;
};
```

Each *iovec* entry specifies the base address and length of an area in memory where data should be copied from. The *iovec* array maybe at most *MAXIOV* long.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from *write*, the file pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the device's current position. The value of a file pointer associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set, the file pointer will be set to the end of the file prior to each write.

If a *write* requests that more bytes be written than there is room for (e.g., the *ulimit* (see *ulimit*(2)) or the physical end of a medium), only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes will return 20. The next write of a non-zero number of bytes will give a failure return (except as noted below).

A write to an ordinary file will be blocked if enforcement-mode file and record locking is set, and there is a lock owned by another process on the segment of the file to be written:

> If O_NDELAY is set, the write will return -1 and set errno to EAGAIN.
>
> If O_NDELAY is clear, the write will sleep until the blocking record lock is removed.

If the file being written is a pipe (or FIFO), there is a system dependent maximum number of bytes which it can store (PIPSIZ as defined in <sys/inode.h>). The minimum value of PIPSIZ on any HP-UX system is 4096. In writing a pipe, the following conditions apply:

If the O_NDELAY flag of the file flag word is set:

If *nbyte* is less than or equal to PIPSIZ and there is sufficient room in the pipe or FIFO, then the *write* is successful and returns the number of bytes written;

If *nbyte* is less than or equal to PIPSIZ but there is not enough room in the pipe or FIFO, the *write* returns without error, having written nothing, and with a return value of 0.

If *nbyte* is greater than PIPSIZ the *write* fails and returns -1. [EINVAL]

If the O_NDELAY flag of the file flag word is clear:

the *write* always executes correctly (blocking as necessary) and returns the number of bytes written.

RETURN VALUE

Upon successful completion the number of bytes actually written is returned. Otherwise, -1 is returned and *errno* is set to indicate the error.

EXAMPLES

Assuming a process opened a file for writing, the following call to *write*(2) attempts to write *mybufsize* bytes to the file from the buffrer pointed to by *mybuf*.

```
#include <string.h>

int mybufsize;
char *mybuf = "aeiou and sometimes y";
int nbytes;

            mybufsize = strlen (mybuf);
            nbytes = write (outfd, mybuf, mybufsize);
```

ERRORS

Write will fail and the file pointer will remain unchanged if one of the following conditions is true and **errno** will be set accordingly:

[EBADF]	*Fildes* is not a valid file descriptor open for writing.
[EPIPE and SIGPIPE signal]	An attempt is made to write to a pipe that is not open for reading by any process.
[EINTR]	A signal was caught during the *write* system call.
[EDEADLK]	A resource deadlock would occur as a result of this operation (see *lockf*(2) and *fcntl*(2)).
[EAGAIN]	Enforcement-mode file and record locking was set, O_NDELAY was set, and there was a blocking record lock.
[ENOLCK]	The system record lock table was full, so the write could not go to sleep until the blocking record lock was removed.

In addition, *writev* may return one of the following errors:

[EFAULT]	*Iov_base* or *iov* points outside of the allocated address space. The reliable detection of this error will be implementation dependent.
[EINVAL]	*Iovcnt* was less than or equal to 0, or greater then *MAXIOV*.
[EINVAL]	One of the iov len values in the iov array was negative.
[EINVAL]	The sum of iov len values in the iov array overflowed a 32-bit integer.

[ENOSPC] Not enough space on the file system.

Write or *writev* will fail and the file pointer will be updated to reflect the amount of data transferred if one of the following conditions is true and errno will be set accordingly:

[EFBIG] An attempt was made to write a file that exceeds the process's file size limit or the maximum file size. See *ulimit*(2).

[EFAULT] *Buf* points outside the process's allocated address space. The reliable detection of this error will be implementation dependent.

WARNINGS

Check all references to *signal*(2) for appropriateness on systems that support *sigvector*(2). *Sigvector*(2) can affect the behavior described on this page.

The character special devices, and raw disks in particular, apply constraints on how *write* can be used. See the specific DEV entries for details on particular devices.

HARDWARE DEPENDENCIES

Series 500

If you perform a write operation following an *lseek* past the previous end-of-file, all "unused" bytes from the previous end-of-file up to your new position are zeroed-out before writing your data.

In general, a value of *nbyte* greater than 512K is not supported when *fildes* is associated with a device file. There are two exceptions to this:

the device is a terminal or the null device; or

buf points to a local (not global) buffer, and has been locked with *memlck*(2). A local buffer is an array that is declared within the procedure and resides on the stack.

Any request for greater than 512K megabytes on unsupported device files results in errno being set to EINVAL. Requests for less than 512K megabytes could result in errno being set to ENOMEM.

Writev is not currently supported.

The size of a pipe (PIPSIZ) is currently 5120 bytes.

Integral PC

Under the conditions for whigh O_NDELAY is set, *nbyte* can be less than or equal to 10240 bytes.

Writev is not currently supported.

Series 200, 300, 800

The size of a pipe (PIPSIZ) is currently 8192.

AUTHOR

Write was developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

creat(2), dup(2), lockf(2), lseek(2), open(2), pipe(2), ulimit(2), ustat(2).

NAME

intro – introduction to subroutines and libraries

SYNOPSIS

#include <stdio.h>

#include <math.h>

DESCRIPTION

This section describes functions found in various libraries, other than those functions that directly invoke HP-UX system primitives, which are described in Section (2) of this volume. Certain major collections are identified by a letter after the section identifier (3):

(3C)	These functions, together with the Operating System Calls and those marked (3S), constitute the Standard C Library, which is automatically loaded by the C compiler, *cc*(1). The link editor *ld*(1) searches this library under the **–lc** option. Declarations for some of these functions may be obtained from **#include** files indicated on the appropriate pages.
(3I)	These functions constitute the instrument support library.
(3M)	These functions constitute the Math Library. They are automatically loaded as needed by the FORTRAN compiler *f77*(1). They are not automatically loaded by the C compiler, *cc*(1); however, the link editor searches this library under the **–lm** option. Declarations for these functions may be obtained from the **#include** file **<math.h>**. Several generally useful mathematical constants are also defined there (see *math*(5)).
(3S)	These functions constitute the "standard I/O package" (see *stdio*(3S)). These functions are in the library *libc*, already mentioned. Declarations for these functions may be obtained from the **#include** file **<stdio.h>**.
(3X)	Various specialized libraries. The files in which these libraries are found are given on the appropriate pages.

Definitions

A *character* is any bit pattern able to fit into a byte on the machine. The *null character* is a character with value 0, represented in the C language as "\0". A *character array* is a sequence of characters. A *null-terminated character array* is a sequence of characters, the last of which is the *null character*. A *string* is a designation for a *null-terminated character array*. The *null string* is a character array containing only the null character. A **NULL** pointer is the value that is obtained by casting **0** into a pointer. The C language guarantees that this value will not match that of any legitimate pointer, so many functions that return pointers return it to indicate an error. **NULL** is defined as **0** in **<stdio.h>**; the user can include an appropriate definition if not using **<stdio.h>**.

Many groups of FORTRAN intrinsic functions have generic function names that do not require explicit or implicit type declaration. The type of the function will be determined by the type of its argument(s). For example, the generic function *max* will return an integer value if given integer arguments (*max0*), a real value if given real arguments (*amax1*), or a double-precision value if given double-precision arguments (*dmax1*).

DIAGNOSTICS

Functions in the C and Math Libraries, (3C) and (3M), may return the conventional values **0** or **±HUGE** (the largest-magnitude single-precision floating-point numbers; **HUGE** is defined in the **<math.h>** header file) when the function is undefined for the given arguments or when the value is not representable. In these cases, the external variable *errno* (see *errno*(2)) is set to the value EDOM or ERANGE. As many of the FORTRAN intrinsic functions use the routines found in the Math Library, the same conventions apply.

WARNINGS

Many of the functions in the libraries call and/or refer to other functions and external variables described in this section and in Section (2), Operating Systems Calls. If a program inadvertently defines a function or external variable with the same name, the presumed library version of the function or external variable may not be loaded. The *lint*(1) program checker reports name conflicts of this kind as "multiple declarations" of the names in question. Definitions for the Sections (2), (3C), and (3S) are checked automatically. Other definitions can be included by using the −l option (for example, −l*m* includes definitions for the Math Library, (3M). Use of *lint*(1) is highly recommended.

FILES

/lib/libc.a
/lib/libm.a
/usr/lib/libF77.a

SEE ALSO

intro(2), stdio(3S), math(5), hier(5), ar(1), cc(1), f77(1), ld(1), lint(1), nm(1).

The introduction to this manual.

NAME

a64l, l64a – convert between long integer and base-64 ASCII string

SYNOPSIS

long a64l (s)
char *s;

char *l64a (l)
long l;

DESCRIPTION

These functions are used to maintain numbers stored in *base-64* ASCII characters. This is a notation by which long integers can be represented by up to six characters; each character represents a "digit" in a radix-64 notation.

The characters used to represent "digits" are **.** for 0, **/** for 1, **0** through **9** for 2–11, **A** through **Z** for 12–37, and **a** through **z** for 38–63.

The leftmost character is the least significant digit. For example,

$$\text{a0} = (38 \times 64^0) + (2 \times 64^1) = 166$$

A64l takes a pointer to a null-terminated base-64 representation and returns a corresponding **long** value. If the string pointed to by *s* contains more than six characters, *a64l* will use the first six.

L64a takes a **long** argument and returns a pointer to the corresponding base-64 representation. If the argument is 0, *l64a* returns a pointer to a null string.

BUGS

The value returned by *l64a* is a pointer into a static buffer, the contents of which are overwritten by each call.

NAME

abort – generate an IOT fault

SYNOPSIS

int abort ( )

DESCRIPTION

Abort first closes all open files if possible, then causes the SIGIOT signal to be sent to the process. This usually results in termination with a core dump.

It is possible for *abort* to return control if **SIGIOT** is caught or ignored, in which case the value returned is that of the *kill*(2) system call.

SEE ALSO

adb(1), exit(2), kill(2), signal(2).

DIAGNOSTICS

If **SIGIOT** is neither caught nor ignored, and the current directory is writable, a core dump is produced and the message "abort – core dumped" is written by the shell.

NAME

abs – return integer absolute value

SYNOPSIS

```
int abs (i)
int i;
```

DESCRIPTION

Abs returns the absolute value of its integer operand.

The largest negative integer returns itself.

WARNINGS

In two's-complement representation, the absolute value of the negative integer with largest magnitude is undefined. Some implementations trap this error, but others simply ignore it.

SEE ALSO

floor(3M).

NAME

assert – verify program assertion

SYNOPSIS

#include <assert.h>

assert (expression)
int expression;

DESCRIPTION

This macro is useful for putting diagnostics into programs. When it is executed, if *expression* is false (zero), *assert* prints

"Assertion failed: *expression*, file *xyz*, line *nnn*"

on the standard error output and aborts. In the error message, *xyz* is the name of the source file and *nnn* the source line number of the *assert* statement.

Compiling with the preprocessor option **-DNDEBUG** (see *cpp*(1)), or with the preprocessor control statement "**#define** NDEBUG" ahead of the "**#include** <assert.h>" statement, will stop assertions from being compiled into the program.

SEE ALSO

cpp(1), abort(3C).

NAME

j0, j1, jn, y0, y1, yn – Bessel functions

SYNOPSIS

```
#include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

double jn (n, x)
int n;
double x;

double y0 (x)
double x;

double y1 (x)
double x;

double yn (n, x)
int n;
double x;
```

DESCRIPTION

J0 and *j1* return Bessel functions of x of the first kind of orders 0 and 1 respectively. *Jn* returns the Bessel function of x of the first kind of order n.

Y0 and *y1* return the Bessel functions of x of the second kind of orders 0 and 1 respectively. *Yn* returns the Bessel function of x of the second kind of order n. The value of x must be positive.

DIAGNOSTICS

Non-positive arguments cause *y0*, *y1* and *yn* to return the value **–HUGE** and to set *errno* to **EDOM**. They also cause a message indicating DOMAIN error to be printed on the standard error output; the process will continue.

Arguments too large in magnitude cause *j0*, *j1*, *jn*, *y0*, *y1* and *yn* to return zero and to set *errno* to **ERANGE**. In addition, a message indicating TLOSS error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M).

NAME

blmode – terminal block mode library interface

SYNOPSIS

```
#include <sys/blmodeio.h>

int bfdes;

bfdes = blopen(fildes)
int fildes;

int blclose (bfdes)
int bfdes;

int blread (bfdes, buf, nbyte)
int bfdes;
char *buf;
unsigned nbyte;

int blget (bfdes, arg)
int bfdes;
struct blmodeio *arg;

int blset (bfdes, arg)
int bfdes;
struct blmodeio *arg;
```

DESCRIPTION

This terminal library interface allows support of block mode transfers with HP terminals. Block mode only affects input processing. Therefore, data is written with the standard *write*(2) interface.

In character mode the terminal sends each character to the system as it is typed. However, in block mode data is buffered and possibly edited locally in the terminal memory as it is typed, then sent as a block of data when the **<ENTER>** key is pressed on the terminal. During block mode data transmissions, the incoming data is not echoed by the interface and no special character processing is performed, other than recognizing a data block terminator character. For subsequent character mode transmissions, the existing termio state (see *termio*(7)) will continue to determine echo and character processing.

There are two parts of the block mode protocol, the block mode handshake and the block mode transmission.

Block mode handshake

At the beginning of a read, a *trigger* character is sent to the terminal to notify it that the system wants a block of data. (The *trigger* character, if defined, is sent at the beginning of all reads, character or block mode. It is necessary for block mode reads to work correctly.)

After receiving the *trigger* character, and when the user has typed all the data into the terminal's memory and pressed the **<ENTER>** key, the terminal will send an *alert* character to the system to notify it that the terminal has a block of data to send.

The system may then send user-definable cursor positioning or other data sequences, such as for home cursor or lock keyboard, to the terminal.

The system will then send a second *trigger* character to the terminal. The terminal will then transmit the data block as described in the **Block mode transmission** section.

Block mode transmission

The second part of the block mode protocol is the block mode transmission. After the block mode handshake has successfully completed, the terminal will transmit the data block to the system. During this transmission of data, the incoming data is not echoed by the system and no special character processing is performed, other than recognizing the data block termination character. It is possible to bypass the block mode handshake and have the block mode transmission occur after only the first *trigger* character is sent, see CB_BMTRANS below.

It is possible to intermix both character mode and block mode data transmissions. If CB_BMTRANS (see below) is set, all transfers will be block mode transfers. When CB_BMTRANS is not set, character mode transmissions will be processed as described in *termio*(7). In this case, if an *alert* character is received anywhere in the input data, the transmission mode will be switched to block mode automatically for a single transmission. Any data received before the *alert* will be discarded. The *alert* character may be escaped with a backslash ("\") character.

XON/XOFF flow control

To prevent data loss, XON/XOFF flow control should be used between the system and the terminal. The IXOFF bit (see *termio*(7)) should be set and the terminal strapped appropriately. If flow control is not used, it is possible for incoming data to overflow and be lost. (Note: some older terminals do not support this flow control.)

Read requests

Read requests that receive data from block mode transmissions will not be returned until the transmission is complete (the terminal has transmitted all characters). If the read is satisfied by byte count or if a data transmission error occurs, all subsequent data will be discarded until the transmission is complete. The read will wait until a terminator character is seen, or a time interval specified by the system has passed that is longer than necessary for the number of characters specified.

The data block terminator character will be included in the data returned to the user, and is included in the byte count. If the number of bytes transferred by the terminal in a block mode transfer exceeds the number of bytes requested by the user, the read will return the requested number of bytes and the remaining bytes will be discarded. The user can determine if data was discarded by checking the last character of the returned data. If the last character is not the terminator character, then more data was received than was requested and data was discarded.

The EIO error can be caused by several events, including errors in transmission, framing, parity, break, and overrun, or if the internal timer expires. The internal timer starts when the second trigger character is sent by the computer, and ends when the terminating character is received by the computer. The length of this timer is determined by the number of bytes requested in the read and the current baud rate, plus an additional ten seconds.

User control of handshaking

If desired, the application program can provide its own handshake mechanism in response to the *alert* character by selecting the OWNTERM mode, see CB_OWNTERM below. With this mode selected, the driver will complete a read request when the *alert* character is received. No data will be discarded before the *alert,* and the *alert* will be returned in the data read. The *alert* character may be escaped with a backslash ("\") character. The second *trigger* will be sent when the application issues the next read.

Blmode control calls

First, the standard *open*(2) call to a tty device must be made to obtain a file descriptor for the subsequent block mode control calls (an *open*(2) will be done automatically by the system for *stdin* on the terminal).

```
int bfdes;

bfdes = blopen (fildes)
int fildes;
```

A call to *blopen* must be made before any block mode access is allowed on the specified file descriptor. *Blopen* will initialize the block mode parameters as described below. The return value from *blopen* is a block mode file descriptor that must be passed to all subsequent block mode control calls.

int blclose (bfdes)
int bfdes;

A call to *blclose* must be issued before the standard *close*(2) to ensure proper closure of the device. Otherwise unpredictable results may occur. The argument **bfdes** is the file descriptor returned from a previous *blopen* system call.

int blread (bfdes, buf, nbyte)
int bfdes;
char *buf;
unsigned nbyte;

The *blread* routine has the same parameters as the *read*(2) sytem call. At the beginning of a read, the *cb_trig1c* character (if defined) is sent to the device. If CB_BMTRANS is not set, and no *cb_alertc* character is received, the read data will be processed according to *termio*(7). If CB_BMTRANS is set, or if a non-escaped *cb_alertc* character is received, echo will be turned off for the duration of the transfer, and no further special character processing will be done other than that required for the termination character. The argument **bfdes** is the file descriptor returned from a previous *blopen* system call.

int blget (bfdes, arg)
int bfdes;
struct blmodeio *arg;

A call to *blget* will return the current values of the **blmodeio** structure (see below). The argument **bfdes** is the file descriptor returned from a previous *blopen* system call.

int blset (bfdes, arg)
int bfdes;
struct blmodeio *arg;

A call to *blset* will set the block mode values from the structure whose address is *arg*. The argument **bfdes** is the file descriptor returned from a previous *blopen* system call.

Blmode structure

The two block mode control calls, *blget* and *blset*, use the following structure, defined in **<sys/blmodeio.h>**:

```
#define   NBREPLY    64

struct    blmodeio       {
          unsigned long  cb_flags;             /* Modes */
          unsigned char  cb_trig1c;            /* First trigger */
          unsigned char  cb_trig2c;            /* Second trigger */
          unsigned char  cb_alertc;            /* Alert character */
          unsigned char  cb_termc;             /* Terminating char */
          unsigned char  cb_replen;            /* cb_reply length */
          char           cb_reply[NBREPLY];    /* optional reply */
};
```

The *cb_flags* field controls the basic block mode protocol:

CB_BMTRANS	0000001	Enable mandatory block mode transmission.
CB_OWNTERM	0000002	Enable user control of handshake.

If CB_BMTRANS is set, all transmissions are processed as block mode transmissions. The block mode handshake is not required and data read is processed as block mode transfer data. The block mode handshake may still be invoked by receipt of an *alert* character as the first character seen. A **blread** issued with the CB_BMTRANS bit set will cause any existing input buffer data to be flushed.

If CB_BMTRANS is not set, and if the *alert* character is defined and is detected anywhere in the input stream, the input buffer will be flushed and the block mode handshake will be invoked. The system will then send the *cb_trig2c* character to the terminal, and a block mode transfer will follow. The *alert* character can be escaped by preceding it with a backslash ("\").

If CB_OWNTERM is set, reads will be terminated upon receipt of a non-escaped *alert* character. No input buffer flushing is performed, and the *alert* character is returned in the data read. This allows application code to perform its own block mode handshaking. If the bit is clear, a non-escaped *alert* character will cause normal block mode handshaking to be used.

The initial *cb_flags* value is all-bits-cleared.

There are several special characters (both input and output) that are used with block mode. These characters and the initial values for these characters are described below. Any of these characters may be undefined by setting its value to 0377.

cb_trig1c	is the initial *trigger* character sent to the terminal at the beginning of a read request.
cb_trig2c	is the secondary *trigger* character sent to the terminal after the *alert* character has been seen.
cb_alertc	is the *alert* character sent by the terminal in response to the first *trigger* character. It signifies that the terminal is ready to send the data block. The *alert* character can be escaped by preceding it with a backslash ("\").
cb_termc	is sent by the terminal after the block mode transfer has completed. It signifies the end of the data block to the computer.

The *cb_replen* field specifies the length in bytes of the *cb_reply* field. If set to zero, the *cb_reply* string will not be used. The *cb_replen* field is initially set to zero.

The *cb_reply* array contains a string to be sent out after receipt of the *alert* character, but before the second *trigger* character is sent by the computer. Any character may be included in the reply string. The number of characters sent is specified by *cb_replen.* The initial value of all characters in the *cb_reply* array is NULL.

RETURNS

If an error occurs, all calls will return a value of -1 and *errno* will be set to indicate the error. If no error is detected, *blread* will return the number of characters read. All other calls will return 0 upon successful completion.

During a read, it is possible for the user's buffer to be altered even if an error value is returned. The data in the user's buffer should be ignored as it will not be complete. The following errors may be returned by various library calls described in this document.

blopen

[ENOTTY]	The file descriptor specified is not related to a terminal device.

blclose

[ENOTTY]	No previous **blopen** has been issued for the specified file descriptor.

blread

	[EDEADLK]	A resource deadlock would occur as a result of this operation (see *lockf*(2)).
	[EFAULT]	**Buf** points outside the allocated address space. The reliable detection of this error will be implementation dependent.
	[EINTR]	A signal was caught during the **read** system call.
	[EIO]	An I/O error occured during block mode data transmissions.
	[ENOTTY]	No previous **blopen** has been issued for the specified file descriptor.
blget		
	[ENOTTY]	No previous **blopen** has been issued for the specified file descriptor.
blset		
	[EINVAL]	An illegal value was specified in the structure passed to the system.
	[ENOTTY]	No previous **blopen** has been issued for the specified file descriptor.

WARNINGS

Once **blopen** has been called with a file descriptor and returned successfully, that file descriptor should not subsequently be used as a parameter to the following system calls: *close*(2), *dup*(2), *dup2*(2), *fcntl*(2), *ioctl*(2), *read*(2), or *select*(2) until a **blclose** is called with the same file descriptor as its parameter. Additionally, *scanf*(libc), *fscanf*(libc), *getc*(libc), *getchar*(libc), *fgetc*(libc) and *fgetw*(libc) should not be called for a stream associated with a file descriptor that has been used in a **blopen** call but has not been used in a **blclose** call. These functions call *read*(2) and calling these routines will result in unpredictable behavior.

AUTHOR

Blmode was developed by HP.

SEE ALSO

termio(7).

NAME

bsearch – binary search a sorted table

SYNOPSIS

# include <search.h>

char *bsearch ((char *) key, (char *) base, nel, sizeof (*key), compar)
unsigned nel;
int (*compar)( );

DESCRIPTION

Bsearch is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function. *Key* points to a datum instance to be sought in the table. *Base* points to the element at the base of the table. *Nel* is the number of elements in the table. *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero as accordingly the first argument is to be considered less than, equal to, or greater than the second.

EXAMPLE

The example below searches a table containing pointers to nodes consisting of a string and its length. The table is ordered alphabetically on the string in the node pointed to by each entry.

This code fragment reads in strings and either finds the corresponding node and prints out the string and its length, or prints an error message.

```
#include <stdio.h>

#define TABSIZE        1000

struct node {                   /* these are stored in the table */
        char *string;
        int length;
};
struct node table[TABSIZE];     /* table to be searched */
        .
        .
        .
{
        struct node *node_ptr, node;
        int node_compare( );  /* routine to compare 2 nodes */
        char str_space[20];   /* space to read string into */
        .
        .
        .
        node.string = str_space;
        while (scanf("%s", node.string) != EOF) {
                node_ptr = (struct node *)bsearch((char *)(&node),
                           (char *)table, TABSIZE,
                           sizeof(struct node), node_compare);
                if (node_ptr != NULL) {
                        (void)printf("string = %20s, length = %d\n",
                                node_ptr->string, node_ptr->length);
                } else {
                        (void)printf("not found: %s\n", node.string);
                }
            }
        }
        /* This routine compares two nodes based on an
              alphabetical ordering of the string field.  */
        int
        node_compare(node1, node2)
        struct node *node1, *node2;
        {
            return strcmp(node1->string, node2->string);
        }
```

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

hsearch(3C), lsearch(3C), qsort(3C), tsearch(3C).

DIAGNOSTICS

A NULL pointer is returned if the key cannot be found in the table.

WARNINGS

If more than one entry matches the selection criteria, *bsearch* returns one of them; which one of them is unspecified.

If the table being searched contains two or more entries that match the selection criteria, a random entry is returned by *bsearch* as determined by the search algorithm.

NAME

catread – MPE/RTE-style message catalog support

SYNOPSIS

int catread (fd, set_num, msg_num, msg_buf, buflen [,arg]...)
int fd, set_num, msg_num, buflen;
char *msg_buf, *arg;

DESCRIPTION

Catread is layered on *getmsg*(3C) for supporting message catalog applications from MPE/RTE. Refer to the external specifications for message catalogs on these systems for use of this routine.

The message read from the catalog may have embedded formatting information in the form ![*digit*]. An exclamation mark followed by a digit *n* is replaced by the *n*th argument string. If exclamation marks are not numbered, they are replaced by the arguments in serial order. Either all or none must be numbered.

If successful, returns the number of non-null bytes placed in the buffer.

DIAGNOSTICS

Catread returns a negative integer if *set_num* or *msg_num* are not found in the catalog.

AUTHOR

Catread was developed by the Hewlett-Packard Company.

SEE ALSO

gencat(1), getmsg(3C), hpnls(5).

INTERNATIONAL SUPPORT

8-bit data, messages.

NAME

clock – report CPU time used

SYNOPSIS

long clock ( )

DESCRIPTION

Clock returns the amount of CPU time (in microseconds) used since the first call to *clock*. The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed *wait*(2) or *system*(3S).

The resolution of the clock varies, depending on the hardware and on the software configuration.

WARNINGS

The value returned by *clock* is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution. Because of this, the value returned will wrap around after accumulating only 2147 seconds of CPU time (about 36 minutes).

HARDWARE DEPENDENCIES

Series 200, 300

The clock resolution is 20 milliseconds.

Series 500, 800

The default clock resolution is 10 milliseconds.

SEE ALSO

times(2), wait(2), system(3S).

NAME

toupper, tolower, _toupper, _tolower, toascii – translate characters

SYNOPSIS

```
#include <ctype.h>

int toupper (c)
int c;

int tolower (c)
int c;

int _toupper (c)
int c;

int _tolower (c)
int c;

int toascii (c)
int c;
```

DESCRIPTION

Toupper and *tolower* have as domain the range of *getc*(3S): the integers from –1 through 255. If the argument of *toupper* represents a lower-case letter, the result is the corresponding upper-case letter. If the argument of *tolower* represents an upper-case letter, the result is the corresponding lower-case letter. All other arguments in the domain are returned unchanged.

The macros *_toupper* and *_tolower* accomplish the same thing as *toupper* and *tolower* but have restricted domains and are faster. *_toupper* requires a lower-case letter as its argument; its result is the corresponding upper-case letter. The macro *_tolower* requires an upper-case letter as its argument; its result is the corresponding lower-case letter. Arguments outside the domain cause undefined results. Use of this form will never work with foreign character sets.

Toascii yields its argument with all bits turned off that are not part of a standard 7 bit ASCII character; it is intended for compatibility with other systems.

SEE ALSO

ctype(3C), getc(3S).

NAME

crt0.o, mcrt0.o, frt0.o, mfrt0.o - execution startup routines

DESCRIPTION

The C and Pascal compilers link in either *crt0.o* or *mcrt0.o* to provide startup capabilities and environment for program execution. The only difference between the two is that the latter provides additional functionality for profiling. Similarly, the Fortran compiler will link in either *frt0.o* or *mfrt0.o*.

The following are defined in these routines:

start	Execution start address.
_argc_value	A variable of type *int* containing the number of arguments.
_argv_value	An array of character pointers to the arguments themselves.
_environ	An array of character pointers to the environment in which the program will run. This array is terminated by a null pointer.
float_soft	A variable of type *short* which is zero if the 98635 floating point card is present; non-zero if it is not present.
float_loc	A constant defining the location in memory of the 98635 floating point card.
flag_68881	A variable of type *short* which is non-zero if the 68881 floating point coprocessor is present; zero if it is not present.

In addition, *mcrt0.o* and *mfrt0.o* define **_exit** and **_cntbase** which are only useful to profiling routines.

ORIGIN

AT&T System III

SEE ALSO

cc(1), fc(1), prof(1), pc(1), profil(2), exec(2), monitor(3C).

NAME

crypt, setkey, encrypt – generate hashing encryption

SYNOPSIS

char *crypt (key, salt)
char *key, *salt;

void setkey (key)
char *key;

void encrypt (block, fake)
char *block;
int fake;

DESCRIPTION

Crypt is the password encryption function. It is based on a one way hashing encryption algorithm with variations intended (among other things) to frustrate use of hardware implementations of a key search.

Key is a user's typed password. *Salt* is a two-character string chosen from the set **[a-zA-Z0-9./]**; this string is used to perturb the hashing algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password. The first two characters are the salt itself.

The *setkey* and *encrypt* entries provide (rather primitive) access to the actual hashing algorithm. The argument of *setkey* is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key which is set into the machine. This is the key that will be used with the hashing algorithm to encrypt the string *block* with the function *encrypt*.

The argument to the *encrypt* entry is a character array of length 64 containing only the characters with numerical value 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the hashing algorithm using the key set by *setkey*. *Fake* is not used and is ignored, but should be present if *lint*(1) is used.

SEE ALSO

login(1), passwd(1), getpass(3C), passwd(4).

BUGS

The return value points to static data that are overwritten by each call.

NAME

ctermid – generate file name for terminal

SYNOPSIS

#include <stdio.h>
char *ctermid (s)
char *s;

DESCRIPTION

Ctermid generates the path name of the controlling terminal for the current process, and stores it in a string.

If *s* is a NULL pointer, the string is stored in an internal static area, the contents of which are overwritten at the next call to *ctermid*, and the address of which is returned. Otherwise, *s* is assumed to point to a character array of at least **L_ctermid** elements; the path name is placed in this array and the value of *s* is returned. The constant **L_ctermid** is defined in the *<stdio.h>* header file.

NOTES

The difference between *ctermid* and *ttyname*(3C) is that *ttyname* must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while *ctermid* returns a string (**/dev/tty**) that will refer to the terminal if used as a file name. Thus *ttyname* is useful only if the process already has at least one file open to a terminal.

SEE ALSO

ttyname(3C).

NAME

ctime, nl_ctime, localtime, gmtime, asctime, nl_asctime, timezone, daylight, tzname, tzset – convert date and time to string

SYNOPSIS

#include <time.h>

char *ctime (clock)
long *clock;

char *nl_ctime (clock, format, langid)
long *clock; char *format; int langid;

struct tm *localtime (clock)
long *clock;

struct tm *gmtime (clock)
long *clock;

char *asctime (tm)
struct tm *tm;

char *nl_asctime (tm, format, langid)
struct tm *tm; char *format; int

extern long timezone;

extern int daylight;

extern char *tzname[2];

void tzset ( )

DESCRIPTION

Ctime converts a long integer, pointed to by *clock*, representing the time in seconds since 00:00:00 GMT, January 1, 1970, and returns a pointer to a 26-character string in the following form. All the fields have constant width.

```
Sun Sep 16 01:03:52 1973\n\0
```

NL_ctime extends the capabilities of *ctime* in two ways. First the *format* specification allows the date and time to be output in a variety of ways. *Format* uses the field descriptors defined in *date*(1). If the format is the null string, the D_T_FMT string defined by *langinfo*(3C) is used. Second *langid* provides month and weekday names (when selected as alphabetic by the *format* string) to be in the user's native language.

Localtime and *gmtime* return pointers to "tm" structures, described below. *Localtime* corrects for the time zone and any summer time zone adjustments (Daylight Savings time in the U.S.A.), according to the TZ string in the user's environment. *Gmtime* converts directly to Greenwich Mean Time (GMT), which is the time the HP-UX System uses.

Asctime converts a time value contained in a "tm" structure (as a whole) to a 26-character string, as shown in the above example, and returns a pointer to the string.

NL_asctime, like *nl_ctime*, allows the date string to be formatted, and month and weekday names to be in the user's native language. However, like *asctime* , it takes "tm" as its argument.

Declarations of all the functions and externals, and the "tm" structure, are in the **<time.h>** header file. The structure declaration is:

```
struct tm {
        int tm_sec;         /* seconds (0 - 59) */
        int tm_min;         /* minutes (0 - 59) */
        int tm_hour;        /* hours (0 - 23) */
        int tm_mday;     /* day of month (1 - 31) */
        int tm_mon;      /* month of year (0 - 11) */
        int tm_year;     /* year - 1900 */
        int tm_wday;     /* day of week (Sunday = 0) */
        int tm_yday;     /* day of year (0 - 365) */
        int tm_isdst;
        long tm_tzadj;
};
```

Tm_isdst is non-zero if a summer time zone adjustment such as Daylight Savings time is in effect. *Tm_tzadj* is the difference between GMT and local time expressed in seconds.

The external **long** variable *timezone* contains the difference, in seconds, between GMT and local standard time (in EST, *timezone* is 5*60*60); the external variable *daylight* is non-zero if and only if you have specified a summer time zone adjustment in your TZ environment variable. *Daylight* and *timezone* are derived only from the TZ variable value and are independent of any time value. *Tm_isdst* and *tm_tzadj* indicate what time zone adjustment is in effect for the value contained in the "tm" structure. The method by which TZ is used for summer time zone adjustment is somewhat complex and is described in *tztab*(4). The values of the external variables *timezone*, *daylight*, and *tzname* are set from the environment variable TZ by the function *tzset*, which can be called directly, or indirectly through the functions *localtime, ctime,* or *nl_ctime.* TZ is set by default when the user logs on, to a value in the local **/etc/profile** file, see *profile*(4).

WARNINGS

The return values point to static data whose content is overwritten by each call.

AUTHOR

Ctime was developed by AT&T and Hewlett-Packard Company.

SEE ALSO

time(2), getenv(3C), langinfo(3C), profile(4), tztab(4), environ(5), hpnls(5).

INTERNATIONAL SUPPORT

8-bit data, messages.

NAME

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii – classify characters

SYNOPSIS

```
#include <ctype.h>

int isalpha (c)
int c;

. . .
```

DESCRIPTION

These macros classify character-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. *Isascii* is defined on all integer values; the rest are defined only where *isascii* is true and on the single non-ASCII value **EOF** (–1 – see *stdio*(3S)).

isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, new-line, vertical tab, or form-feed.
ispunct	*c* is a punctuation character (any printing character except space, digit, letter).
isprint	*c* is a printing character, code 040 (space) through 0176 (tilde).
isgraph	*c* is a printing character, like *isprint* except false for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).
isascii	*c* is an ASCII character, code less than 0200.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

SEE ALSO

stdio(3S), ascii(5).

NAME

curses – CRT screen handling and optimization package

SYNOPSIS

#include <curses.h>

cc [flags] files **–lcurses** [libraries]

DESCRIPTION

These routines give the user a method of updating screens with reasonable optimization. In order to initialize the routines, the routine *initscr()* must be called before any of the other routines that deal with windows and screens are used. The routine *endwin()* should be called before exiting. To get character-at-a-time input without echoing, (most interactive, screen oriented-programs want this) after calling *initscr()* you should call *"nonl(); cbreak(); noecho();"*

The full curses interface permits manipulation of data structures called *windows* which can be thought of as two dimensional arrays of characters representing all or part of a CRT screen. A default window called **stdscr** is supplied, and others can be created with **newwin**. Windows are referred to by variables declared "WINDOW *", the type WINDOW is defined in curses.h to be a C structure. These data structures are manipulated with functions described below, among which the most basic are **move**, and **addch**. (More general versions of these functions are included with names beginning with 'w', allowing you to specify a window. The routines not beginning with 'w' affect **stdscr**.) Then *refresh()* is called, telling the routines to make the users CRT screen look like **stdscr**.

Mini-Curses is a subset of curses which does not allow manipulation of more than one window. To invoke this subset, use -DMINICURSES as a **cc** option. This level is smaller and faster than full curses.

If the environment variable TERMINFO is defined, any program using curses will check for a local terminal definition before checking in the standard place. For example, if the standard place is **/usr/lib/terminfo**, and TERM is set to "vt100", then normally the compiled file is found in **/usr/lib/terminfo/v/vt100**. (The "v" is copied from the first letter of "vt100" to avoid creation of huge directories.) However, if TERMINFO is set to **/usr/mark/myterms**, curses will first check **/usr/mark/myterms/v/vt100**, and if that fails, will then check **/usr/lib/terminfo/v/vt100**. This is useful for developing experimental definitions or when write permission in **/usr/lib/terminfo** is not available.

SEE ALSO

terminfo(4).

FUNCTIONS

Routines listed here may be called when using the full curses. Those marked with an asterisk may be called when using Mini-Curses.

addch(ch)*	add a character to *stdscr* (like putchar) (wraps to next line at end of line)
addstr(str)*	calls addch with each character in *str*
attroff(attrs)*	turn off attributes named
attron(attrs)*	turn on attributes named
attrset(attrs)*	set current attributes to *attrs*
baudrate()*	current terminal speed
beep()*	sound beep on terminal
box(win, vert, hor)	draw a box around edges of *win* *vert* and *hor* are chars to use for vert. and hor. edges of box
clear()	clear *stdscr*
clearok(win, bf)	clear screen before next redraw of *win*

clrtobot( )	clear to bottom of *stdscr*
clrtoeol( )	clear to end of line on *stdscr*
cbreak( )*	set cbreak mode
delay_output(ms)*	insert ms millisecond pause in output
delch( )	delete a character
deleteln( )	delete a line
delwin(win)	delete *win*
doupdate( )	update screen from all wnooutrefresh
echo( )*	set echo mode
endwin( )*	end window modes
erase( )	erase *stdscr*
erasechar( )	return user's erase character
fixterm( )	restore tty to "in curses" state
flash( )	flash screen or beep
flushinp( )*	throw away any typeahead
getch( )*	get a char from tty
getstr(str)	get a string through *stdscr*
gettmode( )	establish current tty modes
getyx(win, y, x)	get (y, x) co-ordinates
has_ic( )	true if terminal can do insert character
has_il( )	true if terminal can do insert line
idlok(win, bf)*	use terminal's insert/delete line if bf != 0
inch( )	get char at current (y, x) co-ordinates
initscr( )*	initialize screens
insch(c)	insert a char
insertln( )	insert a line
intrflush(win, bf)	interrupts flush output if bf is TRUE
keypad(win, bf)	enable keypad input
killchar( )	return current user's kill character
leaveok(win, flag)	OK to leave cursor anywhere after refresh if flag!=0 for *win*, otherwise cursor must be left at current position.
longname( )	return verbose name of terminal
meta(win, flag)*	allow meta characters on input if flag != 0
move(y, x)*	move to (y, x) on *stdscr*
mvaddch(y, x, ch)	move(y, x) then addch(ch)
mvaddstr(y, x, str)	similar...
mvcur(oldrow, oldcol, newrow, newcol)	low level cursor motion
mvdelch(y, x)	like delch, but move(y, x) first
mvgetch(y, x)	etc.
mvgetstr(y, x)	
mvinch(y, x)	
mvinsch(y, x, c)	
mvprintw(y, x, fmt, args)	
mvscanw(y, x, fmt, args)	
mvwaddch(win, y, x, ch)	
mvwaddstr(win, y, x, str)	
mvwdelch(win, y, x)	
mvwgetch(win, y, x)	
mvwgetstr(win, y, x)	
mvwin(win, by, bx)	
mvwinch(win, y, x)	

mvwinsch(win, y, x, c)	
mvwprintw(win, y, x, fmt, args)	
mvwscanw(win, y, x, fmt, args)	
newpad(nlines, ncols)	create a new pad with given dimensions
newterm(type, outfd, infd)	set up new terminal of given type to output on outfd, using input (it needed) from infd
newwin(lines, cols, begin_y, begin_x)	create a new window
nl( )*	set newline mapping
nocbreak( )*	unset cbreak mode
nodelay(win, bf)	enable nodelay input mode through getch
noecho( )*	unset echo mode
nonl( )*	unset newline mapping
noraw( )*	unset raw mode
overlay(win1, win2)	overlay win1 on win2
overwrite(win1, win2)	overwrite win1 on top of win2
pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)	like prefresh but with no output until doupdate called
prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)	refresh from pad starting with given upper left corner of pad with output to given portion of screen
printw(fmt, arg1, arg2, ...)	printf on *stdscr*
raw( )*	set raw mode
refresh( )*	make current screen look like *stdscr*
resetterm( )*	set tty modes to "out of curses" state
resetty( )*	reset tty flags to stored value
saveterm( )*	save current modes as "in curses" state
savetty( )*	store current tty flags
scanw(fmt, arg1, arg2, ...)	scanf through *stdscr*
scroll(win)	scroll *win* one line
scrollok(win, flag)	allow terminal to scroll if flag != 0
set_term(new)	now talk to terminal new
setscrreg(t, b)	set user scrolling region to lines t through b
setterm(type)	establish terminal with given type
setupterm(term, filenum, errret)	
standend( )*	clear standout mode attribute
standout( )*	set standout mode attribute
subwin(win, lines, cols, begin_y, begin_x)	create a subwindow
touchwin(win)	change all of *win*
traceoff( )	turn off debugging trace output
traceon( )	turn on debugging trace output
typeahead(fd)	use file descriptor fd to check typeahead
unctrl(ch)*	printable version of *ch*
waddch(win, ch)	add char to *win*
waddstr(win, str)	add string to *win*
wattroff(win, attrs)	turn off *attrs* in *win*
wattron(win, attrs)	turn on *attrs* in *win*

wattrset(win, attrs)	set attrs in *win* to *attrs*
wclear(win)	clear *win*
wclrtobot(win)	clear to bottom of *win*
wclrtoeol(win)	clear to end of line on *win*
wdelch(win, c)	delete char from *win*
wdeleteln(win)	delete line from *win*
werase(win)	erase *win*
wgetch(win)	get a char through *win*
wgetstr(win, str)	get a string through *win*
winch(win)	get char at current (y, x) in *win*
winsch(win, c)	insert char into *win*
winsertln(win)	insert line into *win*
wmove(win, y, x)	set current (y, x) co-ordinates on *win*
wnoutrefresh(win)	refresh but no screen output
wprintw(win, fmt, arg1, arg2, ...)	printf on *win*
wrefresh(win)	make screen look like *win*
wscanw(win, fmt, arg1, arg2, ...)	scanf through *win*
wsetscrreg(win, t, b)	set scrolling region of *win*
wstandend(win)	clear standout attribute in *win*
wstandout(win)	set standout attribute in *win*

TERMINFO LEVEL ROUTINES

These routines should be called by programs wishing to deal directly with the terminfo database. Due to the low level of this interface, it is discouraged. Initially, *setupterm* should be called. This will define the set of terminal dependent variables defined in terminfo(4). The include files <curses.h> and <term.h> should be included to get the definitions for these strings, numbers, and flags. Parameterized strings should be passed through *tparm* to instantiate them. All terminfo strings (including the output of tparm) should be printed with *tputs* or *putp* . Before exiting, *resetterm* should be called to restore the tty modes. (Programs desiring shell escapes or suspending with control Z can call *resetterm* before the shell is called and *fixterm* after returning from the shell.)

fixterm( )	restore tty modes for terminfo use (called by setupterm)
resetterm( )	reset tty modes to state before program entry
setupterm(term, fd, rc)	read in database. Terminal type is the character string *term*, all output is to HP-UX System file descriptor *fd*. A status value is returned in the integer pointed to by *rc*: 1 is normal. The simplest call would be *setupterm(0, 1, 0)* which uses all defaults.
tparm(str, p1, p2, ..., p9)	instantiate string str with parms p_i.
tputs(str, affcnt, putc)	apply padding info to string *str*. *affcnt* is the number of lines affected, or 1 if not applicable. *Putc* is a putchar-like function to which the characters are passed, one at a time.

putp(str)	handy function that calls tputs (str, 1, putchar)
vidputs(attrs, putc)	output the string to put terminal in video attribute mode *attrs*, which is any combination of the attributes listed below. Chars are passed to putchar-like function *putc*.
vidattr(attrs)	Like vidputs but outputs through putchar

TERMCAP COMPATIBILITY ROUTINES

These routines were included as a conversion aid for programs that use termcap. Their parameters are the same as for termcap. They are emulated using the *terminfo* database. They may go away at a later date.

tgetent(bp, name)	look up termcap entry for name
tgetflag(id)	get boolean entry for id
tgetnum(id)	get numeric entry for id
tgetstr(id, area)	get string entry for id
tgoto(cap, col, row)	apply parms to given cap
tputs(cap, affcnt, fn)	apply padding to cap calling fn as putchar

ATTRIBUTES

The following video attributes can be passed to the functions *attron*,*attroff*,*attrset*.

A_STANDOUT	Terminal's best highlighting mode
A_UNDERLINE	Underlining
A_REVERSE	Reverse video
A_BLINK	Blinking
A_DIM	Half bright
A_BOLD	Extra bright or bold
A_BLANK	Blanking (invisible)
A_PROTECT	Protected
A_ALTCHARSET	Alternate character set

FUNCTION KEYS

The following function keys might be returned by *getch* if *keypad* has been enabled. Note that not all of these are currently supported, due to lack of definitions in *terminfo* or the terminal not transmitting a unique code when the key is pressed.

Name	*Value*	*Key name*
KEY_BREAK	0401	break key (unreliable)
KEY_DOWN	0402	The four arrow keys ...
KEY_UP	0403	
KEY_LEFT	0404	
KEY_RIGHT	0405	...
KEY_HOME	0406	Home key (upward+left arrow)
KEY_BACKSPACE	0407	backspace (unreliable)
KEY_F0	0410	Function keys. Space for 64 is reserved.
KEY_F(n)	(KEY_F0+(n))	Formula for fn.
KEY_DL	0510	Delete line
KEY_IL	0511	Insert line
KEY_DC	0512	Delete character
KEY_IC	0513	Insert char or enter insert mode
KEY_EIC	0514	Exit insert char mode
KEY_CLEAR	0515	Clear screen

KEY_EOS	0516	Clear to end of screen
KEY_EOL	0517	Clear to end of line
KEY_SF	0520	Scroll 1 line forward
KEY_SR	0521	Scroll 1 line backwards (reverse)
KEY_NPAGE	0522	Next page
KEY_PPAGE	0523	Previous page
KEY_STAB	0524	Set tab
KEY_CTAB	0525	Clear tab
KEY_CATAB	0526	Clear all tabs
KEY_ENTER	0527	Enter or send (unreliable)
KEY_SRESET	0530	soft (partial) reset (unreliable)
KEY_RESET	0531	reset or hard reset (unreliable)
KEY_PRINT	0532	print or copy
KEY_LL	0533	home down or bottom (lower left)

WARNINGS

The plotting library *plot*(3X) and the curses library *curses*(3X) both use the names erase( ) and move( ). The curses versions are macros. If you need both libraries, put the *plot*(3X) code in a different source file than the *curses*(3X) code, and/or #undef move( ) and erase( ) in the *plot*(3X) code.

WARNINGS

HP only supports terminals listed on the current list of supported devices. However, non-supported and supported terminals can be in the terminfo database. If you use such non-supported terminals, they may not work correctly.

NAME

cuserid – get character login name of the user

SYNOPSIS

#include <stdio.h>

char *cuserid (s)
char *s;

DESCRIPTION

Cuserid generates a character-string representation of the login name that the owner of the current process is logged in under. If *s* is a NULL pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, *s* is assumed to point to an array of at least **L_cuserid** characters; the representation is left in this array. The constant **L_cuserid** is defined in the **<stdio.h>** header file.

DIAGNOSTICS

If the login name cannot be found, *cuserid* returns a NULL pointer; if *s* is not a NULL pointer, a null character (**\0**) will be placed at *s*[0].

BUGS

Cuserid uses *getpwnam* on *getpwent*(3C); thus the results of a user's call to the latter will be obliterated by a subsequent call to the former.

SEE ALSO

getlogin(3C), getpwent(3C).

NAME

cvtnum – convert string to floating point number

SYNOPSIS

```
#include <cvtnum.h>

int cvtnum(src,dst,typ,rnd,ptr,inx)
unsigned char *src,*dst,**ptr;
int typ,rnd,*inx;
```

DESCRIPTION

The function *cvtnum* converts an ASCII character string to a number in one of four floating point formats: single precision, double precision, extended precision, or packed decimal string.

The string pointed to by *src* is the string representation of a standard number, an infinity, or a not-a-number. A standard number begins with an optional sign followed by a string of digits optionally containing a decimal point. It may then have an optional **e** or **E** followed by an optional sign followed by an integer. Infinities are represented by **INF** preceded by an optional sign. The string for a not-a-number is an optional sign followed by **NaN** followed by any number of hexadecimal digits enclosed in parentheses.

The result is moved to *dst* and will be of the size and format as defined for the 68881 floating-point coprocessor.

typ indicates the type of conversion to be done. It may be one of four values: **C_SNGL**, **C_DBLE**, **C_EXT**, or **C_DPACK** indicating single precision, double precision, extended precision and packed decimal string respectively.

rnd specifies the type of rounding mode and may be one of four values: **C_NEAR**, **C_POS_INF**, **C_NEG_INF**, or **C_TOZERO** indicating round to nearest, to positive infinity, to negative infinity and to zero respectively.

If the value of **ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by ptr. If no number can be formed, **ptr* is set to *str* .

If *inx* is not (int *)NULL, *cvtnum* will use this to return an indication of the inexactness of the conversion. A zero indicates exact; a non-zero value, inexact.

SEE ALSO

scanf(3S), strtod(3C), strtol(3C)
MC68881 Floating-Point Coprocessor User's Manual

DIAGNOSTICS

If no errors occur or no non-standard conversions are done, *cvtnum* returns 0. Otherwise, it will return one of the following:

C_BADCHAR - Illegal character or unexpected end of string
C_OVER - Overflow
C_UNDER - Underflow
C_INF - Infinity
C_QNAN - Quiet NaN
C_SNAN - Signalling NaN

NAME

datalock – lock process into memory, after allocating data and stack space

SYNOPSIS

#include <sys/lock.h>
int datalock (datsiz, stsiz);
int datsiz, stsiz;

DESCRIPTION

Datalock allocates at least *datsiz* bytes of data space and *stsiz* bytes of stack space, then locks the program in memory. The data space is allocated with either *malloc*(3C) or *malloc*(3X) (whichever is linked with the program). After the program is locked, this space is released with *free* (on *malloc*(3C)) or *free* (on *malloc*(3X)), making it available for use. This allows the calling program to use that much space dynamically without receiving the *SIGSEGV* signal.

The effective user ID of the calling process must be super-user or be a member of or have an effective group ID of a group having PRIV_MLOCK access to use this call (see **getprivgrp(2)**).

EXAMPLES

The following call to *datalock* allocates 4096 bytes of data space and 2048 bytes of stack space and then locks the process in memory:

```
datalock (4096, 2048);
```

RETURN VALUE

Returns –1 if *malloc* cannot allocate enough memory or *plock*(2) returned an error.

AUTHOR

Datalock was developed by the Hewlett-Packard Company.

SEE ALSO

getprivgrp(2), plock(2).

BUGS

Multiple datalock's may not be the same as one big one.

Methods for calculating the required size are not yet well developed.

NAME

dial, undial – establish an out-going terminal line connection

SYNOPSIS

#include <dial.h>

int dial (call)
CALL call;

void undial (fd)
int fd;

DESCRIPTION

Dial returns a file-descriptor for a terminal line open for read/write. The argument to *dial* is a CALL structure (defined in the <*dial.h*> header file).

When finished with the terminal line, the calling program must invoke *undial* to release the semaphore that has been set during the allocation of the terminal device.

The definition of CALL in the <*dial.h*> header file is:

```
typedef struct {
        struct termio   *attr;       /* pointer to termio attribute struct */
        int             baud;        /* transmission data rate */
        int             speed;       /* 212A modem: low=300, high=1200 */
        char            *line;       /* device name for out-going line */
        char            *telno;      /* pointer to tel-no digits string */
        int             modem;       /* specify modem control for direct lines */
        char            *device;     /*Will hold the name of the device used
                                       to make a connection */
        int             dev_len;     /* The length of the device used to
                                       make connection */
} CALL;
```

The CALL element *speed* is intended only for use with an outgoing dialed call, in which case its value should be either 300 or 1200 to identify the 113A modem, or the high- or low-speed setting on the 212A modem. Note that the 113A modem or the low-speed setting of the 212A modem will transmit at any rate between 0 and 300 bits per second. However, the high-speed setting of the 212A modem transmits and receivers at 1200 bits per second only. The CALL element *baud* is for the desired transmission baud rate. For example, one might set *baud* to 110 and *speed* to 300 (or 1200). However, if *speed* set to 1200 *baud* must be set to high (1200).

If the desired terminal line is a direct line, a string pointer to its device-name should be placed in the *line* element in the CALL structure. Legal values for such terminal device names are kept in the *L-devices* file. In this case, the value of the *baud* element need not be specified as it will be determined from the *L-devices* file.

The *telno* element is for a pointer to a character string representing the telephone number to be dialed. Such numbers may consist only of symbols described below. The termination symbol will be supplied by the *dial* function, and should not be included in the *telno* string passed to *dial* in the CALL structure.

Permissible codes	
0-9	dial 0-9
* or :	dial *
# or ;	dial #
-	4-second delay for second dial tone
e or <	end-of-number
w or =	wait for secondary dial tone

f	flash off hook for 1 second

The CALL element *modem* is used to specify modem control for direct lines. This element should be non-zero if modem control is required. The CALL element *attr* is a pointer to a *termio* structure, as defined in the *termio.h* header file. A NULL value for this pointer element may be passed to the *dial* function, but if such a structure is included, the elements specified in it will be set for the outgoing terminal line before the connection is established. This is often important for certain attributes such as parity and baud-rate.

The CALL element *device* is used to hold the device name (cul..) that establishes the connection.

The CALL element *dev_len* is the length of the device name that is copied into the array device.

FILES

/usr/lib/uucp/L-devices
/usr/spool/uucp/LCK..*tty-device*

SEE ALSO

uucp(1), alarm(2), read(2), write(2), termio(7).

DIAGNOSTICS

On failure, a negative value indicating the reason for the failure will be returned. Mnemonics for these negative indices as listed here are defined in the <*dial.h*> header file.

INTRPT	-1	/* interrupt occurred */
D_HUNG	-2	/* dialer hung (no return from write) */
NO_ANS	-3	/* no answer within 10 seconds */
ILL_BD	-4	/* illegal baud-rate */
A_PROB	-5	/* automatic call unit (acu) problem (open() failure) */
L_PROB	-6	/* line problem (open() failure) */
NO_Ldv	-7	/* can't open LDEVS file */
DV_NT_A	-8	/* requested device not available */
DV_NT_K	-9	/* requested device not known */
NO_BD_A	-10	/* no device available at requested baud */
NO_BD_K	-11	/* no device known at requested baud */

WARNINGS

Including the <*dial.h*> header file automatically includes the <*termio.h*> header file.

The above routine uses <*stdio.h*>, which causes unexpected increases in the size of programs, not otherwise using standard I/O.

BUGS

An *alarm*(2) system call for 3600 seconds is made (and caught) within the *dial* module for the purpose of "touching" the *LCK..* file and constitutes the device allocation semaphore for the terminal device. Otherwise, *uucp*(1) may simply delete the *LCK..* entry on its 90-minute clean-up rounds. The alarm may go off while the user program is in a *read*(2) or *write*(2) system call, causing an apparent error return. If the user program expects to be around for an hour or more, error returns from reads should be checked for **(errno==EINTR)**, and the read possibly reissued.

NAME

opendir, readdir, telldir, seekdir, rewinddir, closedir – directory operations

SYNOPSIS

```
#include <ndir.h>

DIR *opendir(filename)
char *filename;

struct direct *readdir(dirp)
DIR *dirp;

long telldir(dirp)
DIR *dirp;

seekdir(dirp, loc)
DIR *dirp;
long loc;

rewinddir(dirp)
DIR *dirp;

closedir(dirp)
DIR *dirp;
```

DESCRIPTION

The purpose of this library package is to provide functions which allow programs to read directory entries without having to know the actual directory format associated with the file system. This allows programs to be ported from one file system to another. Therefore, this is the recommended way to read directory entries.

Opendir opens the directory named by *filename* and associates a *directory stream* with it. *Opendir* returns a pointer to be used to identify the *directory stream* in subsequent operations. The pointer **NULL** is returned if *filename* cannot be accessed, if *filename* is not a directory, or if sufficient memory cannot be allocated for a buffer of size DIRBLKSIZ blocks (see HARDWARE DEPENDENCIES).

Readdir returns a pointer to the next directory entry. It returns **NULL** upon reaching the end of the directory or detecting an invalid *seekdir* operation.

Telldir returns the current location, in bytes, associated with the named *directory stream.*

Seekdir sets the position of the next *readdir* operation on the *directory stream. Loc* is a byte offset within the directory file. The new position reverts to the one associated with the *directory stream* when the *telldir* operation was performed. Values returned by *telldir* are good only for the lifetime of the DIR pointer from which they are derived. If the directory is closed and then re-opened, the *telldir* value may be invalidated due to undetected directory compaction. It is safe to use a previous *telldir* value immediately after a call to *opendir* and before any calls to *readdir*.

Rewinddir resets the position of the named *directory stream* to the beginning of the directory.

Closedir causes the named *directory stream* to be closed, and the structure associated with the DIR pointer to be freed.

See */usr/include/ndir.h* for a description of the fields available in a directory entry. The preferred way to search the current directory for entry "name" is:

```
        len = strlen(name);
        dirp = opendir(".");
        for (dp = readdir(dirp); dp != NULL; dp = readdir(dirp)) {
                if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
                        closedir(dirp);
                        return FOUND;
                }
        }
        closedir(dirp);
        return NOT_FOUND;
```

HARDWARE DEPENDENCIES

Series 200, 300, 800:
Malloc(3) is used to allocate memory.

Series 500:
Memallc(2) is used to allocate memory.

AUTHOR

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

FILES

/usr/include/ndir.h

SEE ALSO

close(2), lseek(2), open(2), read(2).

NAME

drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48 – generate uniformly distributed pseudo-random numbers

SYNOPSIS

double drand48 ( )

double erand48 (xsubi)
unsigned short xsubi[3];

long lrand48 ( )

long nrand48 (xsubi)
unsigned short xsubi[3];

long mrand48 ( )

long jrand48 (xsubi)
unsigned short xsubi[3];

void srand48 (seedval)
long seedval;

unsigned short *seed48 (seed16v)
unsigned short seed16v[3];

void lcong48 (param)
unsigned short param[7];

DESCRIPTION

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

Functions *drand48* and *erand48* return non-negative double-precision floating-point values uniformly distributed over the interval [0.0, 1.0).

Functions *lrand48* and *nrand48* return non-negative long integers uniformly distributed over the interval $[0, 2^{31})$.

Functions *mrand48* and *jrand48* return signed long integers uniformly distributed over the interval $[-2^{31}, 2^{31})$.

Functions *srand48, seed48* and *lcong48* are initialization entry points, one of which should be invoked before either *drand48, lrand48* or *mrand48* is called. (Although it is not recommended practice, constant default initializer values will be supplied automatically if *drand48, lrand48* or *mrand48* is called without a prior call to an initialization entry point.) Functions *erand48, nrand48* and *jrand48* do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, X_i, according to the linear congruential formula

$$X_{n+1} = (aX_n + c)_{\mathrm{mod}\ m} \qquad n \geq 0.$$

The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless *lcong48* has been invoked, the multiplier value a and the addend value c are given by

$$a = \mathrm{5DEECE66D}_{16} = 273673163155_8$$
$$c = \mathrm{B}_{16} = 13_8.$$

The value returned by any of the functions *drand48, erand48, lrand48, nrand48, mrand48* or *jrand48* is computed by first generating the next 48-bit X_i in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of X_i and transformed into the returned value.

The functions ***drand48***, ***lrand48*** and ***mrand48*** store the last 48-bit X_i generated in an internal buffer; that is why they must be initialized prior to being invoked. The functions ***erand48***, ***nrand48*** and ***jrand48*** require the calling program to provide storage for the successive X_i values in the array specified as an argument when the functions are invoked. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, functions ***erand48***, ***nrand48*** and ***jrand48*** allow separate modules of a large program to generate several ***independent*** streams of pseudo-random numbers, i.e., the sequence of numbers in each stream will ***not*** depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function ***srand48*** sets the high-order 32 bits of X_i to the 32 bits contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value $330E_{16}$.

The initializer function ***seed48*** sets the value of X_i to the 48-bit value specified in the argument array. In addition, the previous value of X_i is copied into a 48-bit internal buffer, used only by ***seed48***, and a pointer to this buffer is the value returned by ***seed48***. This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time — use the pointer to get at and store the last X_i value, and then use this value to reinitialize via ***seed48*** when the program is restarted.

The initialization function ***lcong48*** allows the user to specify the initial X_i, the multiplier value a, and the addend value c. Argument array elements ***param[0-2]*** specify X_i, ***param[3-5]*** specify the multiplier a, and ***param[6]*** specifies the 16-bit addend c. After ***lcong48*** has been called, a subsequent call to either ***srand48*** or ***seed48*** will restore the "standard" multiplier and addend values, a and c, specified on the previous page.

SEE ALSO

rand(3C).

NAME

ecvt, fcvt, gcvt, nl_gcvt – convert floating-point number to string

SYNOPSIS

```
char *ecvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt (value, ndigit, buf)
double value;
int ndigit;
char *buf;

char *nl_gcvt (value, ndigit, buf, langid)
double value;
int ndigit;
char *buf;
int langid;
```

DESCRIPTION

Ecvt converts *value* to a null-terminated string of *ndigit* digits and returns a pointer thereto. The high-order digit is non-zero, unless the value is zero. The low-order digit is rounded. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The decimal point is not included in the returned string. If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero.

Fcvt is identical to *ecvt*, except that the correct digit has been rounded for printf "%f" (FORTRAN F-format) output of the number of digits specified by *ndigit*.

Gcvt converts the *value* to a null-terminated string in the array pointed to by *buf* and returns *buf*. It attempts to produce *ndigit* significant digits in FORTRAN F-format if possible, otherwise E-format, ready for printing. A minus sign, if there is one, or a decimal point will be included as part of the returned string. Trailing zeros are suppressed.

NL_gcvt differs from *gcvt* only in that it uses *langid* to determine what the radix character should be (e.g., '.' or ','). If *langid* is not valid, or information for *langid* has not been installed, the radix character defaults to a period.

SEE ALSO

printf(3S), hpnls(5), langid(5).

BUGS

The values returned by *ecvt* and *fcvt* point to a single static data array whose content is overwritten by each call.

INTERNATIONAL SUPPORT

8-bit data, messages.

NAME

end, etext, edata – last locations in program

SYNOPSIS

extern end;
extern etext;
extern edata;

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. The address of *etext* is the first address above the program text, *edata* above the initialized data region, and *end* above the uninitialized data region. Note that the definition of each of these is implementation-dependent. See *HARDWARE DEPENDENCIES* below.

When execution begins, the program break (the first location beyond the data) coincides with *end*, but the program break may be reset by the routines of *brk*(2), *malloc*(3C), standard input/output (*stdio*(3S)), the profile (**–p**) option of *cc*(1), and so on. Thus, the current value of the program break should be determined by **sbrk(0)** (see *brk*(2)).

HARDWARE DEPENDENCIES

Series 500:

End is the lowest heap address available to the user. *Etext* and *edata* are not supported.

Memallc(2) is more efficient than *malloc*(3C) for setting the program break.

In C, these names must look like addresses. Thus, you would write **&end** instead of **end** to access the current value of *end*.

SEE ALSO

cc(1), brk(2), malloc(3C), stdio(3S).

NAME

erf, erfc – error function and complementary error function

SYNOPSIS

#include <math.h>

double erf (x)
double x;

double erfc (x)
double x;

DESCRIPTION

Erf returns the error function of x, defined as $\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$.

Erfc, which returns 1.0 – *erf*(x), is provided because of the extreme loss of relative accuracy if *erf*(x) is called for large x and the result subtracted from 1.0 (e.g., for $x = 5$, 12 places are lost).

SEE ALSO

exp(3M).

NAME

exp, log, log10, pow, sqrt – exponential, logarithm, power, square root functions

SYNOPSIS

#include <math.h>

double exp (x)
double x;

double log (x)
double x;

double log10 (x)
double x;

double pow (x, y)
double x, y;

double sqrt (x)
double x;

DESCRIPTION

Exp returns e^x.

Log returns the natural logarithm of x. The value of x must be positive.

Log10 returns the logarithm base ten of x. The value of x must be positive.

Pow returns x^y. If x is zero, y must be positive. If x is negative, y must be an integer.

Sqrt returns the non-negative square root of x. The value of x may not be negative.

HARDWARE DEPENDENCIES

Series 200, 300, 500:
The algorithms used are those from HP 9000 BASIC.

ERRORS

Exp returns **HUGE** when the correct value would overflow, or 0 when the correct value would underflow, and sets *errno* to **ERANGE**.

Log and *log10* return **–HUGE** and set *errno* to **EDOM** when x is non-positive. A message indicating DOMAIN error (or SING error when x is 0) is printed on the standard error output.

Pow returns 0 and sets *errno* to **EDOM** when x is 0 and y is non-positive, or when x is negative and y is not an integer. In these cases a message indicating DOMAIN error is printed on the standard error output. When the correct value for *pow* would overflow or underflow, *pow* returns **±HUGE** or 0 respectively, and sets *errno* to **ERANGE**.

Sqrt returns 0 and sets *errno* to **EDOM** when x is negative. A message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

hypot(3M), matherr(3M), sinh(3M).

NAME

fclose, fflush – close or flush a stream

SYNOPSIS

#include <stdio.h>

int fclose (stream)
FILE *stream;

int fflush (stream)
FILE *stream;

DESCRIPTION

Fclose causes any buffered data for the named *stream* to be written out, and the *stream* to be closed. Buffers allocated by the standard input/output system may be freed.

Fclose is performed automatically for all open files upon calling *exit*(2).

Fflush causes any buffered data for the named *stream* to be written to that file. The *stream* remains open.

DIAGNOSTICS

These functions return 0 for success, and **EOF** if any error (such as trying to write to a file that has not been opened for writing) was detected.

SEE ALSO

close(2), exit(2), fopen(3S), setbuf(3S).

NAME

ferror, feof, clearerr, fileno – stream status inquiries

SYNOPSIS

#include <stdio.h>

int ferror (stream)
FILE
***stream;**

int feof (stream)
FILE
***stream;**

void clearerr (stream)
FILE
***stream;**

int fileno (stream)
FILE
***stream;**

DESCRIPTION

Ferror returns non-zero when an I/O error has previously occurred reading from or writing to the named *stream*, otherwise zero. Unless cleared by *clearerr*, or unless the specific *stdio* routine so indicates, the error indication lasts until the stream is closed.

Feof returns non-zero when **EOF** has previously been detected reading the named input *stream*, otherwise zero.

Clearerr resets the error indicator and **EOF** indicator to zero on the named *stream*.

Fileno returns the integer file descriptor associated with the named *stream*; see *open*(2).

NOTE

All these functions are implemented as macros; they cannot be declared or redeclared.

SEE ALSO

open(2), fopen(3S).

NAME

floor, ceil, fmod, fabs – floor, ceiling, remainder, absolute value functions

SYNOPSIS

#include <math.h>

double floor (x)
double x;

double ceil (x)
double x;

double fmod (x, y)
double x, y;

double fabs (x)
double x;

DESCRIPTION

Floor returns the largest integer (as a double-precision number) not greater than x.

Ceil returns the smallest integer not less than x.

Fmod returns the floating-point remainder of the division of x by y: zero if y is zero or if x/y would overflow; otherwise the number f with the same sign as x, such that $x = iy + f$ for some integer i, and $|f| < |y|$.

Fabs returns the absolute value of x, $|x|$.

SEE ALSO

abs(3C).

NAME

fopen, freopen, fdopen – open or re-open a stream file; convert file to stream

SYNOPSIS

#include <stdio.h>

FILE *fopen (file_name, type)
char *file_name, *type;

FILE *freopen (file_name, type, stream)
char *file_name, *type;
FILE *stream;

FILE *fdopen (fildes, type)
int fildes;
char *type;

DESCRIPTION

Fopen opens the file named by *file_name* and associates a *stream* with it. *Fopen* returns a pointer to the FILE structure associated with the *stream*.

File_name points to a character string that contains the name of the file to be opened.

Type is a character string having one of the following values:

"r"	open for reading
"w"	truncate or create for writing
"a"	append; open for writing at end of file, or create for writing
"r+"	open for update (reading and writing)
"w+"	truncate or create for update
"a+"	append; open or create for update at end-of-file

Freopen substitutes the named file in place of the open *stream*. The original *stream* is closed, regardless of whether the open ultimately succeeds. *Freopen* returns a pointer to the FILE structure associated with *stream*.

Freopen is typically used to attach the preopened *streams* associated with **stdin**, **stdout** and **stderr** to other files.

Fdopen associates a *stream* with a file descriptor. File descriptors are obtained from *open*, *dup*, *creat*, or *pipe*(2), which open files but do not return pointers to a FILE structure *stream*. Streams are necessary input for many of the Section 3S library routines. The *type* of *stream* must agree with the mode of the open file.

When a file is opened for update, both input and output may be done on the resulting *stream*. However, output may not be directly followed by input without an intervening *fseek* or *rewind*, and input may not be directly followed by output without an intervening *fseek*, *rewind*, or an input operation which encounters end-of-file.

When a file is opened for append (i.e., when *type* is "a" or "a+"), it is impossible to overwrite information already in the file. *Fseek* may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

SEE ALSO

creat(2), dup(2), open(2), pipe(2), fclose(3S), fseek(3S), popen(3S).

DIAGNOSTICS

Fopen and *freopen* return a NULL pointer if *file_name* cannot be accessed, if there are too many open files, or if the arguments are incorrect.

Fdopen returns a **NULL** if there are too many open files, or if the arguments are ill-formed.

NAME

fread, fwrite – buffered binary input/output to a stream file

SYNOPSIS

#include <stdio.h>

int fread (ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;

int fwrite (ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;

DESCRIPTION

Fread copies, into an array pointed to by *ptr*, *nitems* items of data from the named input *stream*, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. *Fread* stops appending bytes if an end-of-file or error condition is encountered while reading *stream*, or if *nitems* items have been read. *Fread* leaves the file pointer in *stream*, if defined, pointing to the byte following the last byte read if there is one. *Fread* does not change the contents of *stream*.

Fwrite appends at most *nitems* items of data from the array pointed to by *ptr* to the named output *stream*. *Fwrite* stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream*. *Fwrite* does not change the contents of the array pointed to by *ptr*.

The argument *size* is typically *sizeof(*ptr)* where the pseudo-function *sizeof* specifies the length of an item pointed to by *ptr*. If *ptr* points to a data type other than *char* it should be cast into a pointer to *char*.

SEE ALSO

read(2), write(2), fopen(3S), getc(3S), gets(3S), printf(3S), putc(3S), puts(3S), scanf(3S).

DIAGNOSTICS

Fread and *fwrite* return the number of items read or written. If *size* or *nitems* is non-positive, no characters are read or written and 0 is returned by both *fread* and *fwrite*.

NAME

frexp, ldexp, modf – split floating-point into mantissa and exponent

SYNOPSIS

```
double frexp (value, eptr)
double value;
int *eptr;

double ldexp (value, exp)
double value;
int exp;

double modf (value, iptr)
double value, *iptr;
```

DESCRIPTION

Every non-zero number can be written uniquely as $x * 2^n$, where the "mantissa" (fraction) x is in the range $0.5 \leq |x| < 1.0$, and the "exponent" n is an integer.

Frexp returns the mantissa of a double *value*, and stores the exponent indirectly in the location pointed to by *eptr*. If *value* is zero, both results returned by *frexp* are zero.

Ldexp returns the quantity $value * 2^{exp}$.

Modf returns the signed fractional part of *value* and stores the integral part indirectly in the location pointed to by *iptr*.

DIAGNOSTICS

If *ldexp* would cause overflow, **±HUGE** is returned (according to the sign of *value*), and *errno* is set to **ERANGE**.
If *ldexp* would cause underflow, zero is returned and *errno* is set to **ERANGE**.

NAME

fseek, rewind, ftell – reposition a file pointer in a stream

SYNOPSIS

```
#include <stdio.h>

int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

void rewind (stream)
FILE *stream;

long ftell (stream)
FILE *stream;
```

DESCRIPTION

Fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, from the current position, or from the end of the file, according as *ptrname* has the value 0, 1, or 2.

Rewind(stream) is equivalent to *fseek* (stream, 0L, 0), except that no value is returned.

Fseek and *rewind* undo any effects of *ungetc*(3S).

After *fseek* or *rewind*, the next operation on a file opened for update may be either input or output. *Rewind* also does an implicit *clearerr* (on *ferror*(3S)) call.

Ftell returns the offset of the current byte relative to the beginning of the file associated with the named *stream*.

SEE ALSO

lseek(2), ferror(3S), fopen(3S), popen(3S), ungetc(3S).

DIAGNOSTICS

Fseek returns non-zero for improper seeks, otherwise zero. An improper seek can be, for example, an *fseek* done on a file that has not been opened via *fopen*; in particular, *fseek* may not be used on a terminal, or on a file opened via *popen*(3S).

Ftell returns -1 for error conditions.

WARNING

Although on HP-UX an offset returned by *ftell* is measured in bytes, and it is permissible to seek to positions relative to that offset, portability to non-UNIX operating systems requires that an offset be used by *fseek* directly. Arithmetic may not meaningfully be performed on such an offset, which is not necessarily measured in bytes.

NAME

ftw – walk a file tree

SYNOPSIS

#include <ftw.h>

```
int ftw (path, fn, depth)
char *path;
int (*fn) ( );
int depth;
```

DESCRIPTION

Ftw recursively descends the directory hierarchy rooted in ***path***. For each object in the hierarchy, *ftw* calls *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a **stat** structure (see *stat*(2)) containing information about the object, and an integer. Possible values of the integer, defined in the <ftw.h> header file, are FTW_F for a file, FTW_D for a directory, FTW_DNR for a directory that cannot be read, and FTW_NS for an object for which *stat* could not successfully be executed. If the integer is FTW_DNR, descendants of that directory will not be processed. If the integer is FTW_NS, the **stat** structure will contain garbage. An example of an object that would cause FTW_NS to be passed to *fn* would be a file in a directory with read but without execute (search) permission.

Ftw visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a nonzero value, or some error is detected within *ftw* (such as an I/O error). If the tree is exhausted, *ftw* returns zero. If *fn* returns a nonzero value, *ftw* stops its tree traversal and returns whatever value was returned by *fn*. If *ftw* detects an error, it returns -1, and sets the error type in ***errno***.

Ftw uses one file descriptor for each level in the tree. The ***depth*** argument limits the number of file descriptors so used. If ***depth*** is zero or negative, the effect is the same as if it were 1. ***Depth*** must not be greater than the number of file descriptors currently available for use. *Ftw* will run more quickly if ***depth*** is at least as large as the number of levels in the tree.

SEE ALSO

stat(2), malloc(3C).

BUGS

Because *ftw* is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.
It could be made to run faster and use less storage on deep structures at the cost of considerable complexity.
Ftw uses *malloc*(3C) to allocate dynamic storage during its operation. If *ftw* is forcibly terminated, such as by *longjmp* being executed by *fn* or an interrupt routine, *ftw* will not have a chance to free that storage, so it will remain permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn* return a nonzero value at its next invocation.

NAME

gamma, signgam – log gamma function

SYNOPSIS

#include <math.h>

double gamma (x)
double x;

extern int signgam;

DESCRIPTION

Gamma returns $\ln(|\Gamma(x)|)$, where $\Gamma(x)$ is defined as $\int_0^\infty e^{-t} t^{x-1} dt$. The sign of $\Gamma(x)$ is returned in the external integer *signgam*. The argument x may not be a non-positive integer. (*Gamma* is defined over the reals excluding the non-positive integers).

The following C program fragment might be used to calculate Γ:

```
if ((y = gamma(x)) > LN_MAXDOUBLE)
        error( );
y = signgam * exp(y);
```

where LN_MAXDOUBLE is the least value that causes *exp*(3M) to return a range error, and is defined in the <*values.h*> header file.

DIAGNOSTICS

For non-positive integer arguments **HUGE** is returned, and *errno* is set to **EDOM**. A message indicating SING error is printed on the standard error output.

If the correct value would overflow, *gamma* returns **HUGE** and sets *errno* to **ERANGE**.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

exp(3M), matherr(3M), values(5).

NAME

getc, getchar, fgetc, getw – get character or word from a stream file

SYNOPSIS

#include <stdio.h>

int getc (stream)
FILE *stream;

int getchar ()

int fgetc (stream)
FILE *stream;

int getw (stream)
FILE *stream;

DESCRIPTION

Getc returns the next character (i.e., byte) from the named input *stream*, as an integer. It also moves the file pointer, if defined, ahead one character in *stream*. *Getchar* is defined as *getc*(stdin). *Getc* and *getchar* are macros.

Fgetc behaves like *getc*, but is a function rather than a macro. *Fgetc* runs more slowly than *getc*, but it takes less space per invocation and its name can be passed as an argument to a function.

Getw returns the next word (i.e. *int* in C) from the named input *stream*. *Getw* increments the associated file pointer, if defined, to point to the next word. The size of a word is the size of an integer and varies from machine to machine. *Getw* assumes no special alignment in the file.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), gets(3S), putc(3S), scanf(3S).

DIAGNOSTICS

These functions return the constant **EOF** at end-of-file or upon an error. Because **EOF** is a valid integer, *ferror*(3S) should be used to detect *getw* errors.

WARNING

If the integer value returned by *getc*, *getchar*, or *fgetc* is stored into a character variable and then compared against the integer constant **EOF**, the comparison may never succeed, because sign-extension of a character on widening to integer is machine-dependent.

BUGS

Because it is implemented as a macro, *getc* treats incorrectly a *stream* argument with side effects. In particular, **getc(*f++)** does not work sensibly. *Fgetc* should be used instead.
Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be read using *getw* on a different processor.

NAME

getcwd – get path-name of current working directory

SYNOPSIS

char *getcwd (buf, size)
char *buf;
int size;

DESCRIPTION

Getcwd returns a pointer to the current directory path-name. The value of *size* must be at least two greater than the length of the path-name to be returned.

If *buf* is a NULL pointer, *getcwd* will obtain *size* bytes of space using *malloc*(3C). In this case, the pointer returned by *getcwd* may be used as the argument in a subsequent call to *free*.

EXAMPLE

```
        char *cwd, *getcwd();
        .
        .
        .
        if ((cwd = getcwd((char *)NULL, 64)) == NULL) {
                perror("pwd");
                exit(1);
}
printf("%s\n", cwd);
```

SEE ALSO

pwd(1), malloc(3C), popen(3S).

DIAGNOSTICS

Returns **NULL** with *errno* set if *size* is not large enough, or if an error ocurrs in a lower-level function.

NAME

getenv – return value for environment name

SYNOPSIS

char *getenv (name)
char *name;

DESCRIPTION

Getenv searches the environment list (see *environ*(5)) for a string of the form *name*=*value*, and returns a pointer to the *value* in the current environment if such a string is present, otherwise a NULL pointer. *Name* may be either the desired name, null-terminated, or of the form *name*=*value*, in which case *getenv* uses the portion to the left of the "=" as the search key.

SEE ALSO

exec(2), putenv(3C), environ(5).

NAME

getfsent, getfsspec, getfsfile, getfstype, setfsent, endfsent – get file system descriptor file entry

SYNOPSIS

```
#include <checklist.h>

struct checklist *getfsent()

struct checklist *getfsspec(spec)
char *spec;

struct checklist *getfsfile(file)
char *file;

struct checklist *getfstype(type)
char *type;

int setfsent()

int endfsent()
```

DESCRIPTION

Getfsent, *getfsspec*, *getfsfile*, and *getfstype* each return a pointer to an object with the following structure containing the broken-out fields of a line in the /etc/checklist file. The structure is declared in the <checklist.h> header file:

```
struct checklist{
     char     *fs_spec;     /* special file name */
     char     *fs_bspec;    /* block special file name */
     char     *fs_dir;      /* file sys directory name */
     char     *fs_type;     /* type: ro, rw, sw, xx */
     int      fs_passno;    /* fsck pass number */
     int      fs_freq;      /* backup frequency */
};
```

The fields have meanings described in *checklist*(4). If the block special file name, the file system directory name, and the type are not all defined on the associated line in /etc/checklist, these routines will return pointers to NULL in the fs_bspec, fs_file and fs_type fields. If the pass number or the backup frequency field are not present on the line, these routines will return -1 in the corresponding structure member. Fs_freq is reserved for future use.

Getfsent reads the next line of the file, opening the file if necessary.

Setfsent opens and rewinds the file.

Endfsent closes the file.

Getfsspec and *getfsfile* sequentially search from the beginning of the file until a matching special file name or file system file name is found, or until EOF is encountered. *Getfstype* does likewise, matching on the file system type field.

FILES

/etc/checklist

AUTHOR

Getfsent was developed by the Hewlett-Packard Company, and the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

checklist(4)

DIAGNOSTICS

Null pointer (0) returned on EOF, invalid entry or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent, fgetgrent – get group file entry

SYNOPSIS

#include <grp.h>

struct group *getgrent ( )

struct group *getgrgid (gid)
int gid;

struct group *getgrnam (name)
char *name;

void setgrent ( )

void endgrent ( )

struct group *fgetgrent (f)
FILE *f;

DESCRIPTION

Getgrent, *getgrgid* and *getgrnam* each return pointers to an object with the following structure containing the broken-out fields of a line in the **/etc/group** file. Each line contains a "group" structure, defined in the <*grp.h*> header file.

```
struct  group {
        char    *gr_name;      /* the name of the group */
        char    *gr_passwd;    /* the encrypted group password */
        int     gr_gid;        /* the numerical group ID */
        char    **gr_mem;      /* null-terminated vector of pointers to member names */
};
```

Getgrent when first called returns a pointer to the first group structure in the file; thereafter, it returns a pointer to the next group structure in the file; so, successive calls may be used to search the entire file. *Getgrgid* searches from the beginning of the file until a numerical group id matching *gid* is found and returns a pointer to the particular structure in which it was found. *Getgrnam* searches from the beginning of the file until a group name matching *name* is found and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setgrent* has the effect of rewinding the group file to allow repeated searches. *Endgrent* may be called to close the group file when processing is complete.

Fgetgrent returns a pointer to the next group structure in the stream *f*, which matches the format of **/etc/group**.

FILES

/etc/group

SEE ALSO

getlogin(3C), getpwent(3C), group(4).

DIAGNOSTICS

A **NULL** pointer is returned on **EOF** or error.

WARNING

The above routines use **<stdio.h>**, which causes them to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

getlogin – get login name

SYNOPSIS

char *getlogin ( );

DESCRIPTION

Getlogin returns a pointer to the login name as found in **/etc/utmp**. It may be used in conjunction with *getpwnam* to locate the correct password file entry when the same user ID is shared by several login names.

If *getlogin* is called within a process that is not attached to a terminal, it returns a **NULL** pointer. The correct procedure for determining the login name is to call *cuserid*, or to call *getlogin* and if it fails to call *getpwuid*.

FILES

/etc/utmp

SEE ALSO

getgrent(3C), getpwent(3C), cuserid(3S), utmp(4).

DIAGNOSTICS

Getlogin returns the **NULL** pointer if *name* is not found.

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

getmsg – get message from a catalog

SYNOPSIS

char *getmsg (fd, set_num, msg_num, buf, buflen)
int fd, set_num, msg_num, buflen;
char buf[];

DESCRIPTION

Getmsg attempts to read up to *buflen-1* bytes of a message string into the area pointed to by *buf*. A null byte is inserted to terminate the string placed in the buffer.

Fd is the file descriptor returned by a call to *open*(2) the catalog containing the messages. *Set_num* is available to group messages together into a logical unit. For instance, messages in Finnish could be grouped in set number 6 to match the language id for Finnish (See *langinfo*(3C) and *langid*(5)).

DIAGNOSTICS

Returns a pointer to an empty (null) string if *fd* is invalid or *set_num* or *msg_num* is not in the catalog.

AUTHOR

Getmsg was developed by the Hewlett-Packard Company.

SEE ALSO

gencat(1), insertmsg(1), read(2), hpnls(5).

INTERNATIONAL SUPPORT

8- and 16-bit data, messages.

NAME

getopt, optarg, optind, opterr – get option letter from argument vector

SYNOPSIS

int getopt (argc, argv, optstring)
int argc;
char **argv, *opstring;

extern char *optarg;
extern int optind, opterr;

DESCRIPTION

Getopt returns the next option letter in *argv* (starting from *argv*[1]) that matches a letter in *optstring*. *Optstring* is a string of recognized option letters; if a letter is followed by a colon, the option is expected to have an argument that may or may not be separated from it by white space. *Optarg* is set to point to the start of the option argument on return from *getopt*.

Getopt places in *optind* the *argv* index of the next argument to be processed. The external variable *optind* is initialized to 1 before the first call to the function *getopt.*

When all options have been processed (i.e., up to the first non-option argument), *getopt* returns **EOF**. The special option — may be used to delimit the end of the options; **EOF** will be returned, and — will be skipped.

DIAGNOSTICS

Getopt prints an error message on *stderr* and returns a question mark (**?**) when it encounters an option letter not included in *optstring*. This error message may be disabled by setting *opterr* to zero.

WARNING

Options can be any ASCII characters except colon (:), question mark (?), or null (\0). It is impossible to distinguish between a ? used as a legal option, and the character that *getopt* returns when it encounters an invalid option character in the input.

EXAMPLE

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options **a** and **b**, and the options **f** and **o**, both of which require arguments:

```
main (argc, argv)
int argc;
char **argv;
{
        int c;
        extern char *optarg;
        extern int optind;
        .
        .
        .
        while ((c = getopt(argc, argv, "abf:o:")) != EOF)
                switch (c) {
                case 'a':
                        if (bflg)
                                errflg++;
                        else
                                aflg++;
                        break;
                case 'b':
                        if (aflg)
                                errflg++;
                        else
                                bproc( );
                        break;
                case 'f':
                        ifile = optarg;
                        break;
                case 'o':
                        ofile = optarg;
                        break;
                case '?':
                        errflg++;
                }
        if (errflg) {
                fprintf(stderr, "usage: . . . ");
                exit (2);
        }
        for ( ; optind < argc; optind++) {
                if (access(argv[optind], 4)) {
        .
        .
        .
}
```

SEE ALSO

getopt(1).

NAME

getpass – read a password

SYNOPSIS

char *getpass (prompt)
char *prompt;

DESCRIPTION

Getpass reads up to a newline or **EOF** from the file **/dev/tty**, after prompting on the standard error output with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters. If **/dev/tty** cannot be opened, a **NULL** pointer is returned. An interrupt will terminate input and send an interrupt signal to the calling program before returning.

FILES

/dev/tty

SEE ALSO

crypt(3C).

WARNING

The above routine uses **<stdio.h>**, which causes it to increase the size of programs not otherwise using standard I/O, more than might be expected.

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

getpw – get name from UID

SYNOPSIS

int getpw (uid, buf)
int uid;
char *buf;

DESCRIPTION

Getpw searches the password file for a user id number that equals *uid*, copies the line of the password file in which *uid* was found into the array pointed to by *buf*, and returns 0. *Getpw* returns non-zero if *uid* cannot be found. The line is null-terminated.

This routine is included only for compatibility with prior systems and should not be used; see *getpwent*(3C) for routines to use instead.

FILES

/etc/passwd

SEE ALSO

getpwent(3C), passwd(4).

DIAGNOSTICS

Getpw returns non-zero on error.

WARNING

The above routine uses **<stdio.h>**, which causes it to increase, more than might be expected, the size of programs not otherwise using standard I/O.

NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent, fgetpwent – get password file entry

SYNOPSIS

#include <pwd.h>

struct passwd *getpwent ( )

struct passwd *getpwuid (uid)
int uid;

struct passwd *getpwnam (name)
char *name;

void setpwent ( )

void endpwent ( )

struct passwd *fgetpwent (f)
FILE *f;

DESCRIPTION

Getpwent, *getpwuid* and *getpwnam* each return a pointer to an object with the following structure containing the broken-out fields of a line in the **/etc/passwd** file. Each line in the file contains a "passwd" structure, declared in the <*pwd.h*> header file:

```
struct passwd {
        char        *pw_name;
        char        *pw_passwd;
        int         pw_uid;
        int         pw_gid;
        char        *pw_age;
        char        *pw_comment;
        char        *pw_gecos;
        char        *pw_dir;
        char        *pw_shell;
};
```

This structure is declared in <*pwd.h*> so it is not necessary to redeclare it.

The *pw_comment* field is unused; the others have meanings described in *passwd*(4).

Getpwent when first called returns a pointer to the first passwd structure in the file; thereafter, it returns a pointer to the next passwd structure in the file; so successive calls can be used to search the entire file. *Getpwuid* searches from the beginning of the file until a numerical user id matching *uid* is found and returns a pointer to the particular structure in which it was found. *Getpwnam* searches from the beginning of the file until a login name matching *name* is found, and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setpwent* has the effect of rewinding the password file to allow repeated searches. *Endpwent* may be called to close the password file when processing is complete.

Fgetpwent returns a pointer to the next passwd structure in the stream *f*, which matches the format of **/etc/passwd**.

FILES

/etc/passwd

SEE ALSO

getlogin(3C), getgrent(3C), passwd(4).

DIAGNOSTICS

A **NULL** pointer is returned on **EOF** or error.

WARNING

The above routines use **<stdio.h>**, which causes them to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

gets, fgets – get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *gets (s)
char *s;

char *fgets (s, n, stream)
char *s;
int n;
FILE *stream;
```

DESCRIPTION

Gets reads characters from the standard input stream, *stdin,* into the array pointed to by *s*, until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a null character.

Fgets reads characters from the *stream* into the array pointed to by *s*, until *n*–1 characters are read, or a new-line character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), getc(3S), puts(3S), scanf(3S).

DIAGNOSTICS

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise *s* is returned.

NAME

getutent, getutid, getutline, pututline, setutent, endutent, utmpname – access utmp file entry

SYNOPSIS

#include <sys/types.h>
#include <utmp.h>

struct utmp *getutent ( )

struct utmp *getutid (id)
**struct utmp ***id;

struct utmp *getutline (line)
**struct utmp ***line;

void pututline (utmp)
**struct utmp ***utmp;

void setutent ( )

void endutent ( )

void utmpname (file)
**char ***file;

DESCRIPTION

Getutent, *getutid* and *getutline* each return a pointer to a structure of the following type:

```
struct      utmp {
      char        ut_user[8];      /* User login name */
      char        ut_id[4];        /* /etc/inittab id (usually line #) */
      char        ut_line[12];     /* device name (console, lnxx) */
      short       ut_pid;          /* process id */
      short       ut_type;         /* type of entry */
      struct      exit_status      {
                  short            e_termination;  /* Process termination status */
                  short            e_exit;         /* Process exit status */
      }           ut_exit;         /* The exit status of a process
      time_t      ut_time;         /* time entry was made */
};
```

Getutent reads in the next entry from a **utmp**-like file. If the file is not already open, it opens it. If it reaches the end of the file, it fails.

Getutid searches forward from the current point in the *utmp* file until it finds an entry with a *ut_type* matching *id >ut_type* if the type specified is RUN_LVL, BOOT_TIME, OLD_TIME or NEW_TIME. If the type specified in *id* is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS or DEAD_PROCESS *getutid* will return a pointer to the first entry whose type is one of these four and whose *ut_id* field matches *id >ut_id*. If the end of file is reached without a match, it fails.

Getutline searches forward from the current point in the *utmp* file until it finds an entry of the type LOGIN_PROCESS or USER_PROCESS that also has a *ut_line* string matching the *line >ut_line* string. If the end of file is reached without a match, it fails.

Pututline writes out the supplied *utmp* structure into the *utmp* file. It uses *getutid* to search forward for the proper place if it finds that it is not already at the proper place. It is expected that normally the user of *pututline* will have searched for the proper entry using one of the *getut* routines. If so, *pututline* will not search. If *pututline* does not find a matching slot for the new entry, it will add [illegible]

Setutent resets the input stream to the beginning of the file. This should be done before each search for a new entry if it is desired that the entire file be examined.

Endutent closes the currently open file.

Utmpname allows the user to change the name of the file examined, from **/etc/utmp** to any other file. It is most often expected that this other file will be **/etc/wtmp**. If the file does not exist, this will not be apparent until the first attempt to reference the file is made. *Utmpname* does not open the file. It just closes the old file if it is currently open and saves the new file name.

The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. Each call to either *getutid* or *getutline* sees the routine examine the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason to use *getutline* to search for multiple occurrences, it would be necessary to zero out the static after each success, or *getutline* would just return the same pointer over and over again. There is one exception to the rule about removing the structure before further reads are done. The implicit read done by *pututline* (if it finds that it is not already at the correct place in the file) will not hurt the contents of the static structure returned by the *getutent*, *getutid* or *getutline* routines, if the user has just modified those contents and passed the pointer back to *pututline*.

These routines use buffered standard I/O for input, but *pututline* uses an unbuffered non-standard write to avoid race conditions between processes trying to modify the *utmp* and *wtmp* files.

RETURNS

A **NULL** pointer is returned upon failure to read, whether for permissions or having reached the end of file, or upon failure to write.

A **NULL** pointer is also returned if the size of the file is not an integral multiple of **sizeof(struct utmp)**.

WARNINGS

Some vendors' versions of *getutent* erase the *utmp* file if the file exists but is not an integral multiple of **sizeof(struct utmp)**. Given the possiblity of user error in providing a name to *utmpname* (such as giving improper arguments to *who*(1)), HP-UX does not do this, but instead returns an error indication.

FILES

/etc/utmp
/etc/wtmp

SEE ALSO

ttyslot(3C), utmp(4).

NAME

gpio_get_status - return status lines of GPIO card

SYNOPSIS

int gpio_get_status (eid)
int eid;

DESCRIPTION

Gpio_get_status enables you to read the status register of the GPIO interface associated with the device file identified by *eid*. *Eid* is an entity identifier of an open GPIO device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2). The current state of each status line on the interface card is mapped to the value returned, with STS0 mapped to the least significant bit. Only x least-significant bits are used, where x is the number of status lines available on the hardware interface being used.

HARDWARE DEPENDENCIES

Series 200/300 and 500

Eid is an integer file descriptor (fildes) that identifies an open device special file.
For the current GPIO card, x is 2.

RETURNS

Gpio_get_status returns the value of the status register of the GPIO interface associated with *eid*, and −1 if an error was encountered.

ERRORS

Gpio_get_status fails under the following conditions and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to a GPIO device file [ENOTTY].

NAME

gpio_set_ctl - set control lines on GPIO card

SYNOPSIS

int gpio_set_ctl (eid, value)
int eid, value;

DESCRIPTION

Gpio_set_ctl enables you to set the control register of a GPIO interface. *Eid* is an entity identifier of an open GPIO device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Value* is the value to be written into the control register of the GPIO interface associated with *eid*.

Value is mapped onto the control lines on the interface card, with the least significant bit mapped to CTL0. Only the x least significant bits are used, where x is the number of control lines available on the hardware interface being used.

HARDWARE DEPENDENCIES

Series 200, Series 300, Series 500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

For the current GPIO card, x is 2.

RETURNS

Gpio_set_ctl returns 0 if successful, and -1 if an error was encountered.

ERRORS

Gpio_set_ctl fails under the following circumstances and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to a GPIO device file [ENOTTY].

NAME

hpib_abort – stop activity on specified HP-IB bus

SYNOPSIS

int hpib_abort (eid);
int eid;

DESCRIPTION

Hpib_abort terminates activity on the addressed HP-IB bus by pulsing the IFC line. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

Hpib_abort also sets the REN line and clears the ATN line. The status of the SRQ line is not affected. The interface must be the system controller of the bus.

HARDWARE DEPENDENCIES

Series 200/300:

The HP 98625A/B HP-IB interface does not clear the ATN line.

RETURN VALUE

Hpib_abort returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

Hpib_abort fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the specified interface is not the system controller [EIO].

AUTHOR

Hpib_abort was developed by the Hewlett-Packard Company.

NAME

hpib_bus_status – return status of HP-IB interface

SYNOPSIS

int hpib_bus_status (eid, status);
int eid, status;

DESCRIPTION

Hpib_bus_status enables you to determine various status information about an HP-IB channel. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Status* is an integer determining what status information is returned for a particular call. The values defined for *status* and their associated meanings are:

Value	Meaning
0	Is the channel currently in remote state?
1	What is the current state of the SRQ line?
2	What is the current state of the NDAC line?
3	Is the channel currently system controller?
4	Is the channel currently active controller?
5	Is the channel currently addressed as talker?
6	Is the channel currently addressed as listener?
7	What is the channel's bus address?

The remote state status is not defined when the interface is the active controller, although reading remote state status in such a situation is not an error. Determining the status of the NDAC line is not available on all machines, and its use is therefore discouraged to ensure compatibility among various systems. Machines which do not support sensing the NDAC line return an error.

HARDWARE DEPENDENCIES

Series 200, Series 300

The status of those lines being driven by the interface is undefined, although reading them in such a situation is not an error. Non-active controllers cannot sense the SRQ line. Active listeners cannot sense the NDAC line.

The HP 98625A/B HP-IB interface cannot determine the current state of the NDAC line. Attempts to read this line will fail and set ERRNO (see *errno*(2)) to EINVAL.

Series 500

A bug in the HP27110A HP-IB interface causes an erroneous report of the state of the SRQ line. There is a small window when *hpib_bus_status*(eid, 1) reports that the SRQ line is clear when in reality it is set. OR-ing together five successive readings of the state of the SRQ line yields a reading of about 99% accuracy.

RETURN VALUE

Hpib_bus_status's return value depends upon the value of *status*, as follows:

Status	Return Value	Meaning
—	-1	Error condition.
0 – 6	0	False condition (line is clear).
0 – 6	1	True condition (line is set).
7	0 – 30	Bus address of interface card.

ERRORS

Hpib_bus_status fails under the following conditions, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to an HP-IB raw bus device file

AUTHOR

Hpib_bus_status was developed by HP.

NAME

hpib_card_ppoll_resp – control response to parallel poll on HP-IB

SYNOPSIS

int hpib_card_ppoll_resp (eid,flag);
int eid,flag;

DESCRIPTION

Hpib_card_ppoll_resp enables an interface to enable (or disable) itself for parallel polls. It also controls the sense, and determines the line on which the response is sent. This gives the interface the ability to either ignore or respond to a parallel poll depending upon whether or not it is enabled to respond.

Eid is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Flag* is an integer having one of the following bit patterns:

Bit Pattern	**Meaning**
10000	Disable parallel poll response.
0SPPP	Enable parallel poll response, where S = sense of the response, and PPP = 3-bit binary number specifying the line on which the response is sent (0 – 7 octal).

RETURN VALUE

Hpib_card_ppoll_resp returns 0 (zero) if successful, or −1 if an error was encountered.

ERRORS

Hpib_card_ppoll_resp fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EINVAL]	the device cannot respond on the line number specified by *flag*.

HARDWARE DEPENDENCIES

Series 500

Note that the HP 27110A HP-IB interface supports only enabling and disabling the parallel poll response (bit 4 of *flag*). The sense and response line number are not programmable on this card.

Series 200/300

The HP 98625A/B HP-IB interface supports only enabling and disabling the parallel poll response (bit 4 of *flag*). The sense and response line number are not programmable on this card.

Series 800

Since the sense and response line number are not programmable on the HP27110B HP-IB interface, the equivalent parallel poll configuration commands are sent over the HP-IB to the interface. Therefore, this function will fail if the interface is not active controller.

AUTHOR

Hpib_card_ppoll_resp was developed by HP.

SEE ALSO

hpib_ppoll(3I), hpib_ppoll_resp_ctl(3I).

NAME

hpib_eoi_ctl – control EOI mode for HP-IB file

SYNOPSIS

int hpib_eoi_ctl (eid, flag);
int eid, flag;

DESCRIPTION

Hpib_eoi_ctl enables you to turn EOI mode on or off. *Eid* is an entity identifier of an open HP-IB raw device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Flag* is an integer which, if non-zero, enables EOI mode, and otherwise disables it.

EOI mode causes the last byte of all subsequent write operations to be written out with the EOI line asserted, signifying the end of the data transmission. By default, EOI mode is disabled when the device file is opened.

Entity ids for the same device file obtained by separate *open*(2) requests have their own EOI modes associated with them. Entity ids for the same device file obtained by *dup*(2) or inherited by a *fork*(2) request share the same EOI mode. In the latter case, if one process enables EOI mode, then EOI mode is in effect for all such entity ids.

RETURN VALUE

Hpib_eoi_ctl returns a 0 (zero) if successful, or −1 if an error was encountered.

ERRORS

Hpib_eoi_ctl fails under any of the following circumstances and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB device file [ENOTTY].

HARDWARE DEPENDENCIES

Series 800

EOI mode is enabled when the device file is first opened.

EOI mode is associated with a given device. Therefore, multiple opens of the same device share EOI mode.

AUTHOR

Hpib_eoi_ctl was developed by HP.

NAME

hpib_io – perform I/O with an HP-IB channel from buffers

SYNOPSIS

#include <dvio.h>
int hpib_io(eid, iovec, iolen)
int eid;
struct iodetail *iovec;
int iolen;

DESCRIPTION

Hpib_io enables you to perform and control read and/or write operations on the specified HP-IB bus. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Iovec* is a pointer to an array of structures of the form:

```
struct iodetail { 1 1 1.
        charmode;
        char terminator;
        int  count;
        char *buf;
};
```

The *iodetail* structure is defined in the include file **libdvio.h**. *Iolen* specifies the number of structures in *iovec*.

The *mode* parameter in the *iodetail* structure describes what is to be done during I/O on the buffer pointed to by *buf*. *Mode* is constructed by OR-ing flags from the following list:

Only one of the following two flags *must* be specified:

HPIBREAD	Perform a read of the HP-IB bus, placing data into the accompanying buffer.
HPIBWRITE	Perform a write to the HP-IB bus, using data from the accompanying buffer.

The following flags may be used in most combinations (not all combinations are valid), or not at all:

HPIBATN	Data is written with ATN enabled.
HPIBEOI	Data written is terminated with EOI (this flag is ignored when HPIBATN is enabled).
HPIBCHAR	Data read is terminated with the character given in the *terminator* element of the *iodetail* structure.

Terminator describes the termination character, if any, that should be checked for on input. *Count* is an integer specifying the maximum number of bytes to be transferred.

A read operation terminates when either *count* is matched, an EOI is detected, or the designated *terminator* is detected (if HPIBCHAR is set in *mode*).

A write operation terminates when *count* is matched, and the final byte is sent with EOI asserted (if HPIBEOI is set in *mode*).

If HPIBATN is set in *mode*, then write operations occur with ATN enabled. Setting HPIBATN for a read operation is ignored and has no effect.

The members of the *iovec* array are accessed in order.

RETURN VALUES

If all transactions are successful, *hpib_io* returns a zero and updates the *count* element in each structure in the *iovec* array to reflect the actual number of bytes read or written.

If an error is encountered during a transaction defined by an element of *iovec*, *hpib_io* returns without completing any transactions that might follow. In particular, if an error occurs, *hpib_io* returns a −1, and the *count* element of the transaction which caused the error is set to −1.

ERRORS

Hpib_io fails under any of the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	a timeout occurs.
[EIO]	*eid* is not the active controller.

AUTHOR

Hpib_io was developed by the Hewlett-Packard Company.

NAME

hpib_pass_ctl – change active controllers on HP-IB

SYNOPSIS

int hpib_pass_ctl (eid, ba)
int eid, ba;

DESCRIPTION

Hpib_pass_ctl passes control of a bus to an inactive controller on that bus. The inactive controller becomes the active controller of that bus. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Ba* is the bus address of the intended device.

Not all devices can accept control. Pass control passes only active control of the bus. It cannot pass system control of the bus. The specified interface must be the current active controller but need not be the system controller. The pass control operation does not suspend your program if the inactive controller does not take active control of the bus. However, the interface is no longer active controller.

RETURN VALUE

Hpib_pass_ctl returns 0 (zero) if successful, or −1 if an error was encountered.

ERRORS

Hpib_pass_ctl fails under any of the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the interface is not the active controller.

AUTHOR

Hpib_pass_ctl was developed by the Hewlett-Packard Company.

NAME

hpib_ppoll – conduct parallel poll on HP-IB bus

SYNOPSIS

int hpib_ppoll (eid);
int eid;

DESCRIPTION

Hpib_ppoll conducts a parallel poll on an HP-IB bus. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

Devices enabled to respond and that are in need of service can then assert the appropriate DIO line. This enables the controller to determine which devices, if any, need service at a given time. *Hpib_ppoll* delays for 25 microseconds before returning with the response. The interface must be the active controller to conduct a parallel poll.

RETURN VALUE

Hpib_ppoll returns an integer value whose least significant byte corresponds to the byte formed by the 8 data input/output (DIO) lines. Devices enabled to respond to a parallel poll do so on the appropriate DIO line. DIO line 0 corresponds to the least significant bit in the response byte. A –1 return value indicates that an error occurred.

ERRORS

Hpib_ppoll fails under the following situations, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the interface is not current the active controller.

AUTHOR

Hpib_ppoll was developed by the Hewlett-Packard Company.

NAME

hpib_ppoll_resp_ctl – Define interface parallel poll response

SYNOPSIS

int hpib_ppoll_resp_ctl (eid, response)
int eid, response;

DESCRIPTION

Eid is an entity identifier of an open HP-IB raw bus device file, obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

Hpib_ppoll_resp_ctl defines a response to be sent when an active controller performs a parallel poll on an HP-IB interface. The value of *response* indicates whether this computer does or does not need service. A non-zero *response* value indicates that service is required. This statement only sets up a potential response; no actual response if generated when the statement is executed. The sense of the response and the line number to respond on are set by *hpib_card_ppoll_resp*(3I) or by the active controller.

RETURN VALUE

Hpib_ppoll_resp_ctl returns 0 if the response is successfully set, or -1 if an error has occured.

ERRORS

Hpib_ppoll_resp_ctl fails under the following situations, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to a raw HP-IB device file.

AUTHOR

Hpib_ppoll_resp_ctl was developed by the Hewlett-Packard Company.

SEE ALSO

hpib_ppoll(3I), hpib_card_ppoll_resp(3I)

NAME

hpib_ren_ctl – control the Remote Enable line on HP-IB

SYNOPSIS

int hpib_ren_ctl (eid, flag);
int eid, flag;

DESCRIPTION

Hpib_ren_ctl enables/disables the Remote Enable (REN) line depending upon the value of *flag*. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Flag* is an integer which, if non-zero, enables the REN line, and otherwise disables it.

Hpib_ren_ctl, in conjunction with *hpib_send_cmnd*(3I), enables you to place devices into the remote state or local state. The REN line is normally enabled at all times, and is in this state at power-up. Only the system controller may enable/disable the REN line.

RETURN VALUE

Hpib_ren_ctl returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

Hpib_ren_ctl fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the interface is not the system controller.

AUTHOR

Hpib_ren_ctl was developed by the Hewlett-Packard Company.

NAME

hpib_rqst_srvce – allow interface to enable SRQ line on HP-IB

SYNOPSIS

int hpib_rqst_srvce (eid, cv);
int eid, cv;

DESCRIPTION

Hpib_rqst_srvce specifies the response byte that the interface sends when it is serially polled by the active controller. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Cv* is an integer control value representation of the desired response byte.

Hpib_rqst_srvce optionally enables the SRQ line depending upon the response byte. If bit 6 of the response byte is set, the SRQ line is enabled. It remains enabled until the active controller conducts a serial poll or until the computer executes the request function with bit 6 cleared. The SRQ line is not enabled, however, as long as the interface is active controller. If bit 6 is set, the interface remembers its response byte, and enables the SRQ line when control is passed to another device on the bus.

The response byte looks as follows:

Bit	**Meaning**
0	SPOLL bit (least significant bit of response byte)
1	SPOLL bit
2	SPOLL bit
3	SPOLL bit
4	SPOLL bit
5	SPOLL bit
6	SRQ line
7	SPOLL bit (most significant bit of response byte)

HARDWARE DEPENDENCIES

Series 500, 800:
Note that the HP 27110A HP-IB interface card allows only bit 6 to be set. All other bits are cleared.

Series 200, 300:
The HP 98625A/B HP-IB interface card allows only bit 6 to be set. All other bits are cleared.

RETURN VALUE

Hpib_rqst_srvce returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

Hpib_rqst_srvce fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.

AUTHOR

Hpib_rqst_srvce was developed by the Hewlett-Packard Company.

NAME

hpib_send_cmnd – send command bytes over HP-IB

SYNOPSIS

int hpib_send_cmnd (eid, ca, length);
int eid, length;
char *ca;

DESCRIPTION

Hpib_send_cmnd enables you to send arbitrary bytes of information on the HP-IB with the ATN line asserted. This enables you to configure and control the bus. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Ca* is a character pointer to a string of bytes to be written to the HP-IB bus as commands. *Length* is an integer specifying the number of bytes in the string pointed to by *ca.*

The interface must currently be the active controller in order to send commands over the bus.

Note that, for all HP-IB interfaces, both built-in and plug-in, the most significant bit of each byte is overwritten with a parity bit. All commands are written with odd parity.

RETURN VALUE

Hpib_send_cmnd returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

Hpib_send_cmnd fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the interface is not currently the active controller.

AUTHOR

Hpib_send_cmnd was developed by HP.

NAME

hpib_spoll – conduct a serial poll on HP-IB bus

SYNOPSIS

int hpib_spoll (eid, ba);
int eid, ba;

DESCRIPTION

Hpib_spoll conducts a serial poll of the specified device. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Ba* is the bus address of the intended device.

Hpib_spoll polls a single device for its response byte. The information stored in the response byte is device specific with the exception of bit 6. If bit 6 of the response byte is set, the addressed device has asserted the SRQ line, and is requesting service. (Note that the least significant (right-most) bit of the response byte is bit 0.)

Not all devices respond to the serial poll function. Consult the device documentation. Specifying a device that does not support serial polling may cause a timeout error or suspend your program indefinitely (see *hpib_rqst_srvce*(3I)). The interface cannot serial poll itself. The interface must be the active controller.

RETURN VALUE

If *hpib_spoll* is successful, the device response byte is returned in the least significant byte of the return value. Otherwise, -1 is returned, indicating an error.

ERRORS

Hpib_spoll fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBAD]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the device polled did not respond before timeout, or the interface is not the active controller.
[EINVAL]	*ba* is the address of the polling interface itself.

AUTHOR

Hpib_spoll was developed by the Hewlett-Packard Company.

SEE ALSO

hpib_rqst_srvce(3I).

NAME

hpib_status_wait – wait until the requested status condition becomes true

SYNOPSIS

int hpib_status_wait (eid, status);
int eid,status;

DESCRIPTION

Hpib_status_wait enables you to wait until a specific condition has occurred before returning. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Status* is an integer specifying what information is returned. The possible values for *status* and their associated meanings are:

Status	**Meaning**
1	Wait until the SRQ line is enabled.
4	Wait until this channel is the active controller.
5	Wait until this channel is addressed as talker.
6	Wait until this channel is addressed as listener.

The wait is subject to the current timeout in effect. If a timeout occurs before the desired condition occurs, the function returns with an error.

RETURN VALUE

Hpib_status_wait returns zero when the condition requested becomes true. A value of −1 is returned if an error occurs. A −1 is also returned if a timeout occurs before the desired condition becomes true.

ERRORS

Hpib_status_wait fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	a timeout occurred.
[EINVAL]	*status* contains an invalid value.

AUTHOR

Hpib_status_wait was developed by the Hewlett-Packard Company.

NAME

hpib_wait_on_ppoll – wait until a particular parallel poll value occurs

SYNOPSIS

int hpib_wait_on_ppoll (eid, mask, sense);
int eid, mask, sense;

DESCRIPTION

Hpib_wait_on_ppoll waits for a parallel poll response to occur on one or more lines. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

Mask is an integer that specifies on which line the parallel poll response is expected. *Mask*'s value is obtained from an 8-bit binary number, each bit of which corresponds to one of the eight lines. For example, if you want to wait for a response on lines 2 and 6, the correct binary number is 01000100. This converts to a decimal equivalent of 68, which is the number you should assign to *mask.*

Sense simply specifies what response you are expecting on the selected lines. *Sense* is constructed in the same way as *mask* – eight bits for eight lines. If a bit is set, then the function returns when the line corresponding to that bit is *cleared.* Similarly, if a bit in *sense* is clear, the function returns when the corresponding line is *set.* Using the previous example, a *sense* = 00000100 = 4 (decimal) causes the function to return when line 6 is set, and return when line 2 is cleared.

RETURN VALUE

Hpib_wait_on_ppoll returns a value of –1 if an error or timeout condition occurs. A successful completion of the function returns the response byte XOR-ed with the *sense* value and AND-ed with the *mask.*

ERRORS

Hpib_wait_on_ppoll fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	a timeout occurred.
[EIO]	the interface is not currently the active controller.

AUTHOR

Hpib_wait_on_ppoll was developed by the Hewlett-Packard Company.

NAME

hpibegin, hpiclose, hpicontrol, hpidelete, hpiend, hpierror, hpifind, hpifindset, hpiget, hpiinfo, hpilock, hpimemo, hpiopen, hpiput, hpiundo, hpiupdate, chpibegin, chpiclose, chpicontrol, chpidelete, chpiend, chpierror, chpifind, chpifindset, chpiget, chpiinfo, chpilock, chpimemo, chpiopen, chpiput, chpiundo, chpiupdate – ALLBASE/HP-UX HPIMAGE programmatic calls

REMARKS

The ALLBASE/HP-UX product must be previously installed on the system for *hpimage* programmatic calls to function.

DESCRIPTION

This set of calls invokes the appropriate *hpimage* procedure or function calls for programmatically accessing an ALLBASE/HP-UX HPIMAGE network database. FORTRAN and Pascal calls are invoked with the calls that begin with "hpi." C calls are invoked with the calls that begin with "chpi." The following descriptions apply to the C calls as well:

hpibegin	Designates the beginning of a transaction, and optionally writes user information to the log file.
hpiclose	Terminates access to a database or a data set.
hpicontrol	Enables or disables the return of chain information.
hpidelete	Deletes an entry from the database.
hpiend	Defines the end of a transaction, commits the transaction, and optionally writes user information to the log file.
hpierror	Supplies a natural language message that interprets the status array as set by any *hpimage procedure.*
hpifind	Locates the first and last entries of a data chain in preparation for accessing that chain.
hpifindset	Locates entries satisfying a given expression in preparation for access to those entries.
hpiget	Retrieves an entry in a data set.
hpiinfo	Provides structural information about the database being accessed.
hpilock	Locks a database or one or more data sets for exclusive access.
hpimemo	Writes user information to the log file.
hpiopen	Initiates access to a database.
hpiput	Adds a new entry to a data set.
hpiundo	Undoes an uncommitted tranaction and optionally writes user information to the log file. This procedure also defines the end of a transaction.
hpiupdate	Modifies an existing entry in a database.

The *hpimage* programmatic calls can be executed by all system users.

FILES

/usr/bin/hpdbdaemon	cleanup daemon program file
/usr/bin/hpimage	*hpimage* program file
/usr/lib/hpica000	*hpimage* message catalog file

AUTHOR

The *hpimage* programmatic calls were developed by Hewlett-Packard.

SEE ALSO

ALLBASE/HP-UX HPIMAGE Reference Manual.

NAME

hsearch, hcreate, hdestroy – manage hash search tables

SYNOPSIS

#include <search.h>

ENTRY *hsearch (item, action)
ENTRY item;
ACTION action;

int hcreate (nel)
unsigned nel;

void hdestroy ( )

DESCRIPTION

Hsearch is a hash-table search routine generalized from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry can be found. *Item* is a structure of type ENTRY (defined in the *<search.h>* header file) containing two pointers: *item.key* points to the comparison key, and *item.data* points to any other data to be associated with that key. (Pointers to types other than character should be cast to pointer-to-character.) *Action* is a member of an enumeration type ACTION indicating the disposition of the entry if it cannot be found in the table. **ENTER** indicates that the item should be inserted in the table at an appropriate point. **FIND** indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a NULL pointer.

Hcreate allocates sufficient space for the table, and must be called before *hsearch* is used. *Nel* is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

Hdestroy destroys the search table, and may be followed by another call to *hcreate*.

EXAMPLE

The following example will read in strings followed by two numbers and store them in a hash table, discarding duplicates. It will then read in strings and find the matching entry in the hash table and print it out.

```
#include <stdio.h>
#include <search.h>

struct info {           /* this is the info stored in the table */
        int age, room;  /* other than the key. */
};
#define NUM_EMPL    5000    /* # of elements in search table */

main( )
{
        /* space to store strings */
        char string_space[NUM_EMPL*20];

        /* space to store employee info */
        struct info info_space[NUM_EMPL];

        /* next avail space in string_space */
        char *str_ptr = string_space;

        /* next avail space in info_space */
```

```
            struct info *info_ptr = info_space;
            ENTRY item, *found_item, *hsearch( );
            /* name to look for in table */

            char name_to_find[30];
            int i = 0;

            /* create table */
            (void) hcreate(NUM_EMPL);
            while (scanf("%s%d%d", str_ptr, &info_ptr->age,
                   &info_ptr->room) != EOF && i++ < NUM_EMPL) {

                   /* put info in structure, and structure in item */
                   item.key = str_ptr;
                   item.data = (char *)info_ptr;
                   str_ptr += strlen(str_ptr) + 1;
                   info_ptr++;

                   /* put item into table */
                   (void) hsearch(item, ENTER);
            }

            /* access table */
            item.key = name_to_find;
            while (scanf("%s", item.key) != EOF) {
                if ((found_item = hsearch(item, FIND)) != NULL) {

                   /* if item is in the table */
                   (void)printf("found %s, age = %d, room = %d\n",
                          found_item->key,
                          ((struct info *)found_item->data)->age,
                          ((struct info *)found_item->data)->room);
                } else {
                   (void)printf("no such employee %s\n",
                          name_to_find)
                }
            }
      }
```

SEE ALSO

bsearch(3C), lsearch(3C), malloc(3C), string(3C), tsearch(3C), malloc(3X).

DIAGNOSTICS

Hsearch returns a NULL pointer if either the action is **FIND** and the item could not be found or the action is **ENTER** and the table is full.

Hcreate returns zero if it cannot allocate sufficient space for the table.

WARNING

Hsearch and *hcreate* use *malloc*(3C) to allocate space.

BUGS

Only one hash search table may be active at any given time.

NAME

hypot – Euclidean distance function

SYNOPSIS

```
#include <math.h>

double hypot (x, y)
double x, y;
```

DESCRIPTION

Hypot returns sqrt(x * x + y * y), taking precautions against unwarranted overflows.

ERRORS

When the correct value would overflow, *hypot* returns **HUGE** and sets *errno* to **ERANGE.**

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M).

NAME

initgroups – initialize group access list

SYNOPSIS

```
initgroups(name, basegid)
char *name;
int basegid;
```

DESCRIPTION

Initgroups reads through the login group file and sets up, using the *setgroups*(2) call, the group access list for the user specified in *name*. The *basegid* is automatically included in the groups list. Typically this value is given as the group number from the password file. If the login group file is non-existent or empty *basegid* is the only member of the list.

FILES

/etc/logingroup

AUTHOR

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

SEE ALSO

login(1), su(1), setgroups(2), group(4)

DIAGNOSTICS

Initgroups returns −1 if it was not invoked by the super-user.

BUGS

Initgroups uses the routines based on *getgrent*(3C). If the invoking program uses any of these routines, the group structure will be overwritten in the call to *initgroups*.

On most systems, no one seems to keep **/etc/logingroup** up to date.

NAME

intrapoff, intrapon – disable/enable integer trap handler

SYNOPSIS

int intrapoff()

int intrapon()

Remarks:

Intrapoff and *intrapon* are implemented on the Series 500 only.

DESCRIPTION

The Series 500 architecture has a single trap handler for both integer overflow (an integer value greater than 2^31-1) and integer divide-by-zero. By default, an operation which results in integer overflow or integer divide-by-zero invokes the integer trap handler. Any integer divide-by-zero generates the signal SIGFPE. As a side effect, any integer overflow also invokes the integer trap handler. The trap handler recognizes integer overflow as a special case and simply returns to the calling routine. A user sees no difference in results, but could see a severe performance degradation depending on how often the trap handler is invoked.

Intrapoff disables this integer trap handler. Integer overflow and integer divide-by-zero do not invoke the integer trap handler. Instead, integer divide-by-zero returns a large integer (2^31-1). Integer overflow operations simply overflow into the most significant bit. There is no performance penalty since the trap handler is not entered.

A program doing many integer overflows could see a significant performance improvement. A user must take care however, since integer divide-by-zero does not give signal SIGFPE while the integer trap handler is disabled.

Intrapon restores the default condition. Integer divide-by-zero and integer overflow operations invoke the integer trap handler. Integer divide-by-zero gives signal SIGFPE; integer overflow results in a performance penalty caused by entering and leaving the integer trap handler.

When *intrapoff* is used, the integer trap handler is disabled at that procedural level and all levels below it. It is not disabled for any procedural level above the procedure within which *intrapoff* was called. For example,

```
a();
{
        b();      /* Call function b. */
}
b();
{
        intrapoff();
        c();      /* Call function c. */
}
c();
{
        /* Do some work. */
}
```

The integer trap handler is disabled for functions b and c. It is automatically re-enabled on exit from function b. The integer trap handler can also be re-enabled at any time using *intrapon*.

EXAMPLES

The math library routine *rand* generates random integers using:

```
randx = randx * (((1103515245L + 12345)>>16) & 0x7fff)
```

where randx is an unsigned integer. The value assigned to randx is often greater than 2^31-1. To avoid the performance degradation of entering the integer trap handler each time this occurs, the integer trap can be turned off before the assignment using *intrapoff*.

NAME

io_burst – perform low-overhead I/O on an HP-IB/GPIO channel

SYNOPSIS

#include <dvio.h>
io_burst (eid, flag)

DESCRIPTION

Io_burst enables you to perform low-overhead burst transfers on the specified HP-IB or GPIO channel. **Eid** is the entity identifier for an open HP-IB/GPIO device file returned by a previous call to *open*(2), *dup*(2), *creat*(2), or *fcntl*(2) with an FDUPD command option. **Flag** is an integer which, if non-zero, enables burst mode or, if zero, disables it.

In burst mode, memory-mapped I/O address space assigned to the interface card select code is mapped directly into user space such that data can be transferred directly between user memory and the interface card, eliminating the need for kernel calls and the associated overhead. Burst mode affects only *read*(2), *write*(2), *gpio_get_status*(3I), *gpio_set_ctl*(3I), *hpib_io*(3I), and *hpib_send_cmd*(3I) calls. All other operations are unaffected. When burst mode is enabled, the interface is locked so that no other process can access it until burst mode is disabled.

To minimize overhead, termination reason (see *io_get_term_reason*(3I)) is not supported in burst mode.

HARDWARE DEPENDENCIES

Series 200/300:

Eid is in integer file descriptor (fildes) that identifies an open device special file.

Timeouts for *read*(2), *write*(2), *gpio_get_status*(3I), *gpio_set_ctl*(3I), *hpib_io*(3I), and *hpib_send_cmd*(3I) do not work while in burst mode, but these commands can be interrupted by signals.

RETURN VALUE

Io_burst returns zero if successful or –1 if an error is detected.

DIAGNOSTICS

Io_burst fails under any of the following circumstances and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF]	**eid** does not refer to an open file.
[ENOTTY]	**eid** does not refer to an HP-IB or GPIO device special file.
[EIO]	a timeout occurred during the call to *ioburst*.

WARNING

Enabling burst mode locks the interface from all other processes, so it should never be used with any interface that supports a system disk or swap device.

SEE ALSO

read(2), write(2), gpio_get_status(3I), gpio_set_ctl(3I), hpib_io(3I), hpib_send_cmd(3I).

NAME

io_eol_ctl – set up read termination character on special file

SYNOPSIS

int io_eol_ctl (eid, flag, match);
int eid, flag, match;

DESCRIPTION

Io_eol_ctl enables you to specify a character to be used in terminating a read operation from the specified file id.

Eid is an entity identifier of an open HP-IB raw bus or GPIO device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Flag* is an integer which enables or disables character-match termination. A non-zero value enables character-match termination, while a zero value disables it. *Match* is an integer containing the numerical equivalent of the termination character. *Match* is ignored if *flag* is zero. When in 8-bit mode, the lower 8 bits of *match* are used as the termination character. In 16-bit mode, the lower 16 bits are used.

Upon opening a file, the default condition is character-match termination disabled. When enabled, the character specified by *match* is checked for during read operations. The read is terminated upon receipt of this character, or upon any of the other termination conditions normally in effect for this file. Examples of other conditions are satisfying the specified byte count, and receiving a character when the EOI line is asserted (HP-IB). When the read is terminated by a *match* character, this character is the last character returned in the buffer.

Entity ids for the same device file obtained by separate *open*(2) requests have their own termination characters associated with them. Entity ids for the same device file inherited by a *fork*(2) request share the same termination character. In the latter case, if one process changes the termination character, the new termination character is in effect for all such entity ids.

RETURN VALUE

Io_eol_ctl returns 0 (zero) if successful, or -1 if an error was encountered.

ERRORS

Io_eol_ctl fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file [EBADF];
[ENOTTY]	*eid* does not refer to a channel device file.

HARDWARE DEPENDENCIES

Series 500

When termination is requested in 16-bit mode, the upper byte of the halfword is examined first, and then the lower byte. If the lower byte matches the termination character, all is as expected. However, if the upper byte matches, the following action is taken:

both the upper and lower bytes are moved into the given buffer; and the count returned is odd, indicating that there is a lower byte following the matching upper byte. This information is passed to the upper level software to deal with as it pleases.

Series 800

Termination patterns are associated with a given device. Therefore, multiple opens of the same device share termination characters.

The GPIO interface does not support this function.

AUTHOR

Io_eol_ctl was developed by HP.

SEE ALSO

io_width_ctl(3I).

NAME

io_get_term_reason – determine how last read terminated

SYNOPSIS

int io_get_term_reason (eid);
int eid;

DESCRIPTION

Io_get_term_reason returns the termination reason for the last read made on this entity id. *Eid* is an entity identifier of an open HP-IB raw bus or GPIO device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

All entity ids descending from an *open*(2) request (such as from *dup*(2) or *fork*(2)) set this status. For example, if the calling process had opened this entity id, and later forked, the status returned would be from the last read done by either the calling process or its child.

RETURN VALUE

Io_get_term_reason returns a value indicating how the last read on the specified entity id was terminated. This value is interpreted as follows (note that combinations are possible): l l r l. **Value Description** –1 An error was encountered while making this function request. 0 Last read encountered some abnormal termination
reason not covered by any of the other reasons. 1 Last read terminated by reading the number of bytes requested. 2 Last read terminated by detecting the specified termination character. 4 Last read terminated by detecting some device-imposed termination condition. Examples are: EOI for
HP-IB, PSTS line on GPIO, or
some other end-of-record condition, such as the physical
end-of-record mark on a 9-track tape.

ERRORS

Io_get_term_reason fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to a channel device file.

HARDWARE DEPENDENCIES

Series 200, Series 300

For the GPIO interface, PSTS is checked only at the beginning of a transfer. An interrupt caused by an EIR will also terminate a transfer. The value of the termination reason in this case is also 4.

Series 500

If the last read had multiple applicable termination reasons, such as having EOI asserted on the last byte when that byte was the termination match character (see *io_eol_ctl*(3I)), the highest numbered reason is used (in this case, 4). Since interactive terminals are treated as record-oriented devices when they are in cooked mode, the termination reason is 4 when terminated by a new-line character. If no read has been done on the entity id since it was opened, the value of the termination reason is 0.

Series 800

If a read request times out, *io_get_term_reason* will return a value of 8.

The GPIO interface does not support this function.

AUTHOR

Io_get_term_reason was developed by HP.

SEE ALSO

read(2), io_eol_ctl(3I).

NAME

io_interrupt_ctl – enable/disable interrupts for the associated eid

SYNOPSIS

int io_interrupt_ctl (eid, enable_flag)
int eid, enable_flag;

DESCRIPTION

Eid is an entity identifier of an open HP-IB raw bus or GPIO device file, obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Flag* is an integer which enables or disables interrupts for the associated *eid.* A non-zero value enables interrupts.

Interrupts may be disabled or enabled by the user as desired. When an interrupt occurs for a given *eid* the interrupts associated with this *eid* are automatically disabled from reoccurring. Interrupts for this *eid* may be re–enabled by the user with *io_interrupt_ctl.*

RETURN VALUE

io_interrupt_ctl returns 0 (zero) if successful, or -1 if an error was encountered.

ERRORS

Io_interrupt_ctl fails under the following situations, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to a device that supports interrupts.
[EINVAL]	no interrupt conditions were specified for this *eid.*

AUTHOR

Io_interrupt_ctl was developed by the Hewlett-Packard Company.

SEE ALSO

io_on_interrupt(3I)

NAME

io_lock, io_unlock – lock and unlock an interface

SYNOPSIS

int io_lock (eid)
int eid;
int io_unlock (eid)
int eid;

DESCRIPTION

Eid is an entity identifier of an open HP-IB or GPIO, device file, obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

This function attempts to lock the interface associated with an entity identifier to the requesting process. Locking an interface gives exclusive use of the interface associated with the *eid* to the requesting process, thus avoiding unintended interference from other processes during a series of separate I/O requests. All the locks for a process are removed when the process closes the file or terminates.

Other processes that attempt to access or lock a locked interface will either return an error or sleep until the interface becomes unlocked. The action taken is determined by the current setting of the *O_NDELAY* flag (see *open*(2). If the *O_NDELAY* flag is set, accesses to a locked interface will fail and set *errno* to indicate the error. If the *O_NDELAY* flag is not set, accesses to a locked interface will block until the interface is unlocked, the current timeout expires, or the request is interrupted by a signal.

A lock is associated with a process, not an *eid.* Locking an interface with a particular *eid* does not prevent the process that owns the lock from accessing the interface through another *eid.* A lock associated with an *eid* is not inherited by a child process during a *fork*(2).

Nested locking is fully supported. If a process owns a locked interface and calls a generic subroutine that does a lock and unlock, the calling process does not lose its lock on the interface. Locking requests produced by a given process for an interface already locked by the same process will increment the current lock count for that interface.

Io_unlock allows a process to remove a lock from the interface associated with the *eid.* A locked interface can be unlocked only by the process directly owning the lock. When an unlock operation is applied to an *eid* that is currently multiply locked, the unlock operation decrements the current lock counter for that interface, and the interface remains locked until the count is reduced to zero.

RETURNS

Io_lock and *io_unlock* return the integer value of the current lock count if successful. A lock count greater than zero indicates that the interface is still locked. A lock count of zero indicates that the interface is no longer locked. A –1 indicates that an error has occured.

Io_lock and *io_unlock* fail in the following situations, and set *errno* (see *errno*(2)) to the value in square brackets:

[EACCES]	an attempt is made to lock an interface locked by another process with O_NDELAY set.
[EBADF]	an *eid* does not refer to an open file.
[EINTR]	a signal is caught while attempting to perform the lock with O_NDELAY clear.
[EINVAL]	an attempt is made to unlock when the interface is not locked.
[EIO]	a timeout occurs while attempting to perform the lock with O_NDELAY clear.
[ENOTTY]	an *eid* does not refer to a channel device file.

[EPERM] an attempt is made to unlock when lock is not owned by this user.

WARNING

Io_lock provides a mandatory lock enforced by the system and should not be used with any interface supporting a system disk or swap device.

AUTHOR

Io_lock and *io_unlock* were developed by HP.

SEE ALSO

io_timeout_ctl(3I), open(2).

NAME

io_on_interrupt – device interrupt (fault) control

SYNOPSIS

```
int (*io_on_interrupt (eid, causevec, handler))()
int eid;
struct interrupt_struct *causevec;
int (*handler)();

handler (eid, causevec)
int eid;
struct interrupt_struct *causevec;
```

DESCRIPTION

Eid is an entity identifier of an open HP-IB raw bus, or GPIO device file, obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

Causevec is a pointer to a structure of the form:

```
struct interrupt_struct {
     integer   cause;
     integer   mask;
};
```

The *interrupt_struct* structure is defined in the file **dvio.h**.

The *cause* parameter is a bit vector specifying which of the interrupt or fault events will cause the handler routine to be invoked. The interrupt causes are often specific to the type of interface being considered. As well, certain exception (error) conditions can be handled using the *io_on_interrupt* capability. Specifying a zero valued *cause* vector effectively turns off the interrupt for that eid.

The *mask* parameter is used when an *HP-IB* parallel poll interrupt is being defined. *Mask* is an integer that specifies which parallel poll response lines are of interest. *Mask*'s value is obtained from an 8-bit binary number, each bit of which corresponds to on of the eight lines. For example, if you want an interrupt handler invoked for a response on lines 2 or 6, the correct binary number is 01000100. This converts to a decimal equivalent of 68, which is the number you should assign to *mask*.

Upon occurrence of an enabled interrupt condition on the specified eid, the receiving process is to execute the interrupt-handler function pointed to by *handler*. The Entity identifier *eid* and the interrupt condition *cause* will be returned as the first and second parameters respectively.

When a interrupt that is to be caught occurs during a *read*, a *write*, an *open*, or an *ioctl* system call on a slow device (like a terminal; but not a file), during a *pause* system call, during a *sigpause*(2) system call, or during a *wait* system call that does not return immediately due to the existence of a previously stopped or zombie process, the interrupt handling function will be executed and then the interrupted system call will return a -1 to the calling process with *errno* set to EINTR.

Interrupts handlers are not inherited across a *fork*(2). *Eids* for the same device file produced by *dup*(2) share the same handler.

An interrupt for a given eid is implicitly disabled after the occurrence of the event. The interrupt condition may be re-enabled with *io_interrupt_ctl*(3I).

Upon the occurrence of an event specified by *cause*, the receiving process is to execute the interrupt handler function pointed to by *handler*. When the handler returns the user process resumes at the point of execution left when the event occurred.

Handler will be passed two parameters, the *eid* associated with the event, and a pointer to a *causevec* structure. The cause of the interrupt can be determined by the value returned in the *cause* field of the *causevec* structure. If the interrupt handler was invoked due to a parallel poll interrupt, then the *mask* field of the *causevec* structure will contain the parallel poll response byte.

HPIB INTERRUPTS

This section describes interrupt causes specific to an hpib device. For an hpib device the cause is a bit vector which is used as follows. To enable a given event, the appropriate bit (in cause), shown below, must be set to 1:

SRQ	SRQ and active controller.
TLK	Talker addressed.
LTN	Listener addressed.
TCT	Controller in charge.
IFC	IFC has been asserted
REN	Remote enable
DCL	Device clear
GET	Group execution trigger
PPOLL	Parallel poll

GPIO INTERRUPTS

This section describes interrupt causes specific to a GPIO device. For a GPIO device the cause is a bit vector which is used as follows. To enable a given event, the appropriate bit (in cause), shown below, must be set to 1:

EIR	External interrupt
SIE0	Status line 0
SIE1	Status line 1

RETURN VALUE

Io_on_interrupt returns a pointer to the previous handler if the new handler is successfully installed, otherwise it returns a -1 and errno is set.

ERRORS

Io_on_interrupt can fail for any of the following reasons:

[EBADF]	*Eid* does not refer to an open file.
[ENOTTY]	*Eid* does not refer to a GPIO or a raw HP-IB device file.
[EFAULT]	*Handler* points to an illegal address. The reliable detection of this error will be implementation dependent.
[EFAULT]	*causevec* points to an illegal address. [EFAULT] The reliable detection of this error will be implementation dependent.

HARDWARE DEPENDENCIES

Series 200

The default timeout for the GPIO interface is 15 seconds.

Series 200, 300

Timeout resolution is 20 msec.

Series 500

Parallel poll interrupts are not supported. The internal HP-IB supplied with the Model 550 will not support talker addressed, listener addressed, controller in charge, and remote enable interrupts. GPIO interrupts on the EIR lines are not supported.

Series 800

For the HP 27112 GPIO interface, the EIR interrupt is not available. For the HP 27114 GPIO interface, only the EIR interrupt is available.

For the

HP 98622 GPIO interface, only the EIR interrupt is available. For the HP 98265A/B HP-IB interface, the IPC and GET interrupts are not available.

AUTHOR

Io_on_interrupt was developed by HP.

SEE ALSO

pause(2), sigpause(2), io_interrupt_ctl(3I).

NAME

io_reset – reset an I/O interface

SYNOPSIS

int io_reset (eid);
int eid;

DESCRIPTION

Io_reset resets the interface associated with the device file that was opened. It also pulses the peripheral reset line on the GPIO interface, or the IFC line on the HP-IB. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

Io_reset also causes an interface to go through its self-test, and returns a failure indication if the interface fails its test.

RETURN VALUE

Io_reset returns 0 (zero) if successful, or −1 if an error was encountered.

ERRORS

Io_reset fails under the following circumstances, and sets errno (see *errno(2))* to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to a channel device file.
[EIO]	the interface could not be reset, or failed self-test.

HARDWARE DEPENDENCIES

Series 200, Series 300

When an HP-IB interface is reset, the interrupt mask is set to 0, the parallel poll response is set to 0, the serial poll response is set to 0, the HP-IB address is assigned, the IFC line is pulsed (if system controller), the card is put on line, and REN is set (if system controller).

When a GPIO interface is reset, the peripheral reset line is pulled low, the PCTL line is placed in the clear state, and if the DOUT CLEAR jumper is installed, the data out lines are all cleared. The interrupt enable bit is also cleared.

Interface self-test is not supported.

AUTHOR

Io_reset was developed by HP.

NAME

io_speed_ctl – inform system of required transfer speed

SYNOPSIS

int io_speed_ctl (eid, speed);
int eid, speed;

DESCRIPTION

Io_speed_ctl enables you to select the data transfer speed for a data path used for a particular interface. The transfer method (i.e., DMA, fast-handshake) chosen by the system is determined by the speed requirements.

Eid is an entity identifier of an open HP-IB raw bus or GPIO device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Speed* is an integer specifying the data transfer speed in K-bytes per second (one K-byte equals 1024 bytes).

RETURN VALUE

Io_speed_ctl returns 0 if successful, and −1 otherwise.

ERRORS

Io_speed_ctl fails under the following condition, and sets *errno* to the value enclosed in square brackets:

[ENOTTY]	*eid* does not refer to channel device file.
[EBADF]	*eid* does not refer to an open file.

HARDWARE DEPENDENCIES

Series 200, Series 300

For values of speed less than 7, the system will use an interrupt transfer. For larger values, DMA will be used if available; otherwise, the system will use an interrupt transfer. The default transfer method is DMA.

Series 500, Series 800

DMA is the only supported transfer method.

AUTHOR

Io_speed_ctl was developed by HP.

NAME

io_timeout_ctl – establish a time limit for I/O operations

SYNOPSIS

int io_timeout_ctl (eid, time);
int eid;
long time;

DESCRIPTION

Io_timeout_ctl enables you to assign a timeout value to the specified entity id. *Eid* is an entity identifier of an open HP-IB raw bus or GPIO device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Time* is a 32-bit integer value specifying the length of the timeout in microseconds.

This timeout applies to future read and write requests on this entity id. If a read or write request does not complete within the specified time limit, the request is aborted and returns an error indication. If an operation is aborted due to a timeout, *errno*(2) is set to **ETIMEOUT.**

Although the timeout value is specified in microseconds, the resolution of the timeout is system-dependent. For example, a particular system might have a resolution of 10 milliseconds, in which case the specified timeout value is rounded up to the next 10 msec boundary. A timeout value of zero means that the system never causes a timeout. When a file is opened, a zero timeout value is assigned by default.

Entity ids for the same device file obtained by separate *open*(2) requests have their own timeout values associated with them. Entity ids for the same device file obtained by *dup*(2) or inherited by a *fork*(2) request share the same timeout value. In the latter case, if one process changes the timeout, the new timeout is in effect for all such entity ids.

RETURN VALUE

Io_timeout_ctl returns 0 (zero) if successful, or -1 if an error was encountered.

ERRORS

Io_timeout_ctl fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to a channel device file.

HARDWARE DEPENDENCIES

Series 200, Series 300

System timeout resolution is 20 msec.

Series 500

The *errno* value for a timed-out request is **EIO,** specifying that a timeout has occurred. An *errinfo* value of **56** is returned.

Series 800

Timeouts are associated with a given device. Therefore, multiple opens of the same device share timeouts.

AUTHOR

Io_timeout_ctl was developed by HP.

NAME

io_width_ctl – set width of data path

SYNOPSIS

int io_width_ctl (eid, width)
int eid, width;

DESCRIPTION

Io_width_ctl enables you to select the width of the data path to be used for a particular interface. *Eid* is an entity identifier of an open device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Width* is an integer specifying the width of the data path in bits.

An error is given if an invalid width is specified. Specifying a width with this function sets the width for all users of the device file associated with the given entity id. When first opened, the default width is 8 bits.

For the GPIO interface only widths of 8 and 16 bits are currently supported. For the HP-IB interface only a width of 8 bits is supported.

RETURN VALUE

Io_width_ctl returns 0 if successful, and –1 if an error was encountered.

ERRORS

Io_width_ctl fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to a channel device file.
[EINVAL]	the specified *width* is not supported on this device file.

AUTHOR

Io_width_ctl was developed by HP.

NAME

l3tol, ltol3 – convert between 3-byte integers and long integers

SYNOPSIS

void l3tol (lp, cp, n)
long *lp;
char *cp;
int n;

void ltol3 (cp, lp, n)
char *cp;
long *lp;
int n;

DESCRIPTION

L3tol converts a list of *n* three-byte integers packed into a character string pointed to by *cp* into a list of long integers pointed to by *lp*.

Ltol3 performs the reverse conversion from long integers (*lp*) to three-byte integers (*cp*).

These functions are useful for file-system maintenance where the block numbers are three bytes long.

SEE ALSO

fs(4).

BUGS

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent.

NAME

langinfo, langtoid, idtolang, currlangid – information on user's native language as given by NLS

SYNOPSIS

#include <langinfo.h>

char *langinfo(langid, item)
int langid, item;

int langtoid(langname)
char *langname;

char *idtolang(langid)
int langid;

int currlangid()

DESCRIPTION

Langinfo retrieves a null-terminated string containing information unique to a language or cultural area. For example *langinfo(currlangid(),* DAY_1 *)* returns a pointer to the string "Dom" if LANG (see *environ*(5)) is set to "portuguese", and "sun" if LANG is set to "finnish". The following *items* have been defined.

D_T_FMT	string for formatting *date*(1)
DAY_1	Name of the first day of the week ("Sunday" in English)
. . .	
. . .	
DAY_7	Name of the seventh day of the week
ABDAY_1	Abbreviated name of the first day of the week ("Sun" in English)
. . .	
. . .	
ABDAY_7	Abbreviated name of the seventh day of the week
MON_1	Name of the first month in the Gregorian year
. . .	
. . .	
MON_12	Name of the twelfth month
ABMON_1	Abbreviated name of the first month
. . .	
. . .	
ABMON_12	Abbreviated name of the twelfth month
RADIXCHAR	radix character ("decimal point" in English)
THOUSEP	separator for thousands
YESSTR	affirmative response for yes/no questions
NOSTR	negative response for yes/no questions
CRNCYSTR	symbol for currency preceded by '-' if it precedes the number, '+' if it follows the number. For example, "-DM" would be used for German, "+ Kr" for Danish.

Currlangid looks for a LANG string in the user's environment. If it finds it, it returns the corresponding integer listed in *langid*(5). Otherwise it returns 0 to indicate a default to native-computer, the method used before Native Language Support (**NLS**) was available.

Idtolang takes the integer *langid* and attempts to return the corresponding character string defined in *langid*(5). If *langid* is not found, an empty string is returned.

Langtoid is the reverse, trying to convert a string to a language ID, and returning 0 to indicate native-computer if a match cannot be found.

AUTHOR

Langinfo was developed by the Hewlett-Packard Company.

SEE ALSO

getenv(3C), environ(5), hpnls(5), langid(5).

BUGS

Langinfo returns a pointer to a static area which is overwritten on each call.

INTERNATIONAL SUPPORT

8-bit data, messages.

NAME

logname – return login name of user

SYNOPSIS

char *logname( )

DESCRIPTION

Logname returns a pointer to the null-terminated login name; it extracts the **$LOGNAME** variable from the user's environment.

This routine is kept in **/lib/libPW.a**.

FILES

/etc/profile

SEE ALSO

env(1), login(1), profile(4), environ(5).

BUGS

The return values point to static data whose content is overwritten by each call.

This method of determining a login name is subject to forgery.

NAME

lsearch, lfind – linear search and update

SYNOPSIS

```
#include <stdio.h>
#include <search.h>

char *lsearch ((char *)key, (char *)base, nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );

char *lfind ((char *)key, (char *)base, nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );
```

DESCRIPTION

Lsearch is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where a datum may be found. If the datum does not occur, it is added at the end of the table.

Key	points to the datum to be sought in the table.
Base	points to the first element in the table.
Nelp	points to an integer containing the current number of elements in the table. The integer is incremented if the datum is added to the table.
Compar	is the name of the comparison function which the user must supply (*strcmp*, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

Lfind is the same as *lsearch* except that if the datum is not found, it is not added to the table. Instead, a NULL pointer is returned.

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

EXAMPLE

This fragment will read in ≤ TABSIZE strings of length ≤ ELSIZE and store them in a table, eliminating duplicates.

```
#include <stdio.h>

#define TABSIZE 50
#define ELSIZE 120

        char line[ELSIZE], tab[TABSIZE][ELSIZE], *lsearch( );
        unsigned nel = 0;
        int strcmp( );
        . . .
        while (fgets(line, ELSIZE, stdin) != NULL &&
           nel < TABSIZE)
                (void) lsearch(line, (char *)tab, &nel,
                        ELSIZE, strcmp);
          . . .
```

SEE ALSO

bsearch(3C), hsearch(3C), tsearch(3C).

DIAGNOSTICS

If the searched for datum is found, both *lsearch* and *lfind* return a pointer to it. Otherwise, *lfind* returns NULL and *lsearch* returns a pointer to the newly added element.

BUGS

Undefined results can occur if there is not enough room in the table to add a new item.

NAME

malloc, free, realloc, calloc – main memory allocator

SYNOPSIS

char *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;

DESCRIPTION

Malloc and *free* provide a simple general-purpose memory allocation package. *Malloc* returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to *free* is a pointer to a block previously allocated by *malloc*; after *free* is performed this space is made available for further allocation, but its contents are left undisturbed.

Undefined results will occur if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

Malloc allocates the first big enough contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent free blocks as it searches. It calls *sbrk* (see *brk*(2)) to get more memory from the system when there is no suitable space already free.

Realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If no free block of *size* bytes is available in the storage arena, then *realloc* will ask *malloc* to enlarge the arena by *size* bytes and will then move the data to the new space.

Realloc also works if *ptr* points to a block freed since the last call of *malloc*, *realloc*, or *calloc*; thus sequences of *free*, *malloc* and *realloc* can exploit the search strategy of *malloc* to do storage compaction.

Calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

SEE ALSO

brk(2), malloc(3X).

DIAGNOSTICS

Malloc, *realloc* and *calloc* return a NULL pointer if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. When this happens the block pointed to by *ptr* may be destroyed.

BUGS

Free does not check its pointer argument for validity. When passed a null pointer (value 0), it causes a memory fault.

NOTE

Search time increases when many objects have been allocated; that is, if a program allocates but never frees, then each successive allocation takes longer. For an alternate, more flexible implementation, see *malloc*(3X).

NAME

malloc, free, realloc, calloc, mallopt, mallinfo – fast main memory allocator

SYNOPSIS

#include <malloc.h>

char *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;

int mallopt (cmd, value)
int cmd, value;

struct mallinfo mallinfo (max)

DESCRIPTION

Malloc and *free* provide a simple general-purpose memory allocation package, which runs considerably faster than the *malloc*(3C) package. It is found in the library "malloc", and is loaded if the option "–lmalloc" is used with *cc*(1) or *ld*(1).

Malloc returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to *free* is a pointer to a block previously allocated by *malloc*; after *free* is performed this space is made available for further allocation, and its contents will usually have been destroyed (but see *mallopt* below for a way to change this behavior).

Undefined results will occur if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

Realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

Calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

Mallopt provides for control over the allocation algorithm. The available values for *cmd* are: M_MXFAST Set *maxfast* to *value*. The algorithm allocates all blocks below the size of *maxfast* in large groups and then doles them out very quickly. The default value for *maxfast* is 24. M_NLBLKS Set *numlblks* to *value*. The above mentioned "large groups" each contain *numlblks* blocks. *Numlblks* must be greater than 1. The default value for *numlblks* is 100. M_GRAIN Set *grain* to *value*. The sizes of all blocks smaller than *maxfast* are considered to be rounded up to the nearest multiple of *grain*. *Grain* must be greater than 0. The default value of *grain* is the smallest number of bytes which will allow alignment of any data type. Value will be rounded up to a multiple of the default when *grain* is set. M_KEEP Preserve data in a freed block until the next *malloc*, *realloc*, or *calloc*. This option is provided only for compatibility with the old version of *malloc* and is not recommended.

These values are defined in the <*malloc.h*> header file.

Mallopt may be called repeatedly, but may not be called after the first small block is allocated.

Mallinfo provides instrumentation describing space usage, but may not be called until the first small block is allocated. The *max* argument to mallinfo should always be specified as 0 for compatibility with other systems. It returns the structure:

```
struct mallinfo {
        int arena;          /* total space in arena */
        int ordblks;        /* number of ordinary blocks */
        int smblks;         /* number of small blocks */
        int hblkhd;         /* space in holding block headers */
        int hblks;          /* number of holding blocks */
        int usmblks;        /* space in small blocks in use */
        int fsmblks;        /* space in free small blocks */
        int uordblks;       /* space in ordinary blocks in use */
        int fordblks;       /* space in free ordinary blocks */
        int keepcost;       /* space penalty if keep option */
                            /* is used */
}
```

This structure is defined in the <*malloc.h*> header file.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

SEE ALSO

brk(2), malloc(3C).

DIAGNOSTICS

Malloc, realloc and *calloc* return a NULL pointer if there is not enough available memory. When *realloc* returns NULL, the block pointed to by *ptr* is left intact. If *mallopt* is called after any allocation of a small block or if *cmd* or *value* are invalid, non-zero is returned. Otherwise, it returns zero.

WARNINGS

This package usually uses more data space than *malloc*(3C).
The code size is also bigger than *malloc*(3C).
Note that unlike *malloc*(3C), this package does not preserve the contents of a block when it is freed, unless the M_KEEP option of *mallopt* is used.
Undocumented features of *malloc*(3C) have not been duplicated.

NAME

matherr – error-handling function

SYNOPSIS

#include <math.h>

int matherr (x)
struct exception *x;

DESCRIPTION

Matherr is invoked by functions in the Math Library when errors are detected. Users may define their own procedures for handling errors, by including a function named *matherr* in their programs. *Matherr* must be of the form described above. When an error occurs, a pointer to the exception structure *x* will be passed to the user-supplied *matherr* function. This structure, which is defined in the *<math.h>* header file, is as follows:

```
struct exception {
        int type;
        char *name;
        double arg1, arg2, retval;
};
```

The element *type* is an integer describing the type of error that has occurred, from the following list of constants (defined in the header file):

DOMAIN	argument domain error
SING	argument singularity
OVERFLOW	overflow range error
UNDERFLOW	underflow range error
TLOSS	total loss of significance
PLOSS	partial loss of significance

The element *name* points to a string containing the name of the function that incurred the error. The variables *arg1* and *arg2* are the arguments with which the function was invoked. *Retval* is set to the default value that will be returned by the function unless the user's *matherr* sets it to a different value.

If the user's *matherr* function returns non-zero, no error message will be printed, and *errno* will not be set.

If *matherr* is not supplied by the user, the default error-handling procedures, described with the math functions involved, will be invoked upon error. These procedures are also summarized in the table below. In every case, *errno* is set to EDOM or ERANGE and the program continues.

EXAMPLE

```
#include <math.h>

int
matherr(x)
register struct exception *x;
{
        switch (x->type) {
        case DOMAIN:
                /* change sqrt to return sqrt(-arg1), not 0 */
                if (!strcmp(x->name, "sqrt")) {
                        x->retval = sqrt(-x->arg1);
                        return (0); /* print message and set errno */
                }
        case SING:
                /* all other domain or sing errors, print message and abort */
                fprintf(stderr, "domain error in %s\n", x->name);
                abort( );
        case PLOSS:
                /* print detailed error message */
                fprintf(stderr, "loss of significance in %s(%g) = %g\n",
                        x->name, x->arg1, x->retval);
                return (1); /* take no other action */
        }
        return (0); /* all other errors, execute default procedure */
}
```

DEFAULT ERROR HANDLING PROCEDURES

	Types of Errors					
type	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS	PLOSS
errno	EDOM	EDOM	ERANGE	ERANGE	ERANGE	ERANGE
BESSEL:	–	–	–	–	M, 0	*
y0, y1, yn (arg ≤ 0)	M, -H	–	–	–	–	–
EXP:	–	–	H	0	–	–
LOG, LOG10:						
(arg < 0)	M, -H	–	–	–	–	–
(arg = 0)	–	M, -H	–	–	–	–
POW:	–	–	±H	0	–	–
neg ** non-int	M, 0	–	–	–	–	–
0 ** non-pos						
SQRT:	M, 0	–	–	–	–	–
GAMMA:	–	M, H	H	–	–	–
HYPOT:	–	–	H	–	–	–
SINH:	–	–	±H	–	–	–
COSH:	–	–	H	–	–	–
SIN, COS, TAN:	–	–	–	–	M, 0	*
ASIN, ACOS, ATAN2:	M, 0	–	–	–	–	–

ABBREVIATIONS

*	As much as possible of the value is returned.
M	Message is printed (EDOM error).
H	HUGE is returned.
-H	-HUGE is returned.
±H	HUGE or -HUGE is returned.
0	0 is returned.

NAME

memccpy, memchr, memcmp, memcpy, memset – memory operations

SYNOPSIS

```
#include <memory.h>

char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;

char *memchr (s, c, n)
char *s;
int c, n;

int memcmp (s1, s2, n)
char *s1, *s2;
int n;

char *memcpy (s1, s2, n)
char *s1, *s2;
int n;

char *memset (s, c, n)
char *s;
int c, n;
```

DESCRIPTION

These functions operate as efficiently as possible on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

Memccpy copies characters from memory area **s2** into **s1**, stopping after the first occurrence of character **c** has been copied, or after **n** characters have been copied, whichever comes first. It returns a pointer to the character after the copy of **c** in **s1**, or a NULL pointer if **c** was not found in the first **n** characters of **s2**.

Memchr returns a pointer to the first occurrence of character **c** in the first **n** characters of memory area **s**, or a NULL pointer if **c** does not occur.

Memcmp compares its arguments, looking at the first **n** characters only, and returns an integer less than, equal to, or greater than 0, according as **s1** is lexicographically less than, equal to, or greater than **s2**. (*n* less than or equal to zero yields equality). This routine uses **unsigned char** for character comparison on HP-UX. This may not be true for other implementations.

Memcpy copies **n** characters from memory area **s2** to **s1**. It returns **s1**.

Memset sets the first **n** characters in memory area **s** to the value of character **c**. It returns **s**.

NOTE

For user convenience, all these functions are declared in the optional <*memory.h*> header file.

BUGS

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

NAME

mktemp – make a unique file name

SYNOPSIS

char *mktemp (template)
char *template;

DESCRIPTION

Mktemp replaces the contents of the string pointed to by *template* by a unique file name, and returns the address of *template*. The string in *template* should look like a file name with six trailing **X**s; *mktemp* will replace the **X**s with a letter and the current process ID. The letter will be chosen so that the resulting name does not duplicate the name of an existing file. If there are less than 6 **X**s, the letter will be dropped first, and then high order digits of the process ID will be dropped.

RETURN VALUE

Mktemp returns its argument except when it runs out of letters, in which case the result is a pointer to the empty string "".

SEE ALSO

getpid(2).

SEE ALSO

getpid(2), tmpfile(3S), tmpnam(3S).

BUGS

It is possible to run out of letters.

Mktemp does not check to see if the file name part of *template* exceeds the maximum length of a file name.

NAME

monitor – prepare execution profile

SYNOPSIS

#include <mon.h>

void monitor (lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)( ), (*highpc)( );
WORD *buffer;
int bufsize, nfunc;

DESCRIPTION

An executable program created by **cc −p** automatically includes calls for *monitor* with default parameters; *monitor* need not be called explicitly except to gain fine control over profiling.

Monitor is an interface to *profil*(2). *Lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user supplied) array of *bufsize* WORDs (defined in the <*mon.h*> header file). *Monitor* arranges to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. The lowest address sampled is that of *lowpc* and the highest is just below *highpc*. *Lowpc* may not equal 0 for this use of *monitor*. At most *nfunc* call counts can be kept; only calls of functions compiled with the profiling option **−p** of *cc*(1) are recorded. (The C Library and Math Library supplied when **cc −p** is used also have call counts recorded.)

For results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern etext;
...
monitor ((int (*)())2, ((int(*)())& etext, buf, bufsize, nfunc);
```

Etext lies just above all the program text; see *end*(3C).

To stop execution monitoring and write the results on the file **mon.out**, use

```
monitor ((int (*)())0, (int(*)())0, 0, 0, 0);
```

Prof(1) can then be used to examine the results.

FILES

/lib/libp/libc.a
/lib/libp/libm.a
mon.out

SEE ALSO

cc(1), prof(1), profil(2), end(3C).

NAME

nl_toupper, nl_tolower – translate characters for use with NLS

SYNOPSIS

int nl_toupper (c, langid)
int c, langid;

int nl_tolower (c, langid)
int c, langid;

DESCRIPTION

These routines are extensions of their counterparts in *conv*(3C). They function in the same way, but have a second parameter whose value is expected to be one of the values defined in *langid*(5). If *langid* is not one of these legal values, or if shift information for *langid* has not been installed, they function as *toupper()* and *tolower()*.

AUTHOR

NL_conv was developed by the Hewlett-Packard Company.

SEE ALSO

conv(3C), ascii(5), hpnls(5), kana8(5), langid(5), roman8(5).

INTERNATIONAL SUPPORT

8-bit data

NAME

nl_isalpha, nl_isupper, nl_islower, nl_isalnum, nl_ispunct, nl_isprint, nl_isgraph – classify characters for use with NLS

SYNOPSIS

#include <nl_ctype.h>

int nl_isalpha (c, langid)
int c; int langid;

. . .

DESCRIPTION

These routines classify character-coded integer values by table lookup. *Langid* is as defined in *langid*(5). Each is a predicate returning nonzero for true, zero for false. All are defined for the range -1 to 255. If *langid* is not defined, or if type information for that language is not installed, *isalpha, isupper,* etc. will be used, returning 0 for values above octal 0200.

nl_isalpha	*c* is a letter.
nl_isupper	*c* is an upper-case letter.
nl_islower	*c* is a lower-case letter.
nl_isalnum	*c* is an alphanumeric (letter or digit).
nl_ispunct	*c* is a punctuation character (neither control nor alphanumeric).
nl_isprint	*c* is a printing character.
nl_isgraph	*c* is a printing character, like *nl_isprint* except false for space.

DIAGNOSTICS

If the argument to any of these is not in the domain of the function, the result is undefined.

AUTHOR

Nl_ctype was developed by the Hewlett-Packard Company.

SEE ALSO

ctype(3C), stdio(3S), ascii(5), hpnls(5) kana8(5), roman8(5).

INTERNATIONAL SUPPORT

8-bit data

NAME

strcmp8, strncmp8, strcmp16, strncmp16 – non-ASCII string collation

SYNOPSIS

```
int strcmp8 (s1, s2, langid, status)
unsigned char *s1, *s2;
int langid,*status;

int strncmp8 (s1, s2, n, langid, status)
unsigned char *s1, *s2;
int n, langid, *status;

int strcmp16 (s1, s2, file_name, status)
unsigned char *s1, *s2, *file_name;
int *status;

int strncmp16 (s1, s2, n, file_name, status)
unsigned char *s1, *s2, *file_name;
int n, *status;
```

DESCRIPTION

These functions do not check for overflow of any receiving string.

Strcmp8 compares string *s1* and *s2* according to the collating sequence specified by *langid* (See *langid*(5)). An integer greater than, equal to, or less than 0 is returned, depending on whether *s1* is, respectively, greater than, equal to, or less than *s2*. If *langid* or the collation sequence file is not installed, the native machine collating sequence is used. Trailing blanks in string s1 or s2 are ignored. *Strncmp8* makes the same comparison but looks at *n* characters at most.

Strcmp16 compares strings *s1* and *s2* and returns an integer greater than, equal to, or less than 0 depending on whether *s1* is, respectively, greater than, equal to, or less than *s2*. Strings *s1* and *s2* may contain 16-bit characters mixed with 7-bit and 8-bit characters (See *hpnls*(5)). Strings *s1* and *s2* are compared with 8-bit characters collating before 16-bit characters. *Strncmp16* makes the same comparison but looks at *n* characters at most.

Langinit (See *nl_tools_16*(3C)) must be called before the first call to *strcmp16* or *strncmp16*.

ERRORS

The integer pointed to by *status* is set to one of the following non-zero values defined in *usr/include/langinfo.h* if an error condition is encountered. For ENOCFFILE and ENOLFILE, *errno* indicates that a file system call failed.

[ENOCFFILE]	access of the file */usr/lib/nls/config* has failed.
[ENOCONV]	the entry for the language sought is not in the file */usr/lib/nls/config.*
[ENOLFILE]	access of the data file */usr/lib/nls/LANG/collate8* or *file_name* has failed (where *LANG* is the language name associated with the value of *langid*).

WARNINGS

The current versions of *strcmp16* and *strncmp16* do not support a collation sequence table (a null string should be passed as *file_name* to maintain the correct argument count).

AUTHOR

Nl_string was developed by HP.

SEE ALSO

nl_tools_16(3C), col_seq_8(4), hpnls(5), langid(5).

NAME

nl_tools_16 – tools to process 16-bit characters

SYNOPSIS

```
int langinit(langname)
char *langname;

int firstof2(c)
int c;

int secof2(c)
int c;

int byte_status(c, laststatus)
int c, laststatus;

#include <nl_ctype.h>

FIRSTof2(c)
int c;

SECof2(c)
int c;

BYTE_STATUS(c, laststatus)
int c, laststatus;

CHARAT(p)
char *p;

ADVANCE(p)
char *p;

CHARADV(p)
char *p;

PCHAR(c, p)
int c;
char *p;

PCHARADV(c, p)
int c;
char *p;
```

DESCRIPTION

Langinit initializes a table according to the specified language name. This table is used by the other 16-bit tools described herein to determine whether a byte may be the first or second byte of a 16-bit character. This same table is also used by the 8-bit *nl_ctype*(3C) routines for character classification. The *nl_ctype*(3C) routines implicitly call *langinit* if the table has not yet been loaded or if the language specified is different from the language currently loaded in the table. The 16-bit tools do not automatically call *langinit*: you must explicitly call *langinit* as appropriate.

The argument to *langinit*, which is *langname*, must be a pointer to a null terminated string containing a language name as defined in *langid*(5). If *langname* is NULL or points to a zero-length string, *langname* defaults to "n-computer". *Langinit* returns zero if the table for the specified language is loaded without error. If the table for the specified language cannot be loaded, *langinit* loads the table with "n-computer" language data and returns a non-zero value.

FIRSTof2 takes a byte and returns a non-zero value if it may be the first byte of a two-byte character according to the currently loaded *langinit* table, and zero if it is not.

SECof2 takes a byte and returns a non-zero value if it may be the second byte of a two-byte character according to the currently loaded *langinit* table, and zero if it is not.

BYTE_STATUS returns one of the following values based on the value of the byte and the status of the (presumably) last byte passed in as a parameter. These are the status values as defined in <nl_ctype.h>:

ONEBYTE	single byte character
SECOF2	second byte of 2-byte
FIRSTOF2	first byte of 2-byte

Note that in order to validate a two-byte character, both the first and second bytes must be judged individually to be valid. If the value of *laststatus* is FIRSTOF2 but *SECof2(c)* returns false, *BYTE_STATUS(c, laststatus)* will return ONEBYTE.

For the macros *FIRSTof2*, *SECof2*, and *BYTE_STATUS* results are undefined for values of c less than zero or greater than 255.

CHARAT takes as an argument a pointer "p", which is assumed to be pointing at either a one-byte character or the first byte of a two-byte character. In either case it returns the value of the character; analogous to "*p".

ADVANCE advances its pointer argument by the width of the character it is pointing at (either one or two bytes); analogous to "p++".

CHARADV combines the functions of *CHARAT* and *ADVANCE* in a single subroutine that returns a character and advances a pointer argument beyond the last byte of the character; analogous to "*p++".

PCHAR places one (c<256) or two (c>255) bytes of its integer argument, more significant byte first, at the byte location specified by the pointer argument; analogous to "*p = c".

PCHARADV places one (c<256) or two (c>255) bytes of its integer argument, more significant byte first, at the byte location specified by the pointer argument, and advances the pointer past the last byte; analogous to "*p++ = c".

Note that *PCHAR* and *PCHARADV* should not be considered "replace_char" macros. For example, they take no steps to ensure that the second byte of a two-byte character is not left dangling if they over-write the first byte with a single-byte character.

CHARAT, *ADVANCE*, and *CHARADV* examine the byte following the location pointed to by the argument in order to check that it is a valid *SECof2* byte. If it is not a valid *SECof2* byte, the preceding byte will always be treated as a single-byte character.

The functions *firstof2()*, *secof2()*, and *byte_status()*, are subroutine versions of the corresponding macros, and can be called from languages other than C.

AUTHOR

NL_tools_16 was developed by HP.

SEE ALSO

langinfo(3C), nl_ctype(3C), hpnls(5), langid(5).

NAME

nlist – get entries from name list

SYNOPSIS

#include <nlist.h>

int nlist (file-name, nl)
char *file-name;
struct nlist *nl;

REMARKS

The use of symbol table type and value information is inherently non-portable. Use of *nlist* should reduce the effort required to port a program which uses such information, but complete portability across all implementations of HP–UX cannot be expected.

DESCRIPTION

Nlist examines the name list in the executable file whose name is pointed to by *file-name*, and selectively extracts a list of values and puts them in the array of *nlist* structures pointed to by *nl*. The array of *nlist* structures initially contains only the names of variables. Once *nlist* has been called, the variable names are augmented with types and values. The list is terminated by a null name, which consists of a null string in the variable name position of the structure. The name list of the file is searched for each variable name. If the name is found, type and value information from the file is inserted into the name list structure. If the name is not found, type and value fields are set to zero. The structure **nlist** is defined in the include file **<nlist.h>**. See *a.out*(4) and *nlist*(4) for further description of the symbol table structure.

The file must have the organization and symbol table described for an a.out file in *a.out*(4). The information is extracted from the symbol table used by the loader, *ld*(1).

On machines which have such a file, this subroutine is useful for examining the system name list kept in the file **/hp-ux**. In this way programs can obtain system addresses that are up to date.

RETURNS

All nlist structure fields are set to 0 if the file cannot be found or if it is not a valid object file containing a linker symbol table.

Nlist returns –1 upon error; otherwise it returns 0.

NOTES

The <*nlist.h*> header file is automatically included by <*a.out.h*> for compatibility. However, if the only information needed from <*a.out.h*> is for use of *nlist*, then including <*a.out.h*> is discouraged. If <*a.out.h*> is included, the line "#undef n_name" may need to follow it.

SEE ALSO

a.out(4), nlist(4).

NAME

perror, errno, sys_errlist, sys_nerr – system error messages

SYNOPSIS

```
void perror (s)
char *s;

extern int errno;

extern char *sys_errlist[ ];

extern int sys_nerr;
```

DESCRIPTION

Perror produces a message on the standard error output, describing the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a new-line. To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur but not cleared when non-erroneous calls are made.

If given a null-string, the function perror prints only the message and a new-line.

To simplify variant formatting of messages, the array of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the new-line. *Sys_nerr* is the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

If the user's *LANG* shell variable is set, *perror* also attempts to return a translation of the error message.

HARDWARE DEPENDENCIES

Series 500:

The error indicator *errinfo* is implemented in addition to *errno*, enabling you to obtain a more detailed description of the error.

SEE ALSO

errno(2).

NAME

popen, pclose – initiate pipe I/O to/from a process

SYNOPSIS

#include <stdio.h>

FILE *popen (command, type)
char *command, *type;

int pclose (stream)
FILE *stream;

DESCRIPTION

The arguments to *popen* are pointers to null-terminated strings containing, respectively, a shell command line and an I/O mode, either **r** for reading or **w** for writing. *Popen* creates a pipe between the calling program and the command to be executed. The value returned is a stream pointer such that one can write to the standard input of the command, if the I/O mode is **w**, by writing to the file *stream*; and one can read from the standard output of the command, if the I/O mode is **r**, by reading from the file *stream*.

A stream opened by *popen* should be closed by *pclose*, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type **r** command may be used as an input filter and a type **w** as an output filter.

SEE ALSO

pipe(2), wait(2), fclose(3S), fopen(3S), system(3S).

DIAGNOSTICS

Popen returns a NULL pointer if files or processes cannot be created, or if the shell cannot be accessed.

Pclose returns -1 if *stream* is not associated with a "*popen*ed" command.

BUGS

If the original and "*popen*ed" processes concurrently read or write a common file, neither should use buffered I/O, because the buffering gets all mixed up. Problems with an output filter may be forestalled by careful buffer flushing, e.g. with *fflush*; see *fclose*(3S).

NAME

printf, fprintf, sprintf – print formatted output

SYNOPSIS

#include <stdio.h>

int printf (format [, arg] ...)
char *format;

int fprintf (stream, format [, arg] ...)
FILE *stream;
char *format;

int sprintf (s, format [, arg] ...)
char *s, format;

DESCRIPTION

Printf places output on the standard output stream *stdout*. *Fprintf* places output on the named output *stream*. *Sprintf* places "output", followed by the null character (**\0**), in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (not including the **\0** in the case of *sprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its *args* under control of the *format*. The *format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching of zero or more *args*. The results are undefined if there are insufficient *args* for the format. If the format is exhausted while *args* remain, the excess *args* are simply ignored.

Each conversion specification is introduced by the character **%**. After the **%**, the following appear in sequence:

> Zero or more *flags*, which modify the meaning of the conversion specification.
>
> An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag '–', described below, has been given) to the field width. If the field width for an s conversion is preceded by a 0, the string is right adjusted with zero-padding on the left.
>
> A *precision* that gives the minimum number of digits to appear for the **d**, **o**, **u**, **x**, or **X** conversions, the number of digits to appear after the decimal point for the **e** and **f** conversions, the maximum number of significant digits for the **g** conversion, or the maximum number of characters to be printed from a string in **s** conversion. The precision takes the form of a period (**.**) followed by a decimal digit string; a null digit string is treated as zero.
>
> An optional l (ell) specifying that a following **d**, **o**, **u**, **x**, or **X** conversion character applies to a long integer *arg*, or an optional **h** specifying that a following **d**, **o**, **u**, **x**, or **X** conversion character applies to a short integer *arg*. A l before any other conversion character is ignored.
>
> A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*) instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *args* specifying field width or precision must appear *before* the *arg* (if any) to be converted.

The flag characters and their meanings are:

–	The result of the conversion will be left-justified within the field.
+	The result of a signed conversion will always begin with a sign (+ or –).
blank	If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.
#	This flag specifies that the value is to be converted to an "alternate form." For **c**, **d**, **s**, and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x or X** conversion, a non-zero result will have **0x or 0X** prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

d,o,u,x,X	The integer *arg* is converted to signed decimal, unsigned octal, decimal, or hexadecimal notation (**x** and **X**), respectively; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. (For compatibility with older versions, padding with leading zeroes may alternatively be specified by prepending a zero to the field width. This does not imply an octal value for the field width.) The default precision is 1. The result of converting a zero value with a precision of zero is a null string.
f	The float or double *arg* is converted to decimal notation in the style "[–]ddd.ddd", where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, six digits are output; if the precision is explicitly 0, no decimal point appears.
e,E	The float or double *arg* is converted in the style "[–]d.ddde±ddd", where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, six digits are produced; if the precision is zero, no decimal point appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits.
g,G	The float or double *arg* is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** will be used only if the exponent resulting from the conversion is less than –4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.
c	The character *arg* is printed.
s	The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (**\0**) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. A NULL value for *arg* will yield undefined results.
%	Print a **%**; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *printf* and *fprintf* are printed as if *putc*(3S) had been called.

EXAMPLES

To print a date and time in the form "Sunday, July 3, 10:02", where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %d:%.2d", weekday, month, day, hour, min);
```

To print π to 5 decimal places:

```
printf("pi = %.5f", 4 * atan(1.0));
```

SEE ALSO

ecvt(3C), putc(3S), scanf(3S), stdio(3S).

NAME

printmsg, fprintmsg, sprintmsg – print formatted output with numbered arguments

SYNOPSIS

#include <stdio.h>

int printmsg (format [, arg] ...)
char *format;

int fprintmsg (stream, format [, arg] ...)
FILE *stream;
char *format;

int sprintmsg (s, format [, arg] ...)
char *s, format;

DESCRIPTION

Printmsg , *fprintmsg* , and *sprintmsg* are derived from their counterparts in *printf*(3S), with the amplification that the conversion character **%** is replaced by the sequence **%*digit*$** . *Digit* is a decimal digit *n* in the range 1-9, and indicates that this conversion should be applied to the *n*th argument, rather than to the next unused one. All other aspects of formatting are unchanged. All conversion specifications must contain the **%*digit*$** sequence, and it is the user's responsibility to make sure the numbering is correct. All parameters must be used exactly once.

EXAMPLE

To create a language independent date and time printing routine we would write

```
printmsg(format, weekday, month, day, hour, min);
```

For American usage *format* would be a pointer to the string

```
"%1$s, %2$s %3$d, %4$d:%5$.2d"
```

and for German usage to a string

```
"%1$s, %3$d %2$s %4$d:%5$.2d"
```

the resulting outputs will be "Sunday, July 3, 10:02", and "Sonntag, 3 Juli 10:02" , assuming that the proper strings have been passed in.

AUTHOR

Printmsg was developed by the Hewlett-Packard Company.

SEE ALSO

getmsg(3C), printf(3S), hpnls(5).

INTERNATIONAL SUPPORT

8-bit data, messages.

NAME

putc, putchar, fputc, putw – put character or word on a stream

SYNOPSIS

```
#include <stdio.h>

int putc (c, stream)
int c;
FILE *stream;

int putchar (c)
int c;

int fputc (c, stream)
int c;
FILE *stream;

int putw (w, stream)
int w;
FILE *stream;
```

DESCRIPTION

Putc writes the character *c* onto the output *stream* (at the position where the file pointer, if defined, is pointing). *Putchar*(*c*) is defined as *putc*(*c, stdout*). *Putc* and *putchar* are macros.

Fputc behaves like *putc*, but is a function rather than a macro; it may therefore be used as an argument. *Fputc* runs more slowly than *putc*, but it takes less space per invocation and its name can be passed as an argument to a function.

Putw writes the word (i.e., **int** in C) *w* to the output *stream* (at the position at which the file pointer, if defined, is pointing). The size of a word is the size of an integer and varies from machine to machine. *Putw* neither assumes nor causes special alignment in the file.

Output streams, with the exception of the standard error stream *stderr*, are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream *stderr* is by default unbuffered, but use of *freopen* (see *fopen*(3S)) will cause it to become buffered or line-buffered. When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block. When it is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested). *Fflush* can also be used to explicitly write the buffer. *Setbuf*(3S) or *setvbuf* (on *setbuf*(3S)) may be used to change the stream's buffering strategy.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), getc(3S), fread(3S), printf(3S), puts(3S), setbuf(3S).

DIAGNOSTICS

On success, these functions each return the value they have written. On failure, they return the constant **EOF**. This will occur if the file *stream* is not open for writing or if the output file cannot be grown. Because **EOF** is a valid integer, *ferror*(3S) should be used to detect *putw* errors.

Line buffering may cause confusion or malfunctioning of programs which use standard I/O routines but use *read*(2) themselves to read from standard input. In cases where a large amount of computation is done after printing part of a line on an output terminal, it is necessary to *fflush* (on *fclose*(3S)) the standard output before going off and computing so that the output will appear.

BUGS

Because it is implemented as a macro, *putc* treats incorrectly a *stream* argument with side effects. In particular, **putc(c, *f++);** doesn't work sensibly. *Fputc* should be used instead.

Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be read using *getw* on a different processor.

NAME

putenv – change or add value to environment

SYNOPSIS

int putenv (string)
char *string;

DESCRIPTION

String points to a string of the form *"name=value." Putenv* makes the value of the environment variable *name* equal to *value* by altering an existing variable or creating a new one. In either case, the string pointed to by *string* becomes part of the environment, so altering the string will change the environment. The space used by *string* is no longer used once a new string-defining *name* is passed to *putenv*.

DIAGNOSTICS

Putenv returns non-zero if it was unable to obtain enough space via *malloc* for an expanded environment, otherwise zero.

SEE ALSO

exec(2), getenv(3C), malloc(3C), environ(5).

WARNINGS

Putenv manipulates the environment pointed to by *environ*, and can be used in conjunction with *getenv*. However, *envp* (the third argument to *main*) is not changed.
This routine uses *malloc*(3C) to enlarge the environment.
After *putenv* is called, environmental variables are not in alphabetical order.
A potential error is to call *putenv* with an automatic variable as the argument, then exit the calling function while *string* is still part of the environment.

NAME

putpwent – write password file entry

SYNOPSIS

#include <pwd.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;

DESCRIPTION

Putpwent is the inverse of *getpwent*(3C). Given a pointer to a *passwd* structure as created by *getpwent* (or *getpwuid* or *getpwnam*), *putpwent* writes a line on the stream *f*, which matches the format of **/etc/passwd**.

DIAGNOSTICS

Putpwent returns non-zero if an error was detected during its operation, otherwise zero.

SEE ALSO

getpwent(3C).

NAME

puts, fputs – put a string on a stream

SYNOPSIS

```
#include <stdio.h>

int puts (s)
char *s;

int fputs (s, stream)
char *s;
FILE *stream;
```

DESCRIPTION

Puts writes the null-terminated string pointed to by *s*, followed by a new-line character, to the standard output stream *stdout.*

Fputs writes the null-terminated string pointed to by *s* to the named output *stream*.

Neither function writes the terminating null character. Note that *puts* appends a new-line character, but *fputs* does not.

DIAGNOSTICS

Both routines return **EOF** on error. This will happen if the routines try to write on a file that has not been opened for writing.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), printf(3S), putc(3S).

NOTES

Puts appends a new-line character while *fputs* does not.

NAME

qsort – quicker sort

SYNOPSIS

void qsort ((char *) base, nel, sizeof (*base), compar)
unsigned nel;
int (*compar)( );

DESCRIPTION

Qsort is an implementation of the quicker-sort algorithm. It sorts a table of data in place.

Base points to the element at the base of the table. *Nel* is the number of elements in the table. *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function passed as *compar* must return an integer less than, equal to, or greater than zero as a consequence of whether its first argument is to be considered less than, equal to, or greater than the second. This is the same return convention that *strcmp* uses.

NOTES

The pointer to the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
The order in the output of two items which compare as equal is unpredictable.

SEE ALSO

sort(1), bsearch(3C), lsearch(3C), string(3C).

BUGS

If *width* is zero, a divide-by-zero error may be generated.

NAME

rand, srand – simple random-number generator

SYNOPSIS

int rand ( )

void srand (seed)
unsigned seed;

DESCRIPTION

Rand uses a multiplicative congruential random-number generator with period 2^{32} that returns successive pseudo-random numbers in the range from 0 to $2^{15}-1$.

Srand can be called at any time to reset the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

NOTE

The spectral properties of *rand* leave a great deal to be desired. *Drand48*(3C) provides a much better, though more elaborate, random-number generator.

SEE ALSO

drand48(3C).

NAME

regcmp, regex – compile and execute regular expression

SYNOPSIS

char *regcmp (string1 [, string2, ...], (char *)0)
char *string1, *string2, ...;

char *regex (re, subject[, ret0, ...])
char *re, *subject, *ret0, ...;

extern char *__loc1;

DESCRIPTION

Regcmp compiles a regular expression and returns a pointer to the compiled form. *Malloc*(3C) is used to create space for the vector. It is the user's responsibility to free unneeded space so allocated. A NULL return from *regcmp* indicates an incorrect argument. *Regcmp*(1) has been written to generally preclude the need for this routine at execution time.

Regex executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. *Regex* returns NULL on failure or a pointer to the next unmatched character on success. A global character pointer *__loc1* points to where the match began. *Regcmp* and *regex* were mostly borrowed from the editor, *ed*(1); however, the syntax and semantics have been changed slightly. The following are the valid symbols and their associated meanings.

[ ] * . ^	These symbols retain their current meaning.
$	Matches the end of the string; **\n** matches a new-line.
–	Within brackets the minus means *through*. For example, [**a–z**] is equivalent to [**abcd...xyz**]. The – can appear as itself only if used as the first or last character. For example, the character class expression []–] matches the characters] and –.
+	A regular expression followed by + means *one or more times*. For example, [**0–9**]+ is equivalent to [**0–9**][**0–9**]*.
{m} {m,} {m,u}	Integer values enclosed in { } indicate the number of times the preceding regular expression is to be applied. The value *m* is the minimum number and *u* is a number, less than 256, which is the maximum. If only *m* is present (e.g., {m}), it indicates the exact number of times the regular expression is to be applied. The value {m,} is analogous to {m,infinity}. The plus (+) and star (*) operations are equivalent to {1,} and {0,} respectively.
(...)**$***n*	The value of the enclosed regular expression is to be returned. The value will be stored in the *(n+1)*th argument following the subject argument. At most ten enclosed regular expressions are allowed. *Regex* makes its assignments unconditionally.
(...)	Parentheses are used for grouping. An operator, e.g., *, +, { }, can work on a single character or a regular expression enclosed in parentheses. For example, (a*(cb+)*)$0.

By necessity, all the above defined symbols are special. They must, therefore, be escaped to be used as themselves.

EXAMPLES

Example 1:

```
        char *cursor, *newcursor, *ptr;
                ...
        newcursor = regex((ptr = regcmp("^\n", 0)), cursor);
        free(ptr);
```

This example will match a leading new-line in the subject string pointed at by cursor.

Example 2:

```
        char ret0[9];
        char *newcursor, *name;
                ...
        name = regcmp("([A–Za–z][A–za–z0–9_]{0,7})$0", 0);
        newcursor = regex(name, "123Testing321", ret0);
```

This example will match through the string "Testing3" and will return the address of the character after the last matched character (cursor+11). The string "Testing3" will be copied to the character array *ret0*.

Example 3:

```
        #include "file.i"
        char *string, *newcursor;
                ...
        newcursor = regex(name, string);
```

This example applies a precompiled regular expression in **file.i** (see *regcmp*(1)) against *string*.

This routine is kept in **/lib/libPW.a**.

SEE ALSO

ed(1), regcmp(1), malloc(3C).

BUGS

The user program may run out of memory if *regcmp* is called iteratively without freeing the vectors no longer required. The following user-supplied replacement for *malloc*(3C) reuses the same vector saving time and space:

```
        /* user's program */
                ...
        char *
        malloc(n)
        unsigned n;
        {
                static char rebuf[512];
                return (n <= sizeof rebuf) ? rebuf : NULL;
        }
```

NAME

scanf, fscanf, sscanf – formatted input conversion, read from stream file

SYNOPSIS

#include <stdio.h>

int scanf (format [, pointer] ...)
char *format;

int fscanf (stream, format [, pointer] ...)
FILE *stream;
char *format;

int sscanf (s, format [, pointer] ...)
char *s, *format;

DESCRIPTION

Scanf reads from the standard input stream *stdin*. *Fscanf* reads from the named input *stream*. *Sscanf* reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format* described below, and a set of *pointer* arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. White-space characters (blanks, tabs, new-lines, or form-feeds) which, except in two cases described below, cause input to be read up to the next non-white-space character.
2. An ordinary character (not **%**), which must match the next character of the input stream.
3. Conversion specifications, consisting of the character **%**, an optional assignment suppressing character *, an optional numerical maximum field width, an optional **l** (ell) or **h** indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. The suppression of assignment provides a way of describing an input field which is to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted. For all descriptors except "[" and "c", white space leading an input field is ignored.

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument is given. The following conversion codes are legal:

%	a single **%** is expected in the input at this point; no assignment is done.
d	a decimal integer is expected; the corresponding argument should be an integer pointer.
u	an unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.
o	an octal integer is expected; the corresponding argument should be an integer pointer.
x	a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

e,f,g a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or an **e**, followed by an optional +, –, or space, followed by an integer.

s a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating **\0**, which will be added automatically. The input field is terminated by a white-space character. Note that *scanf* will not read a null string.

c a character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use **%1s**. If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.

[indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, which we will call the *scanset,* and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. The circumflex (ˆ), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters *not* contained in the remainder of the scanset string. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct *first–last*, thus [0123456789] may be expressed [0–9]. Using this convention, *first* must be lexically less than or equal to *last*, or else the dash will stand for itself. The dash will also stand for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it will not be syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating **\0**, which will be added automatically. At least one character must match for this conversion to be considered successful.

The conversion characters **d**, **u**, **o**, and **x** may be preceded by **l** or **h** to indicate that a pointer to **long** or to **short** rather than to **int** is in the argument list. Similarly, the conversion characters **e**, **f**, and **g** may be preceded by **l** to indicate that a pointer to **double** rather than to **float** is in the argument list. The **l** or **h** modifier is ignored for other conversion characters.

Scanf conversion terminates at **EOF**, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

Scanf returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, **EOF** is returned.

EXAMPLES

The call:

```
int i, n; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E−1 thompson
```

will assign to *n* the value **3**, to *i* the value **25**, to *x* the value **5.432**, and *name* will contain **thompson\0**. Or:

```
int i; float x; char name[50];
(void) scanf("%2d%f%*d %[0-9]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign **56** to *i*, **789.0** to *x*, skip **0123**, and place the string **56\0** in *name*. The next call to *getchar* (see *getc*(3S)) will return **a**.

SEE ALSO

getc(3S), printf(3S), strtod(3C), strtol(3C).

NOTE

Trailing white space (including a new-line) is left unread unless matched in the control string.

DIAGNOSTICS

These functions return **EOF** on end of input and a short count for missing or illegal data items.

BUGS

The success of literal matches and suppressed assignments is not directly determinable.

NAME

setbuf, setvbuf – assign buffering to a stream file

SYNOPSIS

#include <stdio.h>

void setbuf (stream, buf)
FILE *stream;
char *buf;

int setvbuf (stream, buf, type, size)
FILE *stream;
char *buf;
int type, size;

DESCRIPTION

Setbuf may be used after a stream has been opened but before it is read or written. It causes the array pointed to by *buf* to be used instead of an automatically allocated buffer. If *buf* is the NULL pointer input/output will be completely unbuffered.

A constant **BUFSIZ,** defined in the **<stdio.h>** header file, tells how big an array is needed:

```
char buf[BUFSIZ];
```

Setvbuf may be used after a stream has been opened but before it is read or written. *Type* determines how *stream* will be buffered. Legal values for *type* (defined in stdio.h) are:

_IOFBF	causes input/output to be fully buffered.
_IOLBF	causes output to be line buffered; the buffer will be flushed when a newline is written, the buffer is full, or input is requested.
_IONBF	causes input/output to be completely unbuffered.

If *buf* is not the **NULL** pointer, the array it points to will be used for buffering, instead of an automatically allocated buffer (from *malloc*). *Size* specifies the size of the buffer to be used. The constant **BUFSIZ** in **<stdio.h>** is suggested as a good buffer size. If input/output is unbuffered, *buf* and *size* are ignored.

By default, output to a terminal is line buffered and all other input/output is fully buffered.

HARDWARE DEPENDENCIES

Series 500:

The system call *memallc*(2) is used instead of *malloc.*

SEE ALSO

fopen(3S), getc(3S), malloc(3C), putc(3S), stdio(3S).

DIAGNOSTICS

If an illegal value for *type* or *size* is provided, *setvbuf* returns a non-zero value. Otherwise, the value returned will be zero.

NOTE

A common source of error is allocating buffer space as an "automatic" variable in a code block, and then failing to close the stream in the same block.

NAME

setjmp, longjmp – non-local goto

SYNOPSIS

#include <setjmp.h>

int setjmp (env)
jmp_buf env;

void longjmp (env, val)
jmp_buf env;
int val;

int _setjmp(env)
jmp_buf env;

void _longjmp(env, val)
jmp_buf env;
int val;

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in *env* (whose type, *jmp_buf*, is defined in the *<setjmp.h>* header file) for later use by *longjmp*. It returns the value 0.

Longjmp restores the environment saved by the last call of *setjmp* with the corresponding *env* argument. After *longjmp* is completed, program execution continues as if the corresponding call of *setjmp* (which must not itself have returned in the interim) had just returned the value *val*. *Longjmp* cannot cause *setjmp* to return the value 0. If *longjmp* is invoked with a second argument of 0, *setjmp* will return 1. All accessible data have values as of the time *longjmp* was called.

Upon the return from a *setjmp* call caused by a *longjmp*, the values of any non-static local variables belonging to the routine from which setjmp was called are undefined. Code which depends on such values is not guaranteed to be portable.

Setjmp and *longjmp* save and restore the signal mask (see *sigvector*(2)), while *_setjmp* and *_longjmp* manipulate only the stack and registers. This distinction is only significant for programs which use *sigvector*(2), *sigblock*(2), and/or *sigsetmask*(2).

If a *longjmp* is executed and the environment in which the *setjmp* was executed no longer exists, errors can occur. The conditions under which the environment of the *setjmp* no longer exists include: exiting the procedure which contains the *setjmp* call, and exiting an inner block with temporary storage (e.g. a block with declarations in C, a *with* statement in Pascal). This condition may or may not be detectable. An attempt is made by determining if the stack frame pointer in *env* points to a location not in the currently active stack. If this is the case, *longjmp* will return a -1. Otherwise, the *longjmp* will occur, and if the environment no longer exists, the contents of the temporary storage of an inner block are unpredictable. This condition may also cause unexpected process termination. If the procedure has been exited the results are unpredictable.

Passing *longjmp* a pointer to a buffer not created by *setjmp*, or a buffer that has been modified by the user, can cause all the problems listed above, and more.

Some implementations of Pascal support a *try/recover* mechanism, which also creates stack marker information. If a *longjmp* operation occurs in a scope which is nested inside a try/recover, and the corresponding *setjmp* is not inside the scope of the try/recover, the recover block will not be executed and the currently active recover block will become the one enclosing the *setjmp* (if there is one).

NOTES

A call to *longjmp* to leave the guaranteed stack space reserved by *sigspace*(2) may remove the guarantee that the ordinary execution of the program will not extend into the guaranteed space. It may also cause the program to forever loose its ability to automatically increase the stack size, and the program may then be limited to the guaranteed space.

The result of using *setjmp* within an expression may be unpredictable.

SEE ALSO

sigblock(2), signal(2), sigsetmask(2), sigspace(2), sigvector(2).

WARNING

If *longjmp* is called even though *env* was never primed by a call to *setjmp*, or when the last such call was in a function which has since returned, absolute chaos is guaranteed.

NAME

sinh, cosh, tanh – hyperbolic functions

SYNOPSIS

#include <math.h>

double sinh (x)
double x;

double cosh (x)
double x;

double tanh (x)
double x;

DESCRIPTION

Sinh, *cosh*, and *tanh* return respectively the hyberbolic sine, cosine and tangent of their argument.

ERRORS

Sinh and *cosh* return **HUGE** (and *sinh* may return –**HUGE** for negative *x*) when the correct value would overflow and set *errno* to **ERANGE.**

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M).

NAME

sleep – suspend execution for interval

SYNOPSIS

unsigned long sleep (seconds)
unsigned long seconds;

DESCRIPTION

The current process is suspended from execution for the number of *seconds* specified by the argument. The actual suspension time may be less than that requested for two reasons: (1) Because scheduled wakeups occur at fixed 1-second intervals, (on the second, according to an internal clock) and (2) because any caught signal will terminate the *sleep* following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system. The value returned by *sleep* will be the "unslept" amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested *sleep* time, or premature arousal due to another caught signal.

The routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling *sleep*. If the *sleep* time exceeds the time till such alarm signal, the process sleeps only until the alarm signal would have occurred. The caller's alarm catch routine is executed just before the *sleep* routine returns. If the *sleep* time is less than the time till such alarm, the prior alarm time is reset to go off at the same time it would have without the intervening *sleep*.

Seconds must be less than 2^{32}.

SEE ALSO

alarm(2), pause(2), signal(2).

NAME

sputl, sgetl – access long integer data in a machine-independent fashion

SYNOPSIS

```
void sputl (value, buffer)
long value;
char *buffer;

long sgetl (buffer)
char *buffer;
```

DESCRIPTION

Sputl takes the four bytes of the long integer *value* and places them in memory starting at the address pointed to by *buffer*. The ordering of the bytes is the same across all machines.

Sgetl retrieves the four bytes in memory starting at the address pointed to by *buffer* and returns the long integer value in the byte ordering of the host machine.

The combination of *sputl* and *sgetl* provides a machine-independent way of storing long numeric data in a file in binary form without conversion to characters.

A program which uses these functions must be loaded with the object-file access routine library **libld.a**.

NAME

ssignal, gsignal – software signals

SYNOPSIS

```
#include <signal.h>

int (*ssignal (sig, action))( )
int sig, (*action)( );

int gsignal (sig)
int sig;
```

DESCRIPTION

Ssignal and *gsignal* implement a software facility similar to *signal*(2). This facility is used by the Standard C Library to enable users to indicate the disposition of error conditions, and is also made available to users for their own purposes.

Software signals made available to users are associated with integers in the inclusive range 1 through 15. A call to *ssignal* associates a procedure, *action*, with the software signal *sig*; the software signal, *sig*, is raised by a call to *gsignal*. Raising a software signal causes the action established for that signal to be *taken*.

The first argument to *ssignal* is a number identifying the type of signal for which an action is to be established. The second argument defines the action; it is either the name of a (user-defined) *action function* or one of the manifest constants **SIG_DFL** (default) or **SIG_IGN** (ignore). *Ssignal* returns the action previously established for that signal type; if no action has been established or the signal number is illegal, *ssignal* returns **SIG_DFL**.

Gsignal raises the signal identified by its argument, *sig*:

> If an action function has been established for *sig*, then that action is reset to **SIG_DFL** and the action function is entered with argument *sig*. *Gsignal* returns the value returned to it by the action function.
>
> If the action for *sig* is **SIG_IGN**, *gsignal* returns the value 1 and takes no other action.
>
> If the action for *sig* is **SIG_DFL**, *gsignal* returns the value 0 and takes no other action.
>
> If *sig* has an illegal value or no action was ever specified for *sig*, *gsignal* returns the value 0 and takes no other action.

SEE ALSO

signal(2).

NOTES

There are some additional signals with numbers outside the range 1 through 15 which are used by the Standard C Library to indicate error conditions. Thus, some signal numbers outside the range 1 through 15 are legal, although their use may interfere with the operation of the Standard C Library.

NAME

stdio – standard buffered input/output stream file package

SYNOPSIS

#include <stdio.h>

FILE *stdin, *stdout, *stderr;

DESCRIPTION

The functions described in the entries of sub-class LIBS of this manual constitute an efficient, user-level I/O buffering scheme. The in-line macros *getc*(3S) and *putc*(3S) handle characters quickly. The macros *getchar* and *putchar*, and the higher-level routines *fgetc*, *fgets*, *fprintf*, *fputc*, *fputs*, *fread*, *fscanf*, *fwrite*, *gets*, *getw*, *printf*, *puts*, *putw*, and *scanf* all use or act as if they use *getc* and *putc*; they can be freely intermixed.

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type **FILE**. *Fopen*(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the <stdio.h> header file and associated with the standard open files:

stdin	standard input file
stdout	standard output file
stderr	standard error file

A constant **NULL** (0) designates a nonexistent pointer.

An integer-constant **EOF** (–1) is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

An integer constant **BUFSIZ** specifies the size of the buffers used by the particular implementation.

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

```
#include <stdio.h>
```

The functions and constants mentioned in the entries of sub-class LIBS of this manual are declared in that header file and need no further declaration. The constants and the following "functions" are implemented as macros (redeclaration of these names is perilous): *getc*, *getchar*, *putc*, *putchar*, *ferror*, *feof*, *clearerr*, and *fileno*.

A constant _NFILE defines the maximum number of open files allowed per process.

SEE ALSO

close(2), lseek(2), open(2), pipe(2), read(2), write(2), ctermid(3S), cuserid(3S), fclose(3S), ferror(3S), fopen(3S), fread(3S), fseek(3S), getc(3S), gets(3S), popen(3S), printf(3S), putc(3S), puts(3S), scanf(3S), setbuf(3S), system(3S), tmpfile(3S), tmpnam(3S), ungetc(3S).

DIAGNOSTICS

Invalid *stream* pointers will usually cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

NAME

ftok – standard interprocess communication package

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(path, id)
char *path;
char id;
```

DESCRIPTION

All interprocess communication facilities require the user to supply a key to be used by the *msgget*(2), *semget*(2), and *shmget*(2) system calls to obtain interprocess communication identifiers. One suggested method for forming a key is to use the *ftok* subroutine described below. Another way to compose keys is to include the project ID in the most significant byte and to use the remaining portion as a sequence number. There are many other ways to form keys, but it is necessary for each system to define standards for forming them. If some standard is not adhered to, it will be possible for unrelated processes to unintentionally interfere with each other's operation. Therefore, it is strongly suggested that the most significant byte of a key in some sense refer to a project so that keys do not conflict across a given system.

Ftok returns a key based on *path* and *id* that is usable in subsequent *msgget*, *semget*, and *shmget* system calls. *Path* must be the path name of an existing file that is accessible to the process. *Id* is a character which uniquely identifies a project. Note that *ftok* will return the same key for linked files when called with the same *id* and that it will return different keys when called with the same file name but different *ids*.

EXAMPLES

The following call to ftok() returns a key associated with the file *myfile* and id 'A':

```
        key_t mykey;

                mykey = ftok ("myfile", 'A');
```

SEE ALSO

intro(2), msgget(2), semget(2), shmget(2).

DIAGNOSTICS

Ftok returns **(key_t) –1** if *path* does not exist or if it is not accessible to the process.

WARNING

If the file whose *path* is passed to *ftok* is removed when keys still refer to the file, future calls to *ftok* with the same *path* and *id* will return an error. If the same file is recreated, then *ftok* is likely to return a different key than it did the original time it was called.

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok – character string operations

SYNOPSIS

```
#include <string.h>

char *strcat (s1, s2)
char *s1, *s2;

char *strncat (s1, s2, n)
char *s1, *s2;
int n;

int strcmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
int n;

char *strcpy (s1, s2)
char *s1, *s2;

char *strncpy (s1, s2, n)
char *s1, *s2;
int n;

int strlen (s)
char *s;

char *strchr (s, c)
char *s;
int c;

char *strrchr (s, c)
char *s;
int c;

char *strpbrk (s1, s2)
char *s1, *s2;

int strspn (s1, s2)
char *s1, *s2;

int strcspn (s1, s2)
char *s1, *s2;

char *strtok (s1, s2)
char *s1, *s2;
```

DESCRIPTION

The arguments **s1, s2** and **s** point to strings (arrays of characters terminated by a null character). The functions *strcat*, *strncat*, *strcpy*, and *strncpy* all alter **s1**. These functions do not check for overflow of the array pointed to by **s1**.

Strcat appends a copy of string **s2** to the end of string **s1**. *Strncat* appends at most **n** characters. It copies less if *s2* is shorter than *n* characters. Each returns a pointer to the null-terminated result (the original value of *s1*).

Strcmp compares its arguments and returns an integer less than, equal to, or greater than 0, according as **s1** is lexicographically less than, equal to, or greater than **s2**. (**NULL** values for *s1* and *s2* are treated the same as pointers to null strings.) *Strncmp* makes the same comparison but looks at at most **n** characters (*n* less than or equal to zero yields equality). Both of these routines use **unsigned char** for character comparison.

Strcpy copies string **s2** to **s1**, stopping after the null character has been copied. *Strncpy* copies exactly **n** characters, truncating **s2** or adding null characters to **s1** if necessary. The result will not be null-terminated if the length of **s2** is **n** or more. Each function returns **s1**. Note that *strncpy* should not be used to copy *n* bytes of an arbitrary structure. If that structure contains a null byte anywhere, *strncpy* will terminate the copy when it encounters the null byte, thus returning less than *n* bytes.

Strlen returns the number of characters in **s**, not including the terminating null character.

Strchr (*strrchr*) returns a pointer to the first (last) occurrence of character **c** in string **s**, or a NULL pointer if **c** does not occur in the string. The null character terminating a string is considered to be part of the string.

Strpbrk returns a pointer to the first occurrence in string **s1** of any character from string **s2**, or a NULL pointer if no character from **s2** exists in **s1**.

Strspn (*strcspn*) returns the length of the initial segment of string **s1** which consists entirely of characters from (not from) string **s2**.

Strtok considers the string **s1** to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string **s2**. The first call (with pointer **s1** specified) returns a pointer to the first character of the first token, and will have written a null character into **s1** immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument a NULL pointer) will work through the string **s1** immediately following that token. In this way subsequent calls will work through the string **s1** until no tokens remain. The separator string **s2** may be different from call to call. When no token remains in **s1**, a NULL pointer is returned.

NOTE

For user convenience, all these functions are declared in the optional <*string.h*> header file.

BUGS

The copy operations cannot check for overflow of any receiving string. **NULL** destinations cause errors; **NULL** sources are treated as zero-length strings.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

NAME

strtod, atof, nl_strtod, nl_atof – convert string to double-precision number

SYNOPSIS

double strtod (str, ptr)
char *str, **ptr;

double atof (str)
char *str;

double nl_strtod (str, ptr, langid)
char *str, **ptr;
int langid;

double nl_atof (str, langid)
char *str;
inl langid;

DESCRIPTION

Strtod returns as a double-precision floating-point number the value represented by the character string pointed to by *str*. The string is scanned up to the first unrecognized character.

Strtod recognizes an optional string of "white-space" characters (as defined by *isspace* in *ctype*(3C)), then an optional sign, then a string of digits optionally containing a decimal point, then an optional **e** or **E** followed by an optional sign or space, followed by an integer.

If the value of *ptr* is not (char **)NULL, the variable to which it points is set to point at the character after the last number, if any, that was recognized. If no number can be formed, **ptr* is set to *str*, and zero is returned.

Atof(str) is equivalent to *strtod* (str, (char **)NULL).

NL_strtod and *nl_atof* are similar to the above routines, but use *langid* to determine what the radix character should be, e.g. '.' or ','. If *langid* is not valid, or information for *langid* has not been installed, the radix character defaults to a period.

SEE ALSO

ctype(3C), scanf(3S), strtol(3C), hpnls(5), langid(5).

DIAGNOSTICS

If the correct value would cause overflow, ±HUGE is returned (according to the sign of the value), and *errno* is set to ERANGE. **HUGE_VAL** may be used instead of **HUGE.**
If the correct value would cause underflow, zero is returned and *errno* is set to ERANGE.

AUTHOR

Strtod was developed by AT&T Laboratories and the Hewlett-Packard Company.

INTERNATIONAL SUPPORT

8-bit data, messages.

NAME

strtol, atol, atoi – convert string to integer

SYNOPSIS

long strtol (str, ptr, base)
char *str, **ptr;
int base;

long atol (str)
char *str;

int atoi (str)
char *str;

DESCRIPTION

Strtol returns as a long integer the value represented by the character string pointed to by *str*. The string is scanned up to the first character inconsistent with the base. Leading "white-space" characters (as defined by *isspace* in *ctype*(3C)) are ignored.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no integer can be formed, that location is set to *str*, and zero is returned.

If *base* is greater than 1 (and not greater than 36), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and "0x" or "0X" is ignored if *base* is 16.

If *base* is zero, the string itself determines the base thusly: After an optional leading sign a leading zero indicates octal conversion, and a leading "0x" or "0X" hexadecimal conversion. Otherwise, decimal conversion is used.

Truncation from long to int can, of course, take place upon assignment or by an explicit cast.

Atol(str) is equivalent to *strtol* (str, (char **)NULL, 10).

Atoi (str) is equivalent to (int) *strtol* (str, (char **)NULL, 10).

SEE ALSO

ctype(3C), strtod(3C), scanf(3S).

BUGS

Overflow conditions are ignored.

NAME

swab – swap bytes

SYNOPSIS

```
void swab (from, to, nbytes)
char *from, *to;
int nbytes;
```

DESCRIPTION

Swab copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between byte-swapped and non-byte-swapped machines. *Nbytes* should be even and non-negative. If *nbytes* is odd and positive *swab* uses *nbytes*–1 instead. If *nbytes* is negative, *swab* does nothing.

NAME

system – issue a shell command

SYNOPSIS

#include <stdio.h>

int system (string)
char *string;

DESCRIPTION

System causes the *string* to be given to *sh*(1) as input, as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

FILES

/bin/sh

SEE ALSO

sh(1), exec(2).

DIAGNOSTICS

System forks to create a child process that in turn exec's **/bin/sh** in order to execute *string*. If the fork or exec fails, *system* returns a negative value and sets *errno*.

NAME

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs – emulate /etc/termcap access routines

SYNOPSIS

```
tgetent(bp, name)
char *bp, *name;

tgetnum(id)
char *id;

tgetflag(id)
char *id;

char *tgetstr(id, area)
char *id, **area;

char *tgoto(cm, destcol, destline)
char *cm;

tputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int (*outc)();
```

DESCRIPTION

The *termcap*(3X) functions extract and use capabilities from the compiled terminal capability data bases (see *terminfo*(4)). They are emulation routines that are provided as a part of the *curses*(3X) library.

Tgetent extracts the compiled entry for terminal *name* into buffers accessible by the programmer. Unlike previous termcap routines, all capability strings (except cursor addressing and padding information) are already compiled and stored internally upon return from *tgetent.* The buffer pointer *bp* is redundant in the emulation, and is ignored. It should not be relied upon to point to meaningful information. *Tgetent* returns -1 if it cannot access the *terminfo* directory, 0 if there is no capability file for *name,* and 1 if all goes well. If a TERMINFO environment variable is set, *tgetent* first looks for **TERMINFO/?/name** (where ? is the first character of *name*), and if that file is not accessible, it looks for **/usr/lib/terminfo/?/name**.

Tgetnum gets the numeric value of capability *id,* returning -1 if it is not given for the terminal. *Tgetnum* is useful only with capabilities having numeric values.

Tgetflag returns 1 if the specified capability is present in the terminal's entry, and 0 if it is not. *Tgetflag* is useful only with capabilities that are boolean in nature (i.e. either present or missing in *terminfo*(4)).

Tgetstr returns a pointer to the string value of capability *id.* In addition, if *area* is not a NULL pointer, *tgetstr* will place the capability in the buffer at *area* and advance the area pointer. The returned string capability is compiled except for cursor addressing and padding information. *Tgetstr* is useful only with capabilities having string values.

Tgoto returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline*. (Programs which call *tgoto* should be sure to turn off the TAB3 bit(s), since *tgoto* may now output a tab. See *termio*(7). Note that programs using *termcap* should in general turn off TAB3 anyway since some terminals use control–I for other functions, such as nondestructive space.) If a % sequence is given which is not understood, then *tgoto* returns OOPS.

Tputs decodes the padding information of the string *cp. Affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable. *Outc* is a routine which is called with each character in turn. The *terminfo* variable **pad_char** should contain a pad character to be used (from the *pc* capability) if a null (^@) is inappropriate.

FILES

/usr/lib/libcurses.a	–lcurses library
/usr/lib/terminfo/?/*	data bases

SEE ALSO

ex(1), terminfo(4), termio(7).

NAME

tmpfile – create a temporary file

SYNOPSIS

#include <stdio.h>

FILE *tmpfile ( )

DESCRIPTION

Tmpfile creates a temporary file using a name generated by *tmpnam*(3S), and returns a corresponding FILE pointer. If the file cannot be opened, an error message is printed using *perror*(3C), and a NULL pointer is returned. The file will automatically be deleted when the process using it terminates. The file is opened for update ("w+").

SEE ALSO

creat(2), unlink(2), mktemp(3C), perror(3C), fopen(3S), tmpnam(3S).

NAME

tmpnam, tempnam – create a name for a temporary file

SYNOPSIS

#include <stdio.h>

char *tmpnam (s)
char *s;

char *tempnam (dir, pfx)
char *dir, *pfx;

DESCRIPTION

These functions generate file names that can safely be used for a temporary file.

Tmpnam always generates a file name using the path-prefix defined as **P_tmpdir** in the *<stdio.h>* header file. If *s* is NULL, *tmpnam* leaves its result in an internal static area and returns a pointer to that area. The next call to *tmpnam* will destroy the contents of the area. If *s* is not NULL, it is assumed to be the address of an array of at least **L_tmpnam** bytes, where **L_tmpnam** is a constant defined in *<stdio.h>*; *tmpnam* places its result in that array and returns *s*.

Tempnam allows the user to control the choice of a directory. The argument *dir* points to the name of the directory in which the file is to be created. If *dir* is NULL or points to a string which is not a name for an appropriate directory, the path-prefix defined as **P_tmpdir** in the *<stdio.h>* header file is used. If that directory is not accessible, **/tmp** will be used as a last resort. This entire sequence can be up-staged by providing an environment variable **TMPDIR** in the user's environment, whose value is the name of the desired temporary-file directory.

Many applications prefer their temporary files to have certain favorite initial letter sequences in their names. Use the *pfx* argument for this. This argument may be NULL or point to a string of up to five characters to be used as the first few characters of the temporary-file name.

Tempnam uses *malloc*(3C) to get space for the constructed file name, and returns a pointer to this area. Thus, any pointer value returned from *tempnam* may serve as an argument to *free* (see *malloc*(3C)). If *tempnam* cannot return the expected result for any reason, i.e. *malloc*(3C) failed, or none of the above mentioned attempts to find an appropriate directory was successful, a NULL pointer will be returned.

NOTES

These functions generate a different file name each time they are called.

Files created using these functions and either *fopen*(3S) or *creat*(2) are temporary only in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use *unlink*(2) to remove the file when its use is ended.

SEE ALSO

creat(2), unlink(2), malloc(3C), mktemp(3C), fopen(3S), tmpfile(3S).

BUGS

If called more than 17,576 times in a single process, these functions will start recycling previously used names.
Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using these functions or *mktemp*, and the file names are chosen so as to render duplication by other means unlikely.

NAME

sin, cos, tan, asin, acos, atan, atan2 – trigonometric functions

SYNOPSIS

```
#include <math.h>

double sin (x)
double x;

double cos (x)
double x;

double tan (x)
double x;

double asin (x)
double x;

double acos (x)
double x;

double atan (x)
double x;

double atan2 (y, x)
double y, x;
```

DESCRIPTION

Sin, *cos* and *tan* return respectively the sine, cosine and tangent of their argument, x, measured in radians.

Asin returns the arcsine of x, in the range $-\pi/2$ to $\pi/2$.

Acos returns the arccosine of x, in the range 0 to π.

Atan returns the arctangent of x, in the range $-\pi/2$ to $\pi/2$.

Atan2 returns the arctangent of y/x, in the range $-\pi$ to π, using the signs of both arguments to determine the quadrant of the return value.

HARDWARE DEPENDENCIES

Series 200, 300, 500:

The approximate limit for the values returned by these functions is 1.49^8.

The algorithms used for all functions except *atan2* are from HP 9000 BASIC.

ERRORS

Sin, *cos*, and *tan* lose accuracy when their argument is far from zero. For arguments sufficiently large, these functions return zero when there would otherwise be a complete loss of significance. In this case a message indicating TLOSS error is printed on the standard error output. For less extreme arguments causing partial loss of significance, a PLOSS error is generated but no message is printed. In both cases, *errno* is set to **ERANGE.**

If the magnitude of the argument of *asin* or *acos* is greater than one, or if both arguments of *atan2* are zero, zero is returned and *errno* is set to **EDOM**. In addition, a message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M).

NAME

tsearch, tfind, tdelete, twalk – manage binary search trees

SYNOPSIS

```
#include <search.h>

char *tsearch ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tfind ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tdelete ((char *) key, (char **) rootp, compar)
int (*compar)( );

void twalk ((char *) root, action)
void (*action)( );
```

DESCRIPTION

Tsearch, tfind, tdelete, and *twalk* are routines for manipulating binary search trees. They are generalized from Knuth (6.2.2) Algorithms T and D. All comparisons are done with a user-supplied routine, *compar*. This routine is called with two arguments, the pointers to the elements being compared. It returns an integer less than, equal to, or greater than 0, according to whether the first argument is to be considered less than, equal to or greater than the second argument. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Tsearch is used to build and access the tree. **Key** is a pointer to a datum to be accessed or stored. If there is a datum in the tree equal to *key (the value pointed to by key), a pointer to this found datum is returned. Otherwise, *key is inserted, and a pointer to it returned. Only pointers are copied, so the calling routine must store the data. **Rootp** points to a variable that points to the root of the tree. A NULL value for the variable pointed to by **rootp** denotes an empty tree; in this case, the variable will be set to point to the datum which will be at the root of the new tree.

Like *tsearch*, *tfind* will search for a datum in the tree, returning a pointer to it if found. However, if it is not found, *tfind* will return a NULL pointer. The arguments for *tfind* are the same as for *tsearch*.

Tdelete deletes a node from a binary search tree. The arguments are the same as for *tsearch*. The variable pointed to by **rootp** will be changed if the deleted node was the root of the tree. *Tdelete* returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

Twalk traverses a binary search tree. **Root** is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) *Action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration data type *typedef enum* { *preorder, postorder, endorder, leaf* } *VISIT;* (defined in the *<search.h>* header file), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character. Similarly, although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

EXAMPLE

The following code reads in strings and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

```
        #include <search.h>
        #include <stdio.h>

        struct node {           /* pointers to these are stored in the tree */
                char *string;
                int length;
        };
        char string_space[10000];       /* space to store strings */
        struct node nodes[500];         /* nodes to store */
        struct node *root = NULL;       /* this points to the root */

        main( )
        {
                char *strptr = string_space;
                struct node *nodeptr = nodes;
                void print_node( ), twalk( );
                int i = 0, node_compare( );

                while (gets(strptr) != NULL && i++ < 500)  {

                        /* set node */
                        nodeptr->string = strptr;
                        nodeptr->length = strlen(strptr);

                        /* put node into the tree */
                        (void) tsearch((char *)nodeptr, &root,
                                node_compare);

                        /* adjust pointers, so we don't overwrite tree */
                        strptr += nodeptr->length + 1;
                        nodeptr++;
                }
                twalk(root, print_node);
        }
        /* This routine compares two nodes, based on an
              alphabetical ordering of the string field.  */
        int
        node_compare(node1, node2)
        struct node *node1, *node2;
        {
                return strcmp(node1->string, node2->string);
        }
        /* This routine prints out a node, the first time
              twalk encounters it.  */
        void
        print_node(node, order, level)
        struct node **node;
        VISIT order;
        int level;
        {
                if (order == preorder || order == leaf)  {
                        (void)printf("string = %20s,  length = %d\n",
                               (*node)->string, (*node)->length);
                 }
          }
```

SEE ALSO

bsearch(3C), hsearch(3C), lsearch(3C).

DIAGNOSTICS

A NULL pointer is returned by *tsearch* if there is not enough space available to create a new node.
A NULL pointer is returned by *tsearch, tfind* and *tdelete* if **rootp** is NULL on entry.
If the datum is found, both *tsearch* and *tfind* return a pointer to it. If not, *tfind* returns NULL, and *tsearch* returns a pointer to the inserted item.

WARNINGS

The **root** argument to *twalk* is one level of indirection less than the **rootp** arguments to *tsearch* and *tdelete*.
There are two nomenclatures used to refer to the order in which tree nodes are visited. *Tsearch* uses preorder, postorder and endorder to respectively refer to visting a node before any of its children, after its left child and before its right, and after both its children. The alternate nomenclature uses preorder, inorder and postorder to refer to the same visits, which could result in some confusion over the meaning of postorder.

BUGS

If the calling function alters the pointer to the root, results are unpredictable.

NAME

ttyname, isatty – find name of a terminal

SYNOPSIS

char *ttyname (fildes)
int fildes;

int isatty (fildes)
int fildes;

DESCRIPTION

Ttyname returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor *fildes*.

Isatty returns 1 if *fildes* is associated with a terminal device, 0 otherwise.

FILES

/dev/* /dev/pty/*

DIAGNOSTICS

Ttyname returns a NULL pointer if *fildes* does not describe a terminal device in directory **/dev**.

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

ttyslot – find the slot in the utmp file of the current user

SYNOPSIS

int ttyslot ( )

DESCRIPTION

Ttyslot returns the index of the current user's entry in the **/etc/utmp** file. This is accomplished by actually scanning the file **/etc/inittab** for the name of the terminal associated with the standard input, the standard output, or the error output (0, 1 or 2).

FILES

/etc/inittab
/etc/utmp

SEE ALSO

getut(3C), ttyname(3C).

DIAGNOSTICS

A value of 0 is returned if an error was encountered while searching for the terminal name or if none of the above file descriptors is associated with a terminal device.

NAME

ungetc – push character back into input stream

SYNOPSIS

#include <stdio.h>

int ungetc (c, stream)
int c;
FILE *stream;

DESCRIPTION

Ungetc inserts the character *c* into the buffer associated with an input *stream*. That character, *c*, will be returned by the next *getc*(3S) call on that *stream*. *Ungetc* returns *c*, and leaves the file *stream* unchanged.

One character of pushback is guaranteed, provided something has already been read from the stream and the stream is actually buffered. In the case that *stream* is *stdin*, one character may be pushed back onto the buffer without a previous read statement.

If *c* equals **EOF**, *ungetc* does nothing to the buffer and returns **EOF**.

Fseek(3S) erases all memory of inserted characters.

SEE ALSO

fseek(3S), getc(3S), setbuf(3S).

DIAGNOSTICS

Ungetc returns **EOF** if it cannot insert the character.

NAME

vprintf, vfprintf, vsprintf – print formatted output of a varargs argument list

SYNOPSIS

```
#include <stdio.h>
#include <varargs.h>

int vprintf (format, ap)
char *format;
va_list ap;

int vfprintf (stream, format, ap)
FILE *stream;
char *format;
va_list ap;

int vsprintf (s, format, ap)
char *s, *format;
va_list ap;
```

DESCRIPTION

vprintf, *vfprintf*, and *vsprintf* are the same as *printf*, *fprintf*, and *sprintf* respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by *varargs*(5).

EXAMPLE

The following demonstrates how *vfprintf* could be used to write an error routine.

```
#include <stdio.h>
#include <varargs.h>
        .
        .
        .
/*
 *      error should be called like
 *              error(function_name, format, arg1, arg2...);
 */
/*VARARGS0*/
void
error(va_alist)
/* Note that the function_name and format arguments cannot be
 *       separately declared because of the definition of varargs.
 */
va_dcl
{
        va_list args;
        char *fmt;

        va_start(args);

        /* print out name of function causing error */
        (void)fprintf(stderr, "ERROR in %s: ", va_arg(args, char *));
        fmt = va_arg(args, char *);

        /* print out remainder of message */
        (void)vfprintf(stderr, fmt, args);
        va_end(args);
```

```
                (void)abort( );
        }
```

SEE ALSO

printf(3S), varargs(5).

NAME

intro – introduction to file formats

DESCRIPTION

This section outlines the formats of various files. The C **struct** declarations for the file formats are given where applicable. Usually, these structures can be found in the directories **/usr/include** or **/usr/include/sys**.

SEE ALSO

hier(5).

The introduction to this manual.

NAME

a.out – assembler and link editor output

REMARKS

There will be separate manual pages for each implementation that wishes to describe its *a.out* format to its customers.

DESCRIPTION

The *a.out* (i.e., object file) format is completely machine-dependent except for the first word, which contains a magic number as defined in *magic*(4).

The archive symbol table is also completely machine dependent except for its name in the archive. See *ar*(4). This page should also describe the format of the archive symbol table.

SEE ALSO

ar(4), magic(4)

NAME

a.out – assembler and link editor output

Remarks:

This manual page describes the *a.out* file format for Series 200 and 300 computers. Refer to other *a.out*(5) manual pages for descriptions of other valid implementations.

DESCRIPTION

A.out is the output file of the link editor *ld*. *Ld* will make **a.out** executable if there were no linking errors and no unresolved external references. The assembler *as* produces non-executable files with the same structure.

File *a.out* has seven sections: a header, the program text and data segments, a pascal interface section, a symbol table, information for debugger support, and text and data relocation information (in that order). The pascal interface text will only be present in those pascal code segments that have not been linked. The last three sections may be missing if the program was linked with the **–s** option of *ld*(1) or if the symbol table, debug information, and relocation bits were removed by *strip*(1). Also note that if there were no unresolved external references after linking, the relocation information will be removed.

The file section containing information for debugger support has three tables – the debug name table (DNTT), the source line table (SLT), and the value table (VT). These tables contain symbolic information used by the HP-UX debugger *cdb*(1). HP-UX compilers create this information under control of the –g option.

When an **a.out** file is loaded into memory for execution, three logical segments are set up: the text segment, the data segment (initialized data followed by uninitialized, the latter actually being initialized to all 0's), and a stack. The text segment begins at location 0x0 in the core image; the header is not loaded. If the magic number (the first field in the header) is EXEC_MAGIC, it indicates that the text segment is not to be write-protected or shared, so the data segment will be contiguous with the text segment. If the magic number is SHARE_MAGIC or DEMAND_MAGIC, the data segment begins at the first 0 mod 0x1000 byte boundary following the text segment, and the text segment is not writable by the program; if other processes are executing the same **a.out** file, they will share a single text segment. If the magic number is DEMAND_MAGIC, the text and data segments are not read in from the file until they are referenced by the program.

The stack will occupy the highest possible locations in the core image and grow downward (the stack is automatically extended as required). The data segment is only extended as requested by the *brk*(2) system call.

The start of the text segment in the **a.out** file is given by the macro TEXT_OFFSET(hdr), where hdr is a copy of the file header. The macro DATA_OFFSET(hdr) provides the starting location of the data segment.

The value of a word in the text or data portions that is not a reference to an undefined external symbol is exactly the value that will appear in memory when the file is executed. If a word in the text or data portion involves a reference to an undefined external symbol, as indicated by the relocation information (discussed below) for that word, then the value of the word as stored in the file is an offset from the associated external symbol. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added to the word in the file.

Header

The format of the **a.out** header for the MC68000 is as follows (segment sizes are in bytes):

```
struct exec {
        MAGIC   a_magic;        /* magic number */
        short   a_stamp;        /* version stamp */
        short   a_unused;
        long    a_sparehp;
        long    a_text;         /* size of text segment */
        long    a_data;         /* size of data segment */
        long    a_bss;          /* size of bss segment */
        long    a_trsize;       /* size of text relocation info */
        long    a_drsize;       /* size of data relocation info */
        long    a_pasint;       /* size of interface text */
        long    a_lesyms;       /* size of symbol table */
        long    a_dnttsize;     /* debug name table size */
        long    a_entry;        /* entry point of program */
        long    a_sltsize;      /* source-line table size */
        long    a_vtsize;       /* value table size */
        long    a_spare3;
        long    a_spare4;
};
```

Pascal Interface Section

The Pascal interface section consists of the ascii representation of the interface text for that Pascal module.

The start of the Pascal interface section is given by the macro MODCAL_OFFSET(hdr).

Symbol Table

The symbol table consists of entries of the form:

```
        struct nlist {
                long                    n_value;
                unsigned char           n_type;
                unsigned char           n_length;
                short                   n_almod;
                short                   n_unused;
        };
```

Following this structure is *n_length* ascii characters which compose the symbol name.

The *n_type* field indicates the type of the symbol; the following values are possible:

00	undefined symbol
01	absolute symbol
02	text segment symbol
03	data segment symbol
04	bss segment symbol

One of these values ANDed with 040 indicates an external symbol. One of these values ANDed with 020 indicates an aligned symbol.

The start of the symbol table is given by the macro LESYM_OFFSET(hdr).

Relocation

If relocation information is present, it amounts to eight bytes per relocatable datum.

The format of the relocation data is:

```
struct  r_info      {
        long        r_address;
        short       r_symbolnum;
        char        r_segment;
        char        r_length;
};
```

The *r_address* field indicates the position of the relocation within the segment.

The *r_segment* field indicates the segment referred to by the text or data word associated with the relocation word:

00	indicates the reference is to the text segment;
01	indicates the reference is to initialized data;
02	indicates the reference is to bss (uninitialized data);
03	indicates the reference is to an undefined external symbol.

The field *r_symbolnum* contains a symbol number in the case of external references, and is unused otherwise. The first symbol is numbered 0, the second 1, etc.

The field *r_length* indicates the length of the datum to be relocated.

00	indicates it is a byte
01	indicates it is a short
02	indicates it is a long
03	indicates it is a special align symbol

The start of the text relocation section is provided by the macro RTEXT_OFFSET(hdr).

The start of the data relocation section is provided by the macro RDATA_OFFSET(hdr).

SEE ALSO

as(1), ld(1), nm(1), strip(1), magic(5).

NAME

a.out – executable linker output file

Remarks:

This manual entry describes the *a.out* file format for the Series 500. Refer to other *a.out* manual pages for information valid for other implementations.

DESCRIPTION

A.out is the output file of the linker *ld*(1). *Ld* makes *a.out* executable if no errors occured during compilation and linking, and there are no remaining unresolved external references.

The *a.out* file has five sections: a file header, a segment table, a segment information section, a symbol table(s) section, and a name pool(s) section, logically arranged as follows:

***a.out* File Structure**

File header
Segment Table
Segment Information: • segment image (code/data) • fix-up information (loader) • relocation information (*ld*)
Symbol Tables: • linker symbol table • information for debugger support
Name Pool (strings)

Note that this pictorial representation depicts the logical structure of the file, but does not necessarily reflect the physical arrangement of data in the file. The following topics discuss each logical section of the *a.out file.*

File Header

The *a.out* file header is conceptually divided into two parts. The first part contains "scalar" values, while the second part contains "file map" data pertaining to the rest of the file. The entire file header contains 128 bytes of information, 32 of which make up the scalar section. The scalar section can be represented pictorially as follows:

Byte Arrangement: File Header Scalar Section

Bytes	Contents	
0 thru 3	System ID (2 bytes)	File Type (2 bytes)
4 thru 7	Reserved for future use	
8 thru 11	Flags	
12 thru 15	Program Entry Point	
16 thru 19	Version Stamp	
20 thru 23	Memory Offset	
24 thru 27	Working Set gurarantee	
28 thru 31	Reserved for future use	

Each group of four bytes forms a word that has a specific function:

Magic Number
: The first word, called the "magic number", is divided into two half-words containing the system ID and the file type. The system ID identifies the target machine upon which the object code can run. The file type specifies whether or not the file is executable (hex 107), shareable (hex 108), or relocatable (hex 106).

Flags
: The third word specifies the settings of five flags. The left-most five bits of this word are significant; the remainder of the word is ignored.

Bit 1	(the left-most of the flag bits) If set, bit 1 marks the program as using a single data segment. You can override this with the **−T** or **−A** *ld* options, which force the program to reside in one or two data segments, respectively.
Bit 2	If set, marks the file as relinkable (meaning that the file contains relocation records and a symbol table).
Bit 3	If set, the file is debuggable.
Bit 5	If set, marks the program as allowing null pointer dereferencing. To set bit 5, use the **−Z** option of *ld*.

Program Entry Point
: This word contains an external program pointer (EPP) that references the starting code for the program. *Ld* normally assigns the starting address of the main program to this word. This can be changed with the **−e** linker option.

Version Stamp
: The *Version Stamp* is a user-supplied 32-bit integer used to distinguish one version of an application program from another. The user can specify this integer by using the **−V** *ld*(1) option at link time.

The file map portion of the header is structured as follows:

Byte Arrangement: File Header File Map Section

Bytes	Contents	Bytes	Contents
32 thru 35	Code Segment Table: offset	80 thru 83	VT: offset
36 thru 39	Code Segment Table: size	84 thru 87	VT: size
40 thru 43	Code Segment Images: offset	88 thru 91	SLT: offset
44 thru 47	Code Segment Images: size	92 thru 95	SLT: size
48 thru 51	Data Segment Table: offset	96 thru 99	Name Pool: offset
52 thru 55	Data Segment Table: size	100 thru 103	Name Pool: size
56 thru 59	Data Segment Images: offset	104 thru 107	Interface Information: offset
60 thru 63	Data Segment Images: size	108 thru 111	Interface Information: size
64 thru 67	Link Symbol Table: offset	112 thru 115	Reserved for future use
68 thru 71	Link Symbol Table: size	116 thru 119	Reserved for future use
72 thru 75	DNTT: offset	120 thru 123	Reserved for future use
76 thru 79	DNTT: size	124 thru 127	Reserved for future use

Each *offset* entry in the file map shows where the given section starts, relative to the beginning of the *a.out* file. Each *size* entry gives the size (in bytes) for that section.

Each *offset* entry in the file map shows where the given section starts, relative to the beginning of the *a.out* file. Each *size* entry gives the size (in bytes) for that section.

DNTT is the Debug Symbol Table, VT is the Value Table, and SLT is the Source Line Table. All are used by the symbolic debugger, *cdb*.

Segment Table

The segment table collects, in one place, all information about the code and data segments making up the program. The segment table consists of an array of entries. Each entry describes one code or data segment of the program.

Both code and data segment table entries include the following information:

segment name	Consists of an offset into the name pool, relative to the beginning of the name pool. Useful for symbolically referring to code or data segments (not currently implemented).
segment type	Specifies one of three possible types of segments: code; direct data (in GDS), or indirect data (in GDS or EDS).
list of *segment attributes*	Segments can be paged, virtual, demand loadable, writable, or privileged. The linker sets the attributes for executable files.
segment offset	References a particular code or data segment within the segment image area. The reference is given relative to the beginning of the segment image area.
segment size	Size (in bytes) of the particular code or data segment being described in the entry.
segment fixup size	Specifies the size (in bytes) of the loader fixup area in the particular segment being described.
segment relocation information size	Specifies the number of bytes of relocation records for this segment.

The following entries appear only in data segment table entries:

segment limit	Specifies the maximum number of bytes that the indirect data segment can contain. Attempting to increase the size beyond this stated limit produces an error. The linker assigns a default value of 1.5 Mbytes to this field, but it can be changed with the **−m** *chatr*(1) option.
segment zero-padding size	A byte count of the uninitialized data area. The linker computes this value from the data relocation records.

The following entries appear only in code segment table entries:

segment local procedures count	Specifies the number of procedures defined in that segment, that are only known locally within the segment.
segment external procedures count	Specifies the number of procedures defined in that segment, that are externally known.

Several words are left unused in each segment table entry to allow for future growth.

Segment Information

This section of the file contains the segment images for each segment included in the final, executable file. This section contains a subsection for each program segment. Each subsection is in turn made up of three parts – the contents of the segment (code or data), a list of pointers that the loader must "fix up" in that segment, and the relocation records for that segment. Each subsection looks as follows:

Segment Information Structure

Code/Data Image
Loader Initialization Information
Loader Fixup Information
Relocation Records

The code image contains the compiled machine code for each program segment. The data image contains an image of initialized data for the program. Contained in this code are pointers. The loader fixup information area contains offsets that reference these pointers (the offsets are given relative to the beginning of the code/data image area). These offsets must be "fixed up" at run time (i.e., the program loader *exec* must update the segment number fields with the correct values). The linker generates the loader fixup information.

Symbol Tables

The linker symbol table contains data on relocatable symbols relevant to the linker (e.g. name and type for each global symbol). Refer to *nm*(1) (Series 500 only version) for a complete description of each symbol type and the parameters associated with them. The contents of the symbol table may be listed in several different ways with *nm*.

Name Pool

The name pool contains a list of null-terminated strings, which specify the names of the symbols in the program. The symbol table entries contain indexes into the name pool instead of the names themselves. This permits arbitrarily long names to be used instead of fixed-length names. The first string in the name pool is always a null string. This enables zero to be used as an index into the name pool for entities which have no names.

SEE ALSO

chatr(1), ld(1), nm(1), strip(1), magic(5).

NAME

a.out – assembler and link editor output

SYNOPSIS

#include <a.out.h>

DESCRIPTION

The file name **a.out** is the output file from the assembler *as*(1), the compilers, and the linker *ld*(1). The assembler and compilers create relocatable object files ready for input to the linker; the linker creates executable object files.

An object file consists of a file header, auxiliary headers, space dictionary, subspace dictionary, symbol table, relocation information, compiler records, space string table, symbol string table, and the data for initialized code and data. Not all of these sections are required for all object files. The file must begin with the file header, but the remaining sections do not have to be in any particular order; the file header contains pointers to each of the other sections of the file.

A relocatable object file, created by the assembler or compiler, must contain at least the following sections: file header, space dictionary, subspace dictionary, symbol table, relocation information, space string table, symbol string table, and code and data. It may also contain auxiliary headers and compiler records. Relocatable files generally contain unresolved symbols; the linker combines relocatable files and searches libraries to produce an executable file. The linker can also be used to combine relocatable files and produce a new relocatable file as output, suitable for input to a subsequent linker run.

An executable file, created by the linker, typically contains the following sections: file header, an HP-UX auxiliary header, space dictionary, subspace dictionary, symbol table, space string table, symbol string table, and code and data. The linker also copies any auxiliary headers and compiler records from the input files to the output file. If the file has been stripped (see *strip*(1)), it will not contain a symbol table, symbol string table, or compiler records. An executable file must not contain any unresolved symbols.

Programs for the Series 800 architecture consist of two loadable spaces: a shared, non-writable, code space named "$TEXT$"; and a private, writable, data space named "$PRIVATE$". A program may contain other non-loadable spaces that contain data needed by development tools; for example, symbolic debugging information is contained in a space named "$DEBUG$". The linker treats loadable and unloadable spaces exactly the same, so the full generality of symbol resolution and relocation is available for the symbolic debugging information. Spaces have an addressing range of 4,294,967,296 (2^32) bytes; each loadable space is divided into four 1,073,741,824 (2^30) byte quadrants. The HP-UX operating system places all code in the first quadrant of the $TEXT$ space, and all data in the second quadrant of the $PRIVATE$ space. (Thus, it is sufficient to use a 32-bit pointer to access any address in either space.)

Each space is also divided into logical units called subspaces. When the linker combines relocatable object files, it groups all subspaces from the input files by name, then arranges the groups within the space by a sort key associated with each subspace. Subspaces are not architecturally significant; they merely provide a mechanism for combining individual parts of spaces independently from many input files. Some typical subspaces in a program are shown in the following table.

$MILLICODE$	Code for millicode routines
LIT	Sharable literals
$CODE$	Code
$UNWIND$	Stack unwind information
$GLOBAL$	Outer block declarations for Pascal
$DATA$	Static initialized data
$COMMON$	FORTRAN common
BSS	Uninitialized data

Subspaces can be initialized or uninitialized (typically, only BSS is uninitialized). The subspace dictionary entry for an initialized subspace contains a file pointer to the initialization data, while the entry for an uninitialized subspace contains only a 32-bit pattern used to initialize the entire area at load time.

In a relocatable file, initialized code and data often contains references to locations elsewhere in the file, and to unresolved symbols defined in other files. These references are patched at link time using the relocation information. Each entry in the relocation information (a "fixup") specifies a location within the initialized data for a subspace, and an expression that defines the actual value that should be placed at that location, relative to one or two symbols.

The linker summarizes the subspace dictionary in the HP-UX auxiliary header when creating an executable file. HP-UX programs contain only three separate sections: one for the code, one for initialized data, and one for uninitialized data. By convention, this auxiliary header is placed immediately following the file header.

When an **a.out** file is loaded into memory for execution, three areas of memory are set up: the code is loaded into the first quadrant of a new, sharable space; the data (initialized followed by uninitialized) is loaded into the second quadrant of a new, private space; and a stack is created beginning at a fixed address near the middle of the second quadrant of the data space.

The file format described here is a common format for all operating systems designed for HP's Precision Architecture. Therefore, there are some fields and structures that are not used on HP-UX or have been reserved for future use.

File Header

The format of the file header is described by the following structure declaration from **<filehdr.h>**.

```
struct header {
      short int system_id;                    /* 0x020b */
      short int a_magic;                      /* magic number */
      unsigned int version_id;                /* a.out format version */
      struct sys_clock {
          unsigned int secs;
          unsigned int nanosecs;
      } file_time;                            /* timestamp */
      unsigned int entry_space;               /* reserved */
      unsigned int entry_subspace;            /* reserved */
      unsigned int entry_offset;              /* reserved */
      unsigned int aux_header_location;       /* file ptr to aux hdrs */
      unsigned int aux_header_size;           /* sizeof aux hdrs */
      unsigned int som_length;                /* length of object module */
      unsigned int presumed_dp;               /* reserved */
      unsigned int space_location;            /* file ptr to space dict */
      unsigned int space_total;               /* # of spaces */
      unsigned int subspace_location;         /* file ptr to subsp dict */
      unsigned int subspace_total;            /* # of subspaces */
      unsigned int loader_fixup_location;     /* reserved */
      unsigned int loader_fixup_total;        /* reserved */
      unsigned int space_strings_location;    /* file ptr to sp. strings */
      unsigned int space_strings_size;        /* sizeof sp. strings */
      unsigned int init_array_location;       /* reserved */
      unsigned int init_array_total;          /* reserved */
      unsigned int compiler_location;         /* file ptr to comp recs */
      unsigned int compiler_total;            /* # of compiler recs */
```

```
        unsigned int symbol_location;          /* file ptr to sym table */
        unsigned int symbol_total;             /* # of symbols */
        unsigned int fixup_request_location;   /* file ptr to fixups */
        unsigned int fixup_request_total;      /* # of fixups */
        unsigned int symbol_strings_location;  /* file ptr to sym strings */
        unsigned int symbol_strings_size;      /* sizeof sym strings */
        unsigned int unloadable_sp_location;   /* file ptr to debug info */
        unsigned int unloadable_sp_size;       /* size of debug info */
        unsigned int checksum;                 /* header checksum */
};
```

Auxiliary Headers

The auxiliary headers are contained in a single contiguous area in the file, and are located by a pointer in the file header. Auxiliary headers are used for two purposes: users can attach version and copyright strings to an object file, and an auxiliary header contains the information needed to load an executable program. In an executable program, the HP-UX auxiliary header must precede all other auxiliary headers. The following declarations are found in **<aouthdr.h>**.

```
struct aux_id {
        unsigned int    mandatory : 1;          /* reserved */
        unsigned int    copy : 1;               /* reserved */
        unsigned int    append : 1;             /* reserved */
        unsigned int    ignore : 1;             /* reserved */
        unsigned int    reserved : 12;          /* reserved */
        unsigned int    type : 16;              /* aux hdr type */
        unsigned int    length;                 /* sizeof rest of aux hdr */
};

/* Values for the aux_id.type field */
#define HPUX_AUX_HEADER 4
#define USER_AUX_HEADER 6

struct som_exec_auxhdr {
        struct aux_id   som_auxhdr;             /* aux header id */
        long            exec_tsize;             /* text size */
        long            exec_tmem;              /* start address of text */
        long            exec_tfile;             /* file ptr to text */
        long            exec_dsize;             /* data size */
        long            exec_dmem;              /* start address of data */
        long            exec_dfile;             /* file ptr to data */
        long            exec_bsize;             /* bss size */
        long            exec_entry;             /* address of entry point */
        long            exec_flags;             /* loader flags */
        long            exec_bfill;             /* bss initialization value */
};

/* Values for exec_flags */
#define TRAP_NIL_PTRS       01

struct user_string_aux_hdr {
        struct aux_id   header_id;              /* aux header id */
        unsigned int    string_length;          /* strlen(user_string) */
        char            user_string[1];         /* user-defined string */
};
```

Space Dictionary

The space dictionary consists of a sequence of space records as defined in **<spacehdr.h>**.

```
struct space_dictionary_record {
        union name_pt  name;                     /* index to space name */
        unsigned int   is_loadable: 1;           /* space is loadable */
        unsigned int   is_defined: 1;            /* space is defined within file */
        unsigned int   is_private: 1;            /* space is not sharable */
        unsigned int   reserved: 13;             /* reserved */
        unsigned int   sort_key: 8;              /* sort key for space */
        unsigned int   reserved2: 8;             /* reserved */
        int            space_number;             /* space index */
        int            subspace_index;           /* index to first subspace */
        unsigned int   subspace_quantity;        /* # of subspaces in space */
        int            loader_fix_index;         /* index to first ldr fixup */
        unsigned int   loader_fix_quantity;      /* # of ldr fixups in space */
        int            init_pointer_index;       /* index to first init ptr */
        unsigned int   init_pointer_quantity;    /* # of init ptrs in space */
};
```

The strings for the space names are contained in the space strings table, which is located by a pointer in the file header. Each entry in the space strings table is preceded by a 4-byte integer that defines the length of the string, and is terminated by at one to five null characters to pad the string out to a word boundary. Indices to this table are relative to the start of the table, and point to the first byte of the string (not the preceding length word). The union defined below is used for all such string pointers; the character pointer is defined for programs that read the string table into memory and wish to relocate in-memory copies of space records.

```
union name_pt {
        char *n_name;
        unsigned int   n_strx;
};
```

Subspace Dictionary

The subspace dictionary consists of a sequence of subspace records as defined in **<scnhdr.h>**. Strings for subspace names are contained in the space strings table.

```
struct subspace_dictionary_record {
        int            space_index;
        unsigned int   access_control_bits: 7;   /* reserved */
        unsigned int   memory_resident: 1;       /* reserved */
        unsigned int   dup_common: 1;            /* COBOL-style common */
        unsigned int   is_common: 1;             /* subspace is a common block */
        unsigned int   is_loadable: 1;           /* subspace is loadable */
        unsigned int   quadrant: 2;              /* reserved */
        unsigned int   initially_frozen: 1;      /* reserved */
        unsigned int   is_first: 1;              /* reserved */
        unsigned int   code_only: 1;             /* subspace contains only code */
        unsigned int   sort_key: 8;              /* subspace sort key */
        unsigned int   reserved: 8;              /* reserved */
        int            file_loc_init_value;      /* file location or init value */
        unsigned int   initialization_length;    /* length of initialization */
        unsigned int   subspace_start;           /* starting offset */
        unsigned int   subspace_length;          /* total subspace length */
        unsigned int   reserved2: 16;            /* reserved */
        unsigned int   alignment: 16;            /* alignment required */
        union name_pt  name;                       /* index of subspace name */
        int            fixup_request_index;       /* index to first fixup */
        unsigned int   fixup_request_quantity;    /* # of fixup requests */
};
```

Symbol Table

The symbol table consists of a sequence of entries described by the structure shown below, from **<syms.h>**. Strings for symbol and qualifier names are contained in the symbol strings table, whose structure is identical with the space strings table.

```
struct symbol_dictionary_record {
        unsigned int   hidden: 1;                 /* reserved */
        unsigned int   symbol_type: 7;            /* symbol type */
        unsigned int   symbol_scope: 4;           /* symbol value */
        unsigned int   check_level: 3;            /* type checking level */
        unsigned int   must_qualify: 1;           /* qualifier required */
        unsigned int   initially_frozen: 1;       /* reserved */
        unsigned int   memory_resident: 1;        /* reserved */
        unsigned int   is_common: 1;              /* common block */
        unsigned int   dup_common: 1;             /* COBOL-style common */
        unsigned int   xleast: 2;                 /* reserved */
        unsigned int   arg_reloc: 10;             /* reserved */
        union name_pt  name;                      /* index to symbol name */
        union name_pt  qualifier_name;            /* index to qual name */
        unsigned int   symbol_info;               /* subspace index */
        unsigned int   symbol_value;              /* symbol value */
};

/* Values for symbol_type */
#define ST_NULL            0        /* unused symbol entry */
#define ST_ABSOLUTE        1        /* non-relocatable symbol */
#define ST_DATA            2        /* data symbol */
#define ST_CODE            3        /* generic code symbol */
#define ST_PRI_PROG        4        /* program entry point */
#define ST_SEC_PROG        5        /* secondary prog entry pt.*/
#define ST_ENTRY           6        /* procedure entry point */
#define ST_STORAGE         7        /* storage request */
#define ST_STUB            8        /* reserved */
#define ST_MODULE          9        /* Pascal module name */
#define ST_SYM_EXT         10       /* symbol extension record */
#define ST_ARG_EXT         11       /* argument extension record */
#define ST_MILLICODE       12       /* millicode entry point */
#define ST_PLABEL          13       /* reserved */
#define ST_OCT_DIS         14       /* reserved */
#define ST_MILLI_EXT       15       /* reserved */

/* Values for symbol_scope */
#define SS_UNSAT           0        /* unsatisfied reference */
#define SS_EXTERNAL        1        /* reserved */
#define SS_LOCAL           2        /* local symbol */
#define SS_UNIVERSAL       3        /* global symbol */
```

The meaning of the symbol value depends on the symbol type. For the code symbols (generic code, program entry points, procedure and millicode entry points), the low-order two bits of the symbol value encode the execution privilege level, which is not used on HP-UX, but is generally set to 3. The symbol value with those bits masked out is the address of the symbol (which is always a multiple of 4). For data symbols, the symbol value is simply the address of the symbol. For storage requests, the symbol value is the number of bytes requested; the linker allocates space for the largest request for each symbol in the BSS subspaces, unless a local or universal symbol is found for that symbol (in which case the storage request is treated like an unsatisfied reference).

If a relocatable file was compiled with parameter type checking, then extension records follow symbols that define and reference procedure entry points and global variables. The first extension record, the *symbol extension record*, defines the type of the return value or global variable, and (if a procedure or function) the number of parameters and the types of the first three parameters. If more parameter type descriptors are needed, one or more *argument extension records* follow, each containing four more descriptors. A check level of 0 specifies no type checking; no extension records follow. A check level of 1 or more specifies checking of the return value or global variable type. A check level of 2 or more specifies checking of the number of parameters, and a check level of 3 specifies checking the types of each individual parameter. The linker performs the requested level of type checking between unsatisfied symbols and local or universal symbols as it resolves symbol references.

```
union arg_descriptor {
      struct {
            unsigned int      reserved: 3;        /* not used */
            unsigned int      packing: 1;         /* reserved */
            unsigned int      alignment: 4;       /* byte alignment */
            unsigned int      reserved2: 1;       /* not used */
            unsigned int      mode: 3;            /* use of symbol */
            unsigned int      structure: 4;       /* structure of symbol */
            unsigned int      hash: 1;            /* set if arg_type is hashed */
            int               arg_type: 15;       /* data type */
      }                 arg_desc;
      unsigned int      word;
};

struct symbol_extension_record {
      unsigned int      type: 8;                  /* always ST_SYM_EXT */
      unsigned int      max_num_args: 8;          /* max # of parameters */
      unsigned int      min_num_args: 8;          /* min # of parameters */
      unsigned int      num_args: 8;              /* actual # of parameters */
      union arg_descriptor symbol_desc;           /* symbol type desc. */
      union arg_descriptor argument_desc[3];      /* first 3 parameters */
};

struct argument_desc_array {
      unsigned int      type: 8;                  /*  always ST_ARG_EXT */
      unsigned int      reserved: 24;             /* not used */
      union arg_descriptor argument_desc[4];      /* next 4 parameters */
};
```

The values for the *alignment*, *mode*, *structure*, and *arg_type* (when the data type is not hashed) fields are given in the following table.

value	alignment	mode	structure	arg_type
0	byte	any	any	any
1	half-word	value parm	scalar	void
2	word	reference parm	array	signed byte
3	dbl word	value-result	struct	unsigned byte
4		name	pointer	signed short
5		variable	long ptr	unsigned short
6	64-byte	function return	C string	signed long
7		procedure	Pascal string	unsigned long
8		long ref parm	procedure	signed dbl word
9			function	unsigned dbl word
10			label	short real
11	page			real
12				long real
13				short complex
14				complex
15				long complex
16				packed decimal
17				struct/array

Relocation Information

Each initialized subspace defines a range of fixups that apply to the data in that subspace. A fixup request is associated with every word that requires relocation or that contains a reference to an unsatisfied symbol. Fixups can compute an expression involving zero, one, or two symbols and a constant, then extract a field of bits from that result, and deposit those bits in any of several different formats (corresponding to the Precision Architecture instruction set). The structure of a fixup request is contained in **<reloc.h>**.

```
struct fixup_request_record {
        unsigned int    need_data_ref: 1;       /* reserved */
        unsigned int    arg_reloc: 10;          /* reserved */
        unsigned int    expression_type: 5;     /* how to compute value */
        unsigned int    exec_level: 2;          /* reserved */
        unsigned int    fixup_format: 6;        /* how to deposit bits */
        unsigned int    fixup_field: 8;         /* field to extract */
        unsigned int    subspace_offset;        /* subspace offset of word */
        unsigned int    symbol_index_one;       /* index of first symbol */
        unsigned int    symbol_index_two;       /* index of second symbol */
        int             fixup_constant;         /* constant */
};

/* Values for expression_type */
#define e_one               0        /* symbol1 + constant */
#define e_two               1        /* symbol1 - symbol2 + constant */
#define e_pcrel             2        /* symbol1 - pc + constant */
#define e_con               3        /* constant */
#define e_subst             4        /* reserved */
#define e_subend            5        /* reserved */
#define e_subrel            6        /* reserved */
#define e_plabel            7        /* symbol1 + constant */

/* Values for fixup_field (assembler mnemonics shown) */
#define e_fsel              0        /* F': no change */
```

```
#define e_lssel           1       /* LS': inverse of RS' */
#define e_rssel           2       /* RS': rightmost 11 bits, signed */
#define e_lsel            3       /* L': leftmost 21 bits */
#define e_rsel            4       /* R': rightmost 11 bits */
#define e_ldsel           5       /* LD': inverse of RD' */
#define e_rdsel           6       /* RD': rightmost 11 bits, filled left with ones */
#define e_lrsel           7       /* LR': L' with "rounded" constant */
#define e_rrsel           8       /* RR': R' with "rounded" constant */

/* Values for fixup_format (typical instructions shown) */
#define i_exp14           0       /* 14-bit immediate (LDW, STW) */
#define i_exp21           1       /* 21-bit immediate (LDIL, ADDIL) */
#define i_exp11           2       /* 11-bit immediate (ADDI, SUBI) */
#define i_rel17           3       /* 17-bit pc-relative (BL) */
#define i_rel12           4       /* 12 bit pc-relative (COMBT, COMBF, etc.) */
#define i_data            5       /* whole word */
#define i_none            6       /* not used */
#define i_abs17           7       /* 17-bit absolute (BE, BLE) */
#define i_milli           8       /* 17-bit millicode call (BLE) */
#define i_break           9       /* reserved (no effect on HP-UX) */
```

Compiler Records

Compiler records are placed in relocatable files by each compiler or assembler to identify the version of the compiler that was used to produce the file. These records are copied into the executable file by the linker, but are strippable. The structure of a compiler record is shown below. All strings are contained in the symbol string table.

```
struct compilation_unit {
      union name_pt  name;
      union name_pt  language_name;
      union name_pt  product_id;
      union name_pt  version_id;
      int            reserved;
      struct sys_clock compile_time;
      struct sys_clock source_time;
};
```

SEE ALSO

as(1), cc(1), ld(1), nm(1), strip(1).

NAME

acct – per-process accounting file format

SYNOPSIS

#include <sys/acct.h>

DESCRIPTION

Files produced as a result of calling *acct*(2) have records in the form defined by **<sys/acct.h>**, whose contents are:

```
typedef ushort comp_t;        /* "floating point": 13-bit fraction, 3-bit exponent  */
struct   acct {
      char       ac_flag;     /* Accounting flag */
      char       ac_stat;     /* Exit status */
      ushort     ac_uid;      /* Accounting user ID */
      ushort     ac_gid;      /* Accounting group ID */
      dev_t      ac_tty;      /* control typewriter */
      time_t     ac_btime;    /* Beginning time */
      comp_t     ac_utime;    /* acctng user time in clock ticks */
      comp_t     ac_stime;    /* acctng system time in clock ticks */
      comp_t     ac_etime;    /* acctng elapsed time in clock ticks */
      comp_t     ac_mem;      /* memory usage in clicks */
      comp_t     ac_io;       /* chars trnsfrd by read/write */
      comp_t     ac_rw;       /* number of block reads/writes */
      char       ac_comm[8];  /* command name */
};
#define AFORK    01       /* has executed fork, but no exec */
#define ASU      02       /* used super-user privileges */
#define ACCTF    0300     /* record type: 00 = acct */
```

In *ac_flag*, the AFORK flag is turned on by each *fork*(2) and turned off by an *exec*(2). The *ac_comm* field is inherited from the parent process and is reset by any *exec*. Each time the system charges the process with a clock tick, it also adds to *ac_mem* the current process size, computed as follows:

(data size) + (text size) + (number of in-core processes sharing text) +
sum of ((shared memory segment size) / (number of in-core processes attached to segment))

For systems with virtual memory, the text, data, and shared memory sizes refer to the resident portion of the memory segments. The value of *ac_mem / (ac_stime + ac_utime)* can be viewed as an approximation to the mean process size, as modified by text-sharing.

The structure **tacct.h**, which resides with the source files of the accounting commands, represents the total accounting format used by the various accounting commands:

```
/*
 * total accounting (for acct period), also for day
 */
struct          tacct {
      uid_t            ta_uid;          /* userid */
      char             ta_name[8];      /* login name */
      float            ta_cpu[2];       /* cum. cpu time, p/np (mins) */
      float            ta_kcore[2];     /* cum kcore-minutes, p/np */
      float            ta_con[2];       /* cum. connect time, p/np, mins */
      float            ta_du;           /* cum. disk usage */
      long             ta_pc;           /* count of processes */
      unsigned short   ta_sc;           /* count of login sessions */
      unsigned short   ta_dc;           /* count of disk samples */
      short            ta_fee;          /* fee for special services */
};
```

HARDWARE DEPENDENCIES

Series 500

Ac_mem includes only certain resident segments still held by a process when it terminates. Because ac_mem does not account for shared or vitual memory, or for changes in the amount of memory allocated dynamically, ac_mem / (ac_stime + ac_utime) may not always furnish a good approximation of memory usage.

SEE ALSO

acct(2), acct(1M), acctcom(1M), exec(2), fork(2).

BUGS

The *ac_mem* value for a short-lived command gives little information about the actual size of the command, because *ac_mem* may be incremented while a different command (e.g., the shell) is being executed by the process.

NAME

ar – common archive file format

SYNOPSIS

#include <ar.h>

DESCRIPTION

Ar(1) is used to concatenate several files into an archival file. Archives are used mainly as libraries to be searched by the link editor *ld*(1).

Each archive begins with the archive magic string.

```
#define ARMAG   "!<arch>\n"          /* magic string */
#define SARMAG  8                    /* length of magic string */
```

Each archive which contains object files (see *a.out*(4)) includes an archive symbol table. This symbol table is used by the link editor *ld*(1) to determine which archive members must be loaded during the link edit process. The archive symbol table (if it exists) is always the first file in the archive (but is never listed) and is automatically created and/or updated by *ar*.

Following the archive magic string are the archive file members. Each file member is preceded by a file member header which is of the following format:

```
#define ARFMAG   "`\n"          /* header trailer string */

struct ar_hdr {                 /* file member header */
   char   ar_name[16];          /* '/' terminated file member name */
   char   ar_date[12];          /* file member date */
   char   ar_uid[6];            /* file member user identification */
   char   ar_gid[6];            /* file member group identification */
   char   ar_mode[8];           /* file member mode (octal) */
   char   ar_size[10];          /* file member size */
   char   ar_fmag[2];           /* header trailer string */
};
```

All information in the file member headers is in printable ASCII. The numeric information contained in the headers is stored as decimal numbers (except for *ar_mode* which is in octal). Thus, if the archive contains printable files, the archive itself is printable.

The *ar_name* field is blank-padded and slash (/) terminated. The *ar_date* field is the modification date of the file at the time of its insertion into the archive. Common format archives can be moved from system to system as long as the portable archive command *ar*(1) is used. Note that older versions or *ar*(1) did not use the common archive format, and those archives cannot be read or written by the common archiver.

Each archive file member begins on an even byte boundary; a newline is inserted between files if necessary. Nevertheless the size given reflects the actual size of the file exclusive of padding.

Notice there is no provision for empty areas in an archive file. If the archive symbol table exists, the first file in the archive has a zero length name (i.e., **ar_name[0]** == '/'). The contents of this archive member are machine dependent. Further details can be found in the *a.out*(4) manual page for each machine.

SEE ALSO

ar(1), ld(1), strip(1), a.out(4), magic(4).

CAVEATS

Strip(1) will remove all archive symbol entries from the header. The archive symbol entries must be restored via the **ts** option of the *ar*(1) command before the archive can be used with the link editor *ld*(1).

NAME

bif – bell interchange format utilities

DESCRIPTION

BIF (Bell Interchange Format) is the name given to the format of mounted media used by HP-UX Series 200, revisions 2.0 and 2.1, and the Integral Personal Computer. This format is based upon that used in System III Unix.

The BIF utilities listed under SEE ALSO below are provided for non-Series 200 (revision 2.0 and 2.1) and non-Integral Personal Computer access to the BIF media. These utilities read and write data to and from BIF volumes, as well as retrieve and store information on BIF volumes.

The BIF utilities listed below are the only HP-UX utilities that recognize the internal contents of a BIF volume. To the rest of HP-UX, a BIF volume is simply a file/disk containing unspecified data. Therefore, do not use *mount*(1) on a BIF volume; the operating system cannot recognize it.

BIF file names are specified to the BIF utilities by concatenating the HP-UX path name for the BIF volume with the BIF file name, separating the two with a colon (:). For example,

/dev/rdsk/1s0:/users/ivy
: specifies BIF file **/users/ivy** within HP-UX device special file **/dev/rdsk/1s0**

Note that this file naming convention is applicable only for use as arguments to the BIF utilities and does not constitute a legal path name for any other use within HP-UX. The shell *sh*(1) "meta" characters * ? and [...] do not work for specifying an arbitrary pattern for file name matching when using the BIF utilities.

If the device name and a trailing colon are specified without a file or directory name following, e.g., **/dev/rdsk/1s0:**, the root (/) of the BIF file system is assumed by convention.

A primitive form of data protection is provided by a lockfile **/tmp/BIF..LCK** that only allows one process and its immediate children to use the BIF utilities at a time.

AUTHOR

Bif was developed by HP.

SEE ALSO

bifchmod(1), bifchown(1), bifcp(1), bifdf(1M), biffind(1), biffsck(1M), biffsdb(1M), bifls(1), bifmkdir(1), bifmkfs(1M), bifrm(1).

NAME

checklist – static information about the file systems

SYNOPSIS

#include <checklist.h>

DESCRIPTION

Checklist is an ASCII file and resides in directory **/etc.** It is only read by programs, and not written; it is the duty of the system administrator to properly create and maintain this file. **/etc/checklist** contains a list of mountable file system entries. The fields within each entry of a file system are separated by one or more blanks. Each file system entry is contained on a separate line. The order of entries in **/etc/checklist** is important because *fsck*, *mount*, and *umount* sequentially iterate through **/etc/checklist.**

Each file system entry must contain a **special file name** and may additionally contain all of the following fields, in order:

block special file name
directory
type
pass number on parallel fsck
backup frequency
comment

These additional fields are ignored in an HP-UX system if the set of system administration tools implemented on that system does not support them:

special file name is either a character or block special file name. This field is used by the *fsck*(1M) command.

block special file name
is used by the *mount*(1M) and other commands.

directory is the name of the root of the mounted file system which corresponds to the block special file name. The *directory* must already exist and must be given as an absolute path name.

type can be "rw", "ro", "sw" or "xx". If *type* is "rw" or "ro" then the file system whose name is given in the *block special file* field is mounted read-write or read-only on the specified *directory* by **mount -a**. If *type* is "sw" then the *special file name* is made available as a piece of swap space by the *swapon*(1M) command. The fields *pass number* and *backup frequency* are ignored for "sw" entries. Entries marked "xx" are ignored by all commands and can be used to mark unused sections. If *type* is specified as either "xx" or "sw" the entry is ignored by the *mount*(1M) command.

pass number field is used by the *fsck*(1M) command to determine the order in which file system checks are done when using the -p option of *fsck*. The root file system should be specified with a *pass number* of 1, and other file systems should have larger numbers. File systems within a drive should have distinct numbers, but file systems on different drives can be checked on the same pass to utilize possible parallelism available in the hardware. A file system with a *pass number* of zero will be ignored by the *fsck*(1M) command. If a pass number is not present, *fsck* will check each such file system sequentially after all eligible file systems with pass numbers have been checked.

backup frequency field is reserved for possible use by future backup utilities.

comment field is an optional field which starts with a pound sign (#) and ends with a newline. Space from the *backup frequency* up to the comment field, if present, or the newline is reserved for future use.

Examples of file system entries specified in **/etc/checklist:**

For system which supports only special file name field:

```
/dev/rdsk/0s0
```

For system which supports multi-fields:

```
/dev/rdsk/0s0  /dev/dsk/0s0  /  rw  1  0  #root disk
```

HARDWARE DEPENDENCIES

Series 500:

All of the optional fields in a file system entry will be ignored.

Series 200, 300, 800:

There is no limit to the number of special file names in */etc/checklist.* However, the commands *mount*-a and *umount*-a give an error if the number of mountable file system entries in */etc/checklist* exceeds NMOUNT.

AUTHOR

Checklist was developed by the Hewlett-Packard Company, AT&T Bell Laboratories, and the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

fsck[HFS](1M), mount[HFS](1M), swapon[HFS](1M), getfsent(3X),

NAME

col_seq_8 – collating sequence table for languages with 8-bit character sets

DESCRIPTION

There are four language dependent collation algorithms for European languages. These algorithms are:

Two_to_one conversions: Some languages such as Spanish require two adjacent characters to occupy one position in the collating sequence. Examples are "CH" (which follows "C") and "LL" (which follows "L").

One_to_two conversions: Some languages such as German require one character (e.g. "sharp S") to occupy two adjacent positions in the collating sequence.

Don't care characters: Some languages designate certain characters to be ignored in character comparisons. For example, if "–" is a "don't care" character, then the strings "REACT" and "RE–ACT" would equal each other when compared.

Case and accent priority: Many languages require a "two pass" collating algorithm: in pass one, the accents are stripped off the letters and the resulting two strings are compared; if they are equal, a second pass with the accents back in place is performed to break the tie. The case of letters may also be used in this fashion.

This table has four sections: a file header, a sequence table, a two_to_one mapping table, and a one_to_two mapping table.

The file header has the following format:

```
        struct header {
             short int    table_len;             /* Table length */
             short int    lang_id;               /* Language id number */
             short int    reserved1;             /* Reserved */
             short int    seq_tab;               /* Address of sequence table */
             short int    seq_len;               /* Length of sequence table */
             short int    two_to_one;            /* Address of two_to_one table */
             short int    two_to_one_len;        /* Length of two_to_one table */
             short int    one_to_two;            /* Address of one_to_two table */
             short int    one_to_two_len;        /* Length of one_to_two table */
             char         low_char;              /* Lowest character */
             char         high_char;             /* Highest character */
        }
```

Sequence Table

Entries in the sequence table have the following format:

```
struct seq_ent {
     unsigned char    seq_no;       /* Sequence number */
     unsigned char    type_info;    /* Character type  */
}
```

The byte value of a given character is used as an index into the sequence table. The first two bits of *type_info* are used to keep track of the character type. A value zero means the character is a one_to_one character, and the other six bits in *type_info* contain its priority. A value of one or two means that *type_info* contains an index value into either the two_to_one or the one_to_two mapping table respectively. A value zero in *seq_no* means the character is a "don't care" character.

Mapping Table for two_to_one Mapped Characters

Entries in the two_to_one table have the following format:

```
struct two_to_one {
     char            reserved1;    /* Reserved */
     char            legal_char;   /* Legal character */
     struct seq_ent  seq2;         /* Sequence entry for this pair */
}
```

"Legal" two_to_one characters are listed for each particular character. "Legal" means that the combination of two characters is treated as a single character. If a match is found, then the corresponding sequence entry is used for the two. Whenever a legal successor is not found in table, the character is treated according to one_to_one mapping, and the priority in the last entry combined with sequence number of the character creates the sequence entry.

Mapping Table for one_to_two Mapped Characters

Entries in the one_to_two mapping table have the same format as entries in the sequence table. The sequence number of the first character is known from the entry in the sequence table. The sequence number of the second character is found in the one_to_two mapping entry, and the priority is used for both characters.

AUTHOR

Col_seq_8 was developed by the Hewlett-Packard Company.

SEE ALSO

nl_string(3C), sort(1).

INTERNATIONAL SUPPORT

8-bit data.

NAME

core – format of core image file

DESCRIPTION

The HP-UX system writes out a core image of a terminated process when any of various errors occur. See *signal*(2) for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called **core** and is written in the process's working directory (provided it can be; normal access controls apply). A process with an effective user ID different from the real user ID will not produce a core image.

The file contains sufficient information to determine what the process was doing at the time of its termination. Debuggers such as *cdb*(1) and *adb*(1) provide a uniform user interface to this information. The actual contents of the file are implementation-dependent; refer to the documentation for a specific HP-UX implementation for further details.

SEE ALSO

adb(1), cdb(1), setuid(2), signal(2).

NAME

core – format of core image file

Remarks:

Core(5) is implemented on Series 200 and 300 only.

DESCRIPTION

The HP-UX system writes out a core image of a terminated process when any of various errors occur. See *signal*(2) for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called **core** and is written in the process's working directory (provided it can be; normal access controls apply). A process with an effective user ID different from the real user ID will not produce a core image.

The first section of the core image is a copy of the system's per-user data for the process, including the registers as they were at the time of the fault. The size of this section depends on the parameter UPAGES which is defined in **/usr/include/sys/param.h**. The remainder represents the actual contents of the user's core area when the core image was written. If the text segment is read-only and shared, or separated from data space, it is not dumped.

The format of the information in the first section is described by the *user* structure of the system, defined in **/usr/include/sys/user.h**. The important stuff not detailed therein is the locations of the registers, which are outlined in **/usr/include/sys/reg.h**.

SEE ALSO

adb(1), cdb(1), setuid(2), signal(2).

NAME

core – format of core image file

DESCRIPTION

The HP-UX system writes out a core image of a terminated process when any of various errors occur. See *signal*(2) for the list of reasons; the most common are memory violations, illegal instructions, floating point exceptions, and user-generated quit signals. The core image is called **core** and is written in the process's working directory (provided it can be; normal access controls apply). A process with an effective user ID different from the real user ID will not produce a core image.

The first section of the core image is a header which contains information about the terminated process. The remainder represents the actual contents of the user's core area when the core image was written. This area contains the stack, user global data, and heap segments. The last object in the core image is the code segment fixup map which maps user code segments into real addresses.

The format of the information in the first section is described by the *user* structure of the system, defined in **<sys/user.h>**.

SEE ALSO

cdb(1), setuid(2), signal(2).

NAME

cpio – format of cpio archive

DESCRIPTION

The *header* structure, when the **–c** option of *cpio*(1) is not used, is:

```
struct {
        short   h_magic,
                h_dev;
        ushort  h_ino,
                h_mode,
                h_uid,
                h_gid;
        short   h_nlink,
                h_rdev,
                h_mtime[2],
                h_namesize,
                h_filesize[2];
        char    h_name[h_namesize rounded to word];
} Hdr;
```

When the **–c** option is used, the *header* information is described by:

```
        sscanf(Chdr,"%6ho%6ho%6ho%6ho%6ho%6ho%6ho%6ho%11lo%6ho%11lo",
                &Hdr.h_magic,&Hdr.h_dev,&Hdr.h_ino,&Hdr.h_mode,
                &Hdr.h_uid,&Hdr.h_gid,&Hdr.h_nlink,&Hdr.h_rdev,
                &Longtime,&Hdr.h_namesize,&Longfile);
```

Longtime and *Longfile* are equivalent to *Hdr.h_mtime* and *Hdr.h_filesize*, respectively. The contents of each file are recorded together with other items describing the file. Every instance of *h_magic* contains the constant 070707 (octal). The items *h_dev* through *h_mtime* have meanings explained in *stat*(2). The length of the null-terminated path name *h_name*, including the null byte, is given by *h_namesize*.

The last record of the *archive* always contains the name TRAILER!!!. Directories and the trailer are recorded with *h_filesize* equal to zero.

It will not always be the case that h_dev and h_ino correspond to the results of *stat(2)*, but the values are always sufficient to tell whether two files in the archive are linked to each other.

When a device special file is archived by HP-UX *cpio* (using -x), h_rdev will contain a magic constant which is dependent upon the implementation which is doing the writing. *H_rdev* flags the device file as an HP-UX 32-bit device specifier, and h_filesize will contain the 32-bit device specifier (see *stat(2)*). If the -x option is not present, special files are not archived or restored. Non-HP-UX device special files are never restored.

SEE ALSO

cpio(1), find(1), stat(2).

NAME
devices – file of driver information for insf, mksf, lssf

DESCRIPTION
The *devices* file contains a description of I/O drivers, pseudo-drivers, hardware addresses and block/character major numbers. It is created by *uxgen*(1). Normally, this file resides in the directory **/etc**.

This is an ASCII file consisting of zero or more lines where each line is terminated by a newline character. Each line begins with a name which normally represents an I/O driver or pseudo-driver. Tokens are separated by white space.

Each parameter in the line is preceded by a keyword. All parameters are optional. The keywords are: lu, address, b_major, c_major. They represent logical unit number, hardware address, block major number, character major number, respectively. Parameters may appear in any order after the name; however, they must be directly preceded by their keyword.

For example, lines from a *devices* file follow:

```
cn                                         c_major 0
disc0   lu 0      address 28.0.0  b_major 0   c_major 4
disc0   lu 1      address 28.0.2  b_major 0   c_major 4
```

AUTHOR
Devices was developed by HP.

SEE ALSO
insf(1), mksf(1), lssf(1).

NAME

dialups, d_passwd – dialup security control

DESCRIPTION

Dialups and *d_passwd* are used to control the dialup security feature of *login*(1). If */etc/dialups* is present, the first word on each line is compared with the name of the line upon which the login is being performed. (Including the **/dev/**, as returned by *ttyname*(3). If the login is occurring on a line found in *dialups*, dialup security is invoked. Anything after a space or tab is ignored.

When dialup security is invoked, *login*(1) will request an additional password, and check it against that found in */etc/d_passwd*. The command name found in the "program to use as shell" field of */etc/passwd* is used to select the password to be used. Each entry in *d_passwd* consists of three fields, separated by colons. The first is the command name, matching an entry in *passwd*. The second is the encrypted password to be used for dialup security for those users logging in to use that program. The third is commentary, but the second colon is required to delimit the end of the password. A null password is designated with two adjacent colons. The entry for */bin/sh* is used if no other entry matches the command name taken from *passwd*.

FILES

/etc/dialups	Dial in tty lines
/etc/d_passwd	Passwords

SEE ALSO

login(1), passwd(4).

NAME

dir – format of directories

SYNOPSIS

#include <types.h>
#include <sys/dir.h>

REMARKS

This entry describes the directory format for the HFS file system. Refer to other *dir*(4) manual pages for information valid for other implementations.

DESCRIPTION

A directory behaves exactly like an ordinary file, except that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry (see *fs*(4)). The structure of a directory entry as given in the **dir.h** include file is:

```
#define DIRSIZ        14
#define DIR_PADSIZE   10

struct  direct {
        u_long      d_ino;              /* inode number of entry */
        u_short     d_reclen;           /* length of this record */
        u_short     d_namlen;           /* length of string in d_name */
        char        d_name[DIRSIZ];     /* name must be no longer than this */
        char        d_pad[DIR_PADSIZE];
};
```

By convention, the first two entries in each directory are for **.** and **..** ("dot" and "dot dot"). The first is an entry for the directory itself. The second is for the parent directory. The meaning of **..** is modified for the root directory of the master file system; there is no parent, so **..** and **.** have the same meaning.

The *direct* structure defined here is the actual directory format for the HFS file system and is not compatible with other HP-UX supported file systems. The *direct* structure defined in */usr/include/ndir.h* should be used in conjunction with the directory(3C) library routines for compatibility across all HP-UX supported file systems.

HARDWARE DEPENDENCIES

The HFS file system is implemented on Series 200 beginning with HP-UX Release 5.0, and on Series 300 and Series 800.

AUTHOR

Dir was developed by the Hewlett-Packard Company, and the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

fs(4), directory(3C).

NAME
dir – format of directories

SYNOPSIS
#include <types.h>
#include <sys/dir.h>

Remarks:
This entry describes the SDF directory format for Series 500. Refer to other *dir*(5) manual pages for information valid for other implementations.

DESCRIPTION
A directory behaves exactly like an ordinary file, except that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry (see *inode*(5)). The structure of a directory entry as given in **sys/dir.h** is:

```
#ifndef DIRSIZ
#define DIRSIZ          14
#endif
struct  direct
{
        char            d_name[DIRSIZ+2];    /* 16-char file name */
        short           d_object_type;       /* not referenced by HP-UX */
        short           d_file_code;         /* not referenced by HP-UX */
        ino_t           d_ino;               /* use fir # for i-node */
};
```

The SDF directory implementation eliminates entries for **.** and **...** Instead, this information is available as part of the i-node.

File names are stored in directories in a special manner in two cases:

When a file name contains **embedded blanks**, the blanks are represented by null characters on the disc. This is apparent when accessing the disc in raw (character) mode.

When a file name is **blank padded**, all unspecified characters are set to blanks. Again, this is apparent only when reading from the disc in raw mode.

When a director has been opened vi *open*(2), file names appear as null-terminated, and contain embedded blanks where they belong.

The *direct* structure defined here is the actual directory format for the SDF file system, and is not compatible with other file systems supported on HP-UX. The *direct* structure defined in */usr/include/ndir.h* should be used in conjunction with the directory(3C) library routines for compatibility across all HP-UX supported file systems.

SEE ALSO
fs(4), inode(4), directory(3C).

NAME

disktab – disk description file

SYNOPSIS

#include <disktab.h>

DESCRIPTION

Disktab is a simple data base which describes disk geometries and disk section characteristics. Entries in *disktab* consist of a number of ':' separated fields. The first entry for each disk gives the names which are known for the disk, separated by '|' characters. The last name given should be a long name fully identifying the disk.

The following list indicates the normal values stored for each disk entry. Sectors are of size DEV_BSIZE, defined in <sys/param.h> on your system.

Name	**Type**	**Description**
ns	num	Number of sectors per track
nt	num	Number of tracks per cylinder
nc	num	Total number of cylinders on the disk
b0	num	Block size for section '0' (bytes)
b1	num	Block size for section '1' (bytes)
b<n>	num	Block size for section '<n>' (bytes)
f0	num	Fragment size for section '0' (bytes)
f1	num	Fragment size for section '1' (bytes)
f<n>	num	Fragment size for section '<n>' (bytes)
s0	num	Size of section '0' in sectors
s1	num	Size of section '1' in sectors
s<n>	num	Size of section '<n>' in sectors
rm	num	Revolution per minute
ty	str	Type of disk (e.g. removable, winchester)

Example:

```
hp7914:    :ty=winchester:ns#16:nt#7:nc#1061:s0#118832\
           :b0#8192:f0#1024:rm#3600:
```

HARDWARE DEPENDENCIES

The HFS file system is implemented on Series 200 beginning with HP-UX Release 5.0, and on Series 300 and Series 800.

Series 200:

The Series 200 5.0 release can have only one section per disk drive.

FILES

/etc/disktab

AUTHOR

Disktab was developed by the Hewlett-Packard Company, and the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

newfs(1M)

NAME

DOSIF – DOS Interchange Format description

DESCRIPTION

DOSIF (DOS Interchange Format) is the name given to the media format used by the DOS operating system. This format is based upon that used in IBM PC and PCAT, HP Vectra, and HP 150 systems.

The DOS utilities described in Section 1 are provided for reading and writing data to and from DOSIF volumes. These utilities (referred to hereafter as *dos**(1) can be used to retrieve information from a DOSIF volume.

The *dos**(1) utilities in Section 1 are the only entities within the HP-UX operating system that can interact directly with the contents of a DOSIF volume. All other HP-UX utilites and facilities can only treat a DOSIF volume as a file containing unspecified data. *Mount*(1) must not be used on a DOSIF volume because the operating system cannot recognize it.

When specifying file names for DOSIF commands, concatenate the HP-UX path name in front of the DOSIF volume and file name, separating the two with a colon (:). For example,

/dev/fd.0:/users/ivy

specifies DOSIF file */users/ivy* accessed through HP-UX device special file */dev/fd.0*.

Note that this file naming convention is only suitable for use as arguments to the *dos**(1) utilities, and does not constitute a legal path name for any other use within HP-UX. Also, the shell (*sh*(1)) meta characters: * ? and [...] cannot be used to specify an arbitrary pattern for file name matching when using DOSIF utilities.

If the HP-UX device name and a trailing colon are specified but no file or directory name is provided (e.g. **/dev/rfd.0:**), the root (/) of the DOS file system is assumed by convention.

A primitive form of data protection is provided by a lockfile */tmp/DOS..LCK* that permits only one process and its immediate children to use the DOSIF utilities at any given time.

SEE ALSO

doschmod(1), doscp(1), dosdf(1), dosls(1), dosmkdir(1), dosrm(1).

WARNINGS

*Dos** routines maintain a cache in order to reduce the number of mass storage device accesses. When referring to several DOSIF files within a single command, be sure to refer to the device file in exactly the same way. For example,

doscp /dev/x:alpha //dev/x:beta

is sure to create unexpected results because the DOSIF utilities treat **/dev/x** and **//dev/x** as two different devices.

NAME

errfile – system error logging file

Remarks:

This manual page describes *errfile* as implemented on the Series 500. Refer to other *errfile* manual pages for information valid for other implementations.

DESCRIPTION

Errfile is a logging file containing lines of ASCII text. Each line describes certain system errors that have occurred, or warnings about serious system conditions. Only those system error messages deemed serious enough to be of interest to the system administrator are logged. Urgent messages are also written to /dev/console.

HP-UX creates *errfile* if it does not exist.

The system administrator should check the contents of *errfile* periodically and note errors that need attention. Also, *errfile* tends to grow without bounds, so outdated information needs to be removed on a regular basis.

FILES

usr/adm/errfile

SPECIAL NOTE

errfile reports the number of logical blocks left on the disk. The size of those logical blocks is defined at system installation and configuration.

NAME

fs – format of file system volume

SYNOPSIS

#include <sys/types.h>
#include <sys/param.h>
#include <sys/fs.h>
#include <sys/ino.h>
#include <sys/inode.h>
#include <sys/sysmacros.h>

MARKETING MODEL

To be determined.

TECHNICAL MODEL

Core System Base
HP+ (HFS Subset)

DESCRIPTION

Every file system storage volume has a common format for certain vital information. The first 8 kbytes on a volume contain a volume header which identifies that volume as a LIF volume. Such volume may be divided into a number of sections.

Each section can contain a file system. The first 8 kbytes in each section is ignored, except where it coincides with the volume header discussed above. The actual file system begins next with the *super block.* The layout of the super block as defined by the include file <*sys/fs.h*> is:

```
#define FS_MAGIC     0x011954
#define FS_CLEAN     0x17
#define FS_OK        0x53
#define FS_NOTOK     0x37

struct fs {
        struct fs    *fs_link;          /* linked list of file systems */
        struct fs    *fs_rlink;         /* used for incore super blocks */
        daddr_t      fs_sblkno;         /* addr of super-block in filesys */
        daddr_t      fs_cblkno;         /* offset of cyl-block in filesys */
        daddr_t      fs_iblkno:         /* offset of inode-blocks in filesys */
        daddr_t      fs_dblkno;         /* offset of first data after cg */
        long         fs_cgoffset;       /* cylinder group offset in cylinder */
        long         fs_cgmask;         /* used to calc mod fs_ntrak */
        time_t       fs_time;           /* last time written */
        long         fs_size;           /* number of blocks in fs */
        long         fs_dsize;          /* number of data blocks in fs */
        long         fs_ncg;            /* number of cylinder groups */
        long         fs_bsize;          /* size of basic blocks in fs */
        long         fs_fsize;          /* size of frag blocks in fs */
        long         fs_frag;           /* number of frags in a block in fs */
/* these are configuration parameters */
        long         fs_minfree;        /* minimum percentage of free blocks */
        long         fs_rotdelay;       /* num of ms for optimal next block */
        long         fs_rps;            /* disk revolutions per second */
/* these fields can be computed from the others */
        long         fs_bmask;          /* "blkoff" calc of blk offsets */
        long         fs_fmask;          /* "fragoff" calc of frag offsets */
        long         fs_bshift;         /* "lblkno" calc of logical blkno */
        long        fs_fshift;                /* "numfrags" calc number of frags */
/* these are configuration parameters */
        long        fs_maxcontig;             /* max number of contiguous blks */
        long        fs_maxbpg;                /* max number of blks per cyl group */
/* these fields can be computed from the others */
        long        fs_fragshift;             /* block to frag shift */
        long        fs_fsbtodb;               /* fsbtodb and dbtofsb shift constant */
        long        fs_sbsize;                /* actual size of super block */
        long        fs_csmask;                /* csum block offset */
        long        fs_csshift;               /* csum block number */
        long        fs_nindir;                /* value of NINDIR */
        long        fs_inopb;                 /* value of INOPB */
        long        fs_nspf;                  /* value of NSPF */
        long        fs_sparecon[6];           /* reserved for future constants */
/* sizes determined by number of cylinder groups and their sizes */
        daddr_t fs_csaddr;                    /* blk addr of cyl grp summary area */
        long        fs_cssize;                /* size of cyl grp summary area */
        long        fs_cgsize;                /* cylinder group size */
/* these fields should be derived from the hardware */
        long        fs_ntrak;                 /* tracks per cylinder */
        long        fs_nsect;                 /* sectors per track */
        long        fs_spc;                   /* sectors per cylinder */
/* this comes from the disk driver partitioning */
        long        fs_ncyl;                  /* cylinders in file system */
/* these fields can be computed from the others */
        long        fs_cpg;                   /* cylinders per group */
        long        fs_ipg;                   /* inodes per group */
        long        fs_fpg;                   /* blocks per group * fs_frag */
/* this data must be re-computed after crashes */
        struct      csum fs_cstotal;          /* cylinder summary information */
/* these fields are cleared at mount time */
        char        fs_fmod;                  /* super block modified flag */
        char        fs_clean;                 /* file system is clean flag */
        char        fs_ronly;                 /* mounted read-only flag */
        char        fs_flags;                 /* currently unused flag */
        char        fs_fsmnt[MAXMNTLEN];      /* name mounted on */
/* these fields retain the current block allocation info */
        long        fs_cgrotor;               /* last cg searched */
        struct      csum *fs_csp[MAXCSBUFS];/* list of fs_cs info buffers */
        long        fs_cpc;                   /* cyl per cycle in postbl */
        short       fs_postbl[MAXCPG][NRPOS];/* head of blocks for each rotation */
        long        fs_magic;                 /* magic number */
        char        fs_name[6];               /* name of file system */
        char        fs_fpack[6];              /* pack name of file system */
        u_char      fs_rotbl[1];              /* list of blocks for each rotation */
/* actually longer */ };
```

A file system consists of a number of cylinder groups. Each cylinder group has inodes and data.

A file system is described by its super-block, which in turn describes the cylinder groups. The super-block is critical data and is replicated in each cylinder group to protect against catastrophic loss. This is done at file system creation time and the critical super-block data does not change, so the copies need not be referenced further unless disaster strikes.

Addresses stored in inodes are capable of addressing fragments of 'blocks'. File system blocks of at most size MAXBSIZE can be optionally broken into smaller pieces, each of which is addressable; these pieces may be DEV_BSIZE, or some multiple of a DEV_BSIZE unit.

Large files consist of exclusively large data blocks. To avoid undue wasted disk space, the last data block of a file is allocated only as many fragments of a large block as are necessary, if that file is small enough to not require indirect data blocks. The file system format retains only a single pointer to such a fragment, which is a piece of a single large block that has been divided. The size of such a fragment is determinable from information in the inode, using the "blksize(fs, ip, lbn)" macro.

The file system records space availability at the fragment level; to determine block availability, aligned fragments are examined.

I-numbers begin at 0. Inodes 0 and 1 are reserved. Inode 2 is used for the root directory of the file system. The *lost+found* directory is given the next available inode when it is initially created by *mkfs*.

Fs_minfree gives the minimum acceptable percentage of file system blocks which may be free. If the freelist drops below this level only the super-user may continue to allocate blocks. This may be set to 0 if no reserve of free blocks is deemed necessary, however severe performance degradations will be observed if the file system is run at greater than 90% full; thus the default value of *fs_minfree* is 10%.

The best trade-off between block fragmentation and overall disk utilization and performance varies for each intended use of the file system. Suggested values can be found in the System Administrator's Manual for each implementation.

Cylinder group related limits: Each cylinder keeps track of the availability of blocks at different rotational positions, so that sequential blocks can be laid out with minimum rotational latency. NRPOS is the number of rotational positions which are distinguished. For example, with NRPOS 8 the resolution of the summary information is 2ms for a typical 3600 rpm drive.

Fs_rotdelay gives the minimum number of milliseconds to initiate another disk transfer on the same cylinder. It is used in determining the rotationally optimal layout for disk blocks within a file; the default value for *fs_rotdelay* is 2ms. Suggested values of fs_rotdelay for different disks can be found in the System Administrator's Manual.

Each file system has a statically allocated number of inodes. An inode is allocated for each NBPI bytes of disk space. The inode allocation strategy is extremely conservative.

MAXIPG bounds the number of inodes per cylinder group, and is needed only to keep the structure simpler by having only a single variable size element (the free bit map).

N.B.: MAXIPG must be a multiple of INOPB(fs).

MINBSIZE is the smallest allowable block size. With a MINBSIZE of 4096 it is possible to create files of size 2^32 with only two levels of indirection. MINBSIZE must be big enough to hold a cylinder group block, thus MINBSIZE must always be greater than sizeof(struct cg). Note that super blocks are never more than size SBSIZE.

The path name on which the file system is mounted is maintained in *fs_fsmnt*. MAXMNTLEN defines the amount of space allocated in the super block for this name. The limit on the amount of summary information per file system is defined by MAXCSBUFS. It is currently parameterized for a maximum of two million cylinders.

Per cylinder group information is summarized in blocks allocated from the first cylinder group's data blocks. These blocks are read in from *fs_csaddr* (size *fs_cssize*) in addition to the super block.

N.B.: sizeof (struct csum) must be a power of two in order for the "fs_cs" macro to work.

Super block for a file system: MAXBPC bounds the size of the rotational layout tables and is limited by the fact that the super block is of size SBSIZE. The size of these tables is inversely proportional to the block size of the file system. The size of the tables is increased when sector sizes are not powers of two, as this increases the number of cylinders included before the rotational pattern repeats (*fs_cpc*). The size of the rotational layout tables is derived from the number of bytes remaining in (struct fs).

MAXBPG bounds the number of blocks of data per cylinder group, and is limited by the fact that cylinder groups are at most one block. The size of the free block table is derived from the size of blocks and the number of remaining bytes in the cylinder group structure (struct cg).

Inode: The inode is the focus of all file activity in the There is a unique inode allocated for each active file, each current directory, each mounted-on file, text file, and the root. An inode is 'named' by its device/i-number pair. For the format of an inode and its flags, see *inode(4)*

HARDWARE DEPENDENCIES

The HFS file system is implemented on Series 200 beginning with HP-UX Release 5.0, and on Series 300 and Series 800.

Series 200

Series 200 5.0 release supports only one section per volume. Thus, there can only be one file system on each volume and the first 8 kbytes of a file system is the boot area. This area contains the LIF volume header, the directory that defines the contents of the volume and the bootstrapping program.

AUTHOR

Fs[HFS] was developed by HP, and the University of California, Berkeley.

SEE ALSO

inode[HFS](4), lif(4).

NAME

fs – format of system volume

SYNOPSIS

#include <sys/param.h>
#include <sys/filsys.h>

Remarks: This manual page describes the format of the system volume as implemented on the Series 500. Refer to other *fs* manual pages for information valid for other implementations.

DESCRIPTION

Every Structured Directory Format (SDF) volume is divided into logical blocks, the size of which is selected when *init* is executed. Block 0 is the superblock. It has the following format:

```
struct filsys {
        ushort      s_format;       /* disc fmt, should = 0x700 Unix */
        ushort      s_corrupt;      /* non-zero if directory corrupt */
        char        s_fname[16];    /* root dir name, blank padded */
        time_ios    s_init;         /* date initialized / unique id */
        int         s_blksz;        /* no. bytes per block */
        daddr_t     s_boot;         /* boot area starting block */
        int         s_bootsz;       /* size of boot area in blks */
        daddr_t     s_fa;           /* FA file starting block */
        int         s_version;      /* version no., 0 for Unix */
        daddr_t     s_maxblk;       /* largest addressable blk */
        char        s_passwd[16];   /* volume password, Unix unused */
        time_ios    s_bkup;         /* last backup date, Unix unused */
                                    /* rest of blk unused */
};
```

The file attributes file (FA file) begins at the block specified by *s_fa* in the superblock. It has five major sections:

Each entry consists of 128 bytes. Entry 0 is the i-node of the FA file itself (see *inode*(5) for a description of the i-node structure). Entry 1 is the i-node for the file system's root directory, /.

Entry 3 through entry n consists of the free map, which keeps track of every free (unused) block of memory on the device. The free map contains a bit for each block on the device. If a bit is set, the corresponding block of memory is free; otherwise, the corresponding block is being used. The free map is zero-padded to guarantee that it ends on a 128-byte boundary.

Entry n+1 through the end of the FA file contains an entry for every file in the system. Each entry is either an i-node, an extent map, or unused. An extent map contains 128 bytes of information, and looks as follows:

```
struct em_rec {
        ushort      e_type;      /* =2 for extent maps */
        ushort      e_exnum;     /* # extents in this rec. */
        int         e_res1;      /* unused */
        ino_t       e_next;      /* next map in list; none = neg */
        ino_t       e_last;      /* last map in list; none = neg */
        ino_t       e_inode;     /* owner i-node no. */
        daddr_t     e_boffset;   /* blk offset of 1st extent from start of file */
        struct {
                daddr_t     e_startblk;     /* extent start blk */
                int         e_numblk;       /* # blks in extent */
        }       e_extent[13];
};
```

FILES

/usr/include/sys/param.h
/usr/include/sys/filsys.h
/usr/include/sys/ino.h

SEE ALSO

inode(4), fsck(1M).

NAME

fspec – format specification in text files

DESCRIPTION

It is sometimes convenient to maintain text files on the HP-UX system with non-standard tabs, (i.e., tabs which are not set at every eighth column). Such files must generally be converted to a standard format, frequently by replacing all tabs with the appropriate number of spaces, before they can be processed by HP-UX system commands. A format specification occurring in the first line of a text file specifies how tabs are to be expanded in the remainder of the file.

A format specification consists of a sequence of parameters separated by blanks and surrounded by the brackets <: and :>. Each parameter consists of a keyletter, possibly followed immediately by a value. The following parameters are recognized:

t*tabs* The **t** parameter specifies the tab settings for the file. The value of *tabs* must be one of the following:

1. a list of column numbers separated by commas, indicating tabs set at the specified columns;
2. a – followed immediately by an integer *n*, indicating tabs at intervals of *n* columns;
3. a – followed by the name of a "canned" tab specification.

Standard tabs are specified by **t–8**, or equivalently, **t1,9,17,25,**etc. The canned tabs which are recognized are defined by the *tabs*(1) command.

s*size* The **s** parameter specifies a maximum line size. The value of *size* must be an integer. Size checking is performed after tabs have been expanded, but before the margin is prepended.

m*margin* The **m** parameter specifies a number of spaces to be prepended to each line. The value of *margin* must be an integer.

d The **d** parameter takes no value. Its presence indicates that the line containing the format specification is to be deleted from the converted file.

e The **e** parameter takes no value. Its presence indicates that the current format is to prevail only until another format specification is encountered in the file.

Default values, which are assumed for parameters not supplied, are **t–8** and **m0**. If the **s** parameter is not specified, no size checking is performed. If the first line of a file does not contain a format specification, the above defaults are assumed for the entire file. The following is an example of a line containing a format specification:

```
* <:t5,10,15 s72:> *
```

If a format specification can be disguised as a comment, it is not necessary to code the **d** parameter.

Several HP-UX system commands correctly interpret the format specification for a file. Among them is *ed*(1), which may be used to convert files to a standard format acceptable to other HP-UX system commands.

SEE ALSO

ed(1), newform(1), tabs(1).

NAME

gettydefs – speed and terminal settings used by getty

DESCRIPTION

The **/etc/gettydefs** file contains information used by *getty*(1M) to set up the speed and terminal settings for a line. It supplies information on what the *login* prompt should look like. It also supplies the speed to try next if the user indicates the current speed is not correct by typing a <*break*> character.

Each entry in **/etc/gettydefs** has the following format:

> label# initial-flags # final-flags # login-prompt #next-label

Each entry is followed by a blank line. The various fields can contain quoted characters of the form **\b**, **\n**, **\c**, etc., as well as \nnn, where *nnn* is the octal value of the desired character. The various fields are:

label	This is the string against which *getty* tries to match its second argument. It is often the speed, such as **1200**, at which the terminal is supposed to run, but it need not be (see below).
initial-flags	These flags are the initial *ioctl*(2) settings to which the terminal is to be set if a terminal type is not specified to *getty*. The flags that *getty* understands are the same as the ones listed in **/usr/include/sys/termio.h** (see *termio*(7)). Normally only the speed flag is required in the *initial-flags*. *Getty* automatically sets the terminal to raw input mode and takes care of most of the other flags. The *initial-flag* settings remain in effect until *getty* executes *login*(1).
final-flags	These flags take the same values as the *initial-flags* and are set just prior to *getty* executes *login*. The speed flag is again required. The composite flag **SANE** takes care of most of the other flags that need to be set so that the processor and terminal are communicating in a rational fashion. The other two commonly specified *final-flags* are **TAB3**, so that tabs are sent to the terminal as spaces, and **HUPCL**, so that the line is hung up on the final close.
login-prompt	This entire field is printed as the *login-prompt*. Unlike the above fields where white space is ignored (a space, tab or new-line), they are included in the *login-prompt* field.
next-label	If this entry does not specify the desired speed, indicated by the user typing a <*break*> character, then *getty* will search for the entry with *next-label* as its *label* field and set up the terminal for those settings. Usually, a series of speeds are linked together in this fashion, into a closed set; For instance, **2400** linked to **1200**, which in turn is linked to **300**, which finally is linked to **2400**.

If *getty* is called without a second argument, then the first entry of **/etc/gettydefs** is used, thus making the first entry of **/etc/gettydefs** the default entry. It is also used if *getty* can not find the specified *label*. If **/etc/gettydefs** itself is missing, there is one entry built into the command which will bring up a terminal at **300** baud.

It is strongly recommended that after making or modifying **/etc/gettydefs**, it be run through *getty* with the check option to be sure there are no errors.

EXAMPLES

The following two lines show an example of 300/1200 baud toggle, which is useful for dial-up ports:

```
1200# B1200 HUPCL # B1200 SANE IXANY IXANY TAB3 #login: #300
300# B300 HUPCL # B300 SANE IXANY IXANY TAB3 #login: #1200
```

The following line shows a typical 9600 baud entry for a hard-wired connection:

```
9600# B9600 # B9600 SANE IXANY IXANY ECHOE TAB3 #login: #9600
```

FILES

/etc/gettydefs

SEE ALSO

getty(1M), login(1), ioctl(2), termio(7).

NAME

group – group file, grp.h

DESCRIPTION

Group contains for each group the following information:

group name
encrypted password
numerical group ID
comma-separated list of all users allowed in the group

This is an ASCII file. The fields are separated by colons; each group is separated from the next by a new-line. If the password field is null, no password is associated with the group.

There are two files of this form in the system, /etc/group and /etc/logingroup. /etc/group exists to supply names for each group, and to support changing groups via *newgrp*(1). /etc/logingroup provides a default group access list for each user via *login*(1) and *initgroups*(3C).

The real and effective group ID set up by *login* for each user is defined in /etc/passwd (see *passwd*(4). If /etc/logingroup is empty or non-existent, the default group access list is empty. If /etc/logingroup and /etc/group are links to the same file, the default access list includes the entire set of groups associated with the user. The group name and password fields in /etc/logingroup are never used; they are included only to give the two files a uniform format, allowing them to be linked together.

All group ID's used in /etc/logingroup or /etc/passwd should be defined in /etc/group. No user should be associated with more than **NGROUPS** (see *setgroups*(2)) groups in /etc/logingroup.

These files reside in directory **/etc**. Because of the encrypted passwords, they can and do have general read permission and can be used, for example, to map numerical group ID's to names.

Grp.h describes the group structure returned by getgrent(3C), etc:

```
/* see getgrent(3C) */
     struct        group {
                   char       *gr_name;
                   char       *gr_passwd;
                   int        gr_gid;
                   char       **gr_mem;
     };
```

WARNINGS

The gid 9 is reserved for the Pascal Language operating system and the BASIC Language operating system. These are operating systems for the Series 200 and Series 300 computers that can coexist with HP-UX on the same disk. Using this gid for other purposes can inhibit file transfer and sharing.

FILES

/etc/group
/etc/logingroup

SEE ALSO

groups(1), newgrp(1), passwd(1), setgroups(2), crypt(3C), getgrent(3C), initgroups(3C), passwd(4).

BUGS

There is no tool that helps you ensure that /etc/passwd, /etc/group, and /etc/logingroup are compatible.

There is no tool that helps you set group passwords in /etc/group.

NAME

inittab – script for the init process

DESCRIPTION

The *inittab* file supplies the script to *init*'s role as a general process dispatcher. The process that constitutes the majority of *init*'s process dispatching activities is the line process **/etc/getty** that initiates individual terminal lines. Other processes typically dispatched by *init* are daemons and the shell.

The *inittab* file is composed of entries that are position dependent and have the following format:

id:rstate:action:process

Each entry is delimited by a newline, however, a backslash (\) preceding a newline indicates a continuation of the entry. Up to 512 characters per entry are permitted. Comments may be inserted in the *process* field using the *sh*(1) convention for comments. Comments for lines that spawn *getty*s are displayed by the *who*(1) command. It is expected that they will contain some information about the line such as the location. There are no limits (other than maximum entry size) imposed on the number of entries within the *inittab* file. The entry fields are:

id This is one or two characters used to uniquely identify an entry.

rstate This defines the *run-level* in which this entry is to be processed. *Run-levels* effectively correspond to a configuration of processes in the system. That is, each process spawned by *init* is assigned a *run-level* or *run-levels* in which it is allowed to exist. The *run-levels* are represented by a number ranging from **0** through **6**. As an example, if the system is in *run-level* **1**, only those entries having a **1** in the *rstate* field will be processed. When *init* is requested to change *run-levels*, all processes which do not have an entry in the *rstate* field for the target *run-level* will be sent the warning signal (**SIGTERM**) and allowed a 20-second grace period before being forcibly terminated by a kill signal (**SIGKILL**). The *rstate* field can define multiple *run-levels* for a process by selecting more than one *run-level* in any combination from **0–6**. If no *run-level* is specified, then the process is assumed to be valid at all *run-levels* **0–6**. There are three other values, **a**, **b** and **c**, which can appear in the *rstate* field, even though they are not true *run-levels*. Entries which have these characters in the *rstate* field are processed only when the *telinit* (see *init*(1M)) process requests them to be run (regardless of the current *run-level* of the system). They differ from *run-levels* in that *init* can never enter *run-level* **a**, **b** or **c**. Also, a request for the execution of any of these processes does not change the current *run-level*. Furthermore, a process started by an **a**, **b** or **c** command is not killed when *init* changes levels. They are only killed if their line in **/etc/inittab** is marked **off** in the *action* field, their line is deleted entirely from **/etc/inittab**, or *init* goes into the *SINGLE USER* state.

action Key words in this field tell *init* how to treat the process specified in the *process* field. The actions recognized by *init* are as follows:

respawn If the process does not exist then start the process, do not wait for its termination (continue scanning the *inittab* file), and when it dies restart the process. If the process currently exists then do nothing and continue scanning the *inittab* file.

wait Upon *init*'s entering the *run-level* that matches the entry's *rstate*, start the process and wait for its termination. All subsequent reads of the *inittab* file while *init* is in the same *run-level* will cause *init* to ignore this entry.

once Upon *init*'s entering a *run-level* that matches the entry's *rstate*, start the process, do not wait for its termination. When it dies, do not restart the process. If upon entering a new *run-level*, where the process is still running from a previous *run-level* change, the program will not be restarted.

boot The entry is to be processed only at *init*'s boot-time read of the *inittab* file. *Init* is to start the process, not wait for its termination; and when it dies, not restart the process. In order for this instruction to be meaningful, the *rstate* should be the default or it must match *init*'s *run-level* at boot time. This action is useful for an initialization function following a hardware reboot of the system.

bootwait The entry is to be processed only at *init*'s boot-time read of the *inittab* file. *Init* is to start the process, wait for its termination and, when it dies, not restart the process.

powerfail Execute the process associated with this entry only when *init* receives a power fail signal (**SIGPWR** see *signal*(2)).

powerwait
Execute the process associated with this entry only when *init* receives a power fail signal (**SIGPWR**) and wait until it terminates before continuing any processing of *inittab*.

off If the process associated with this entry is currently running, send the warning signal (**SIGTERM**) and wait 20 seconds before forcibly terminating the process via the kill signal (**SIGKILL**). If the process is nonexistent, ignore the entry.

ondemand
This instruction is really a synonym for the **respawn** action. It is functionally identical to **respawn** but is given a different keyword in order to divorce its association with *run-levels*. This is used only with the **a**, **b** or **c** values described in the *rstate* field.

initdefault
An entry with this *action* is only scanned when *init* initially invoked. *Init* uses this entry, if it exists, to determine which *run-level* to enter initially. It does this by taking the highest *run-level* specified in the **rstate** field and using that as its initial state. If the *rstate* field is empty, this is interpreted as **0123456** and so *init* will enter *run-level* **6**. Also, the **initdefault** entry cannot specify that *init* start in the *SINGLE USER* state. Additionally, if *init* does not find an **initdefault** entry in **/etc/inittab**, then it will request an initial *run-level* from the user at reboot time.

sysinit Entries of this type are executed before *init* tries to access the console. It is expected that this entry will be only used to initialize devices on which *init* might try to ask the *run-level* question. These entries are executed and waited for before continuing.

process This is a *sh* command to be executed. The entire **process** field is prefixed with *exec* and passed to a forked *sh* as **sh -c 'exec** *command***'**. For this reason, any legal *sh* syntax can appear in the *process* field. Comments can be inserted with the **;** #*comment* syntax.

FILES

/etc/inittab

SEE ALSO

getty(1M), sh(1), exec(2), open(2), signal(2),

NAME

inode – format of an inode

SYNOPSIS

#include <sys/types.h>
#include <sys/ino.h>

REMARKS

This entry describes the inode structure for the HFS file system. Refer to other *inode*(4) manual pages for information valid for other implementations.

DESCRIPTION

An inode for a plain file or directory in a file system has the following structure defined by **<sys/ino.h>**.

```
/* Inode structure as it appears on a disk block */

struct  dinode {

        u_short     di_mode;            /* mode and type of file */
        short       di_nlink;           /* number of links to file */
        short       di_uid;             /* owner's user id */
        short       di_gid;             /* owner's group id */
        quad        di_size;            /* number of bytes in file */
        time_t      di_atime;           /* time last accessed */
        long        di_atspare;
        time_t      di_mtime;           /* time last modified */
        long        di_mtspare;
        time_t      di_ctime;           /* time of last file status change */
        long        di_ctspare;
        daddr_t     di_db[NDADDR];      /* disk block addresses */
        daddr_t     di_ib[NIADDR];      /* indirect blocks */
        long        di_flags;           /* status, currently unused */
        long        di_blocks;          /* blocks actually held */
        long        di_spare[5];        /* reserved, currently unused */
};
```

For the meaning of the defined types *u_short, quad, daddr_t* and *time_t* see *types*(5).

See */usr/include/sys/inode.h* for the definition of inode structures for special files, pipes, or FIFO's.

HARDWARE DEPENDENCIES

The HFS file system is implemented on Series 200 beginning with HP-UX Release 5.0, and on Series 300 and Series 800.

FILES

/usr/include/sys/ino.h

SEE ALSO

stat(2), fs[HFS](4), types(5).

NAME
inode – format of an i-node

SYNOPSIS
#include <sys/types.h>
#include <sys/param.h>
#include <sys/ino.h>

Remarks:
This entry describes the i-node structure for the Series 500. Refer to other *inode* manual pages for information valid for other implementations.

DESCRIPTION
An i-node for an ordinary file or directory in a file system has the following structure, as defined in **sys/ino.h**:

```
/*
* I-node structure as it appears on disc. This i-node is actually
* a file information record (FIR) in the HP SDF disc format.
*/

struct dinode {
        ushort      di_type         /* =1 for inodes */
        ushort      di_ftype;       /* file type */
        ushort      di_count;       /* reference count */
        short       di_uftype;      /* user file type (LIF) */
        time_ios    di_ctime;       /* time created */
        unsigned    di_other;       /* public capabilities */
        ino_t       di_protect;     /* file protect rec. none=-1 */
        ino_t       di_label;       /* file label rec. none=-1 */
        int         di_blksz;       /* file size in blocks */
        int         di_max;         * largest byte writable */
        ushort      di_exsz;        /* recom. extent size */
        ushort      di_exnum;       /* no. i-node extents (1-4) */
        struct {
                    daddr_t         di_startblk;/* extent start blk */
                    int             di_numblk;/* no. blks in extent */
        } di_extent[4];
        ino_t       di_exmap;       /* inode 1st extent map */
                                    /* none = -1 */
        int         di_size;        /* current size, bytes */

        /* Warning! Next 2 fields apply only to directories */

        ino_t       di_parent;      /* inode of parent */
        char        di_name[16];    /* name of this directory */

        /* The remaining fields defined only for local */
        /* implementation of structured directory format.    */

        time_t      di_atime;       /* time last accessed */
        time_ios    di_mtime;       /* time last mod. */
        int         di_recsz;       /* logical record size */
        ushort      di_uid;         /* owner's user id */
        ushort      di_gid;         /* owner's group id */
        ushort      di_mode;        /* mode, type of file */
        char        di_res2[2];     /* unused */
```

```
                /* The next field used only if file is */
                /* a device file; otherwise it is zero */
                dev_t       di_dev;         /* description of device */
        };
```

The meaning of the type declarations included above can be found in *types*(7).

FILES

/usr/include/sys/ino.h

SEE ALSO

dir(4), fs(4), types(5).

NAME

issue – issue identification file

DESCRIPTION

The file **/etc/issue** contains the *issue* or project identification to be printed as a login prompt. This is an ASCII file which is read by program *getty* and then written to any terminal spawned or respawned from the *inittab* file.

FILES

/etc/issue

SEE ALSO

getty(1), login(1).

NAME

lif – logical interchange format description

DESCRIPTION

LIF (Logical Interchange Format) is a Hewlett-Packard standard disk format that may be used for interchange of files among various HP computer systems. A LIF volume contains a header (identifying it as a LIF volume) and a directory that defines the contents (i.e. files) of the volume. The size of the directory is fixed when the volume is initialized (see *lifinit*(1)) and sets an upper bound on the number of files that may be created on the volume.

HP-UX contains a set of utilities (referred to as *lif**(1)) that may be used to initialize a LIF volume (i.e. create a header and an empty directory), copy files to and from LIF volumes, list the contents of LIF volumes, remove LIF files, and rename LIF files.

The *lif**(1) utilities are the only utilities within HP-UX where the internal structure of a LIF volume is known. To the rest of HP-UX, a LIF volume is simply a file containing some unspecified data. The term 'LIF volume' should in no way be confused with the HP-UX notion of a file system volume or mountable volume.

The LIF utility on HP-UX currently supports three file types, ASCII(1), BINARY(-2) and BIN(-23951).

There are three copying modes associated with them.

ASCII If the copying mode is ASCII, and an HP-UX file is being copied to a LIF volume, the utility strips the trailing LF and prepends two bytes of record length to each record. These records are then written to a LIF formatted media. When copying a LIF ASCII file to HP-UX the two byte record length are stripped and a trailing LF is appended. These records are then written to the destination. In this mode of copying the length of the file is preserved. The default file type for this mode of copying is ASCII(1).

BINARY If the copying mode is BINARY, and an HP-UX file is being copied to a LIF volume, the utility simply appends two bytes for record length to each 1K byte record. A trailing fractional block will have a count reflecting the number of bytes in that block. No interpretation is placed on the content of the records. These records are then written to a LIF-format media. When copying a LIF file to an HP-UX file in BINARY copying mode, the record lengths are stripped and the content of records is directly written to the destination. In this mode of copying the length of the binary file is preserved. The default file type for this mode of copying is BINARY(-2).

RAW If the copying mode is RAW, and an HP-UX file is being copied to a LIF volume, the utility simply copies the raw data to the destination. File sizes which are not multiples of 256 bytes will be padded with nulls to the next higher multiple. Therefore, the file sizes are not preserved. When copying a LIF file to an HP-UX file in RAW mode, the information is directly copied without any interpretation placed on the content of the source. The default file type for this mode of copying is BIN(-23951).

A LIF volume may be created on any HP-UX file (either regular disk file or device special file) that supports random access via *lseek*(2). Note that you should not mount the special file before using the *lif**(1) routines. See *lifinit*(1) for details. Within a LIF volume, individual files are identified by 1 to 10 character file names. File names may consist of upper-case alphanumeric characters (A through Z, 0 through 9) and the underscore character (_). The first character of a LIF file name must be a letter. The *lif**(1) utilities will accept any file name, including illegal file names generated on other systems, but will only create legal names. For example, file names containing lower-case letters will be read but not created.

LIF file names are specified to the *lif**(1) utilities by concatenating the HP-UX path name for the LIF volume with the LIF file name, separating the two with a colon (:). For example:

/dev/fd.0:ABC	specifies LIF file ABC within HP-UX device special file /dev/fd.0.
myfile:ABC	specifies LIF file ABC within HP-UX disk file 'myfile'.

Note that this file naming convention is applicable only for use as arguments to the *lif**(1) utilities and do not constitute legal path names for any other use within HP-UX.

HARDWARE DEPENDENCIES

Series 500

You must use a character special file to access the media.

SEE ALSO

lifcp(1), lifinit(1), lifls(1), lifrename(1), lifrm(1).

NAME

magic – magic numbers for HP-UX implementations

SYNOPSIS

#include <magic.h>

DESCRIPTION

Magic.h localizes all information about HP-UX "magic numbers" in one file, and thus facilitates uniform treatment of magic numbers. This file specifies the location of the magic number in a file (always the start of the file) and the structure of the magic number:

```
struct magic_number {
    unsigned    short system_id;
    unsigned    short file_type;
};
typedef struct magic_number MAGIC;
```

Magic.h includes definitions for the system IDs of all HP machines running HP-UX, and file types that are common to all implementations. There may be additional implementation-dependent file types. The predefined file types are:

```
/* for object code files */
    #define RELOC_MAGIC    0x106    /* relocatable only */
    #define EXEC_MAGIC     0x107    /* normal executable */
    #define SHARE_MAGIC    0x108    /* shared executable */
    #define LISP_MAGIC     0x10C    /* compiled Lisp */
    #define HPE_MAGIC      0x150    /* HPE boot image */
```

The values for *system_id* are defined in *model*(4).

WARNINGS

Cpio files use a different form of magic number that is incompatible with *magic*(4).

HARDWARE DEPENDENCIES

Series 200, 300, 800

An additional file type is defined:

```
#define DEMAND_MAGIC   0x10B    /* demand-load executable */
```

SEE ALSO

ar(1), ld(1), a.out(4), ar(4), model(4).

NAME

master – master device information table

HP-UX COMPATIBILITY

Level: Config(1M) Support -- HP-UX/RUN ONLY

Origin: System V and HP

DESCRIPTION

This file is used by *config*(1M) to obtain device information that enables it to generate the configuration file. *Master* contains lines of various forms.

Software drivers are defined as follows:

Field 1: device name, used in the user-specified dfile (8 chars maximum)

Field 2: handler name, used by the kernel to prefix routines such as cs80_read, lp_write, ...

Field 3: element characteristics: 5 bits make up the mask
- Bit 1 - card
- Bit 2 - specified only once
- Bit 3 - required driver
- Bit 4 - block device
- Bit 5 - character device (LSB)

Field 4: functions for the device: 10 bits make up the mask
- Bit 1 - size handler
- Bit 2 - link routine
- Bit 3 - open handler
- Bit 4 - close handler
- Bit 5 - read handler
- Bit 6 - write handler
- Bit 7 - ioctl handler
- Bit 8 - select handler
- Bit 9 - seltru handler
- Bit 10 - C_ALLCLOSES flag (LSB)

Field 5: major device number if a block-type device; otherwise -1.

Field 6: major device number if a character-type device; otherwise -1.

Aliases for names are defined as follows:

Field 1: alias name => product number
Field 2: device name

Parameters are defined as follows:

Field 1: parameter name, as used in the user-specified dfile
Field 2: parameter name, as used in the #define statement in conf.c
Field 3: parameter value

SEE ALSO

config(1M)

NAME

mknod – create a special file entry

SYNOPSIS

#include <sys/mknod.h>

DESCRIPTION

Mknod.h provides utilities to pack and unpack device names as used by *mknod*(2). It contains the macro **dev = makedev(major, minor)** which packs the major and minor fields into a form suitable for *mknod*(2). It also contains **major(dev)** and **minor(dev)** which extract the corresponding fields.

The macro MINOR_FORMAT is a *printf* specification that prints the minor field in the format best suited to the particular implementation. The specification given by MINOR_FORMAT must cause the resulting string to indicate the base of the number in the same format as that used for C: no leading zero for decimal, leading zero for octal, and leading zero and 'x' for hexadecimal.

When a minor field is printed in the format specified by MINOR_FORMAT, each sub-field contained in the minor will be wholly contained in the mininum possible number of digits of the resulting string. (Splitting a field across unnecessary digits for the sake of packing is not done.)

SEE ALSO

mknod(1M), mknod(2).

NAME

mnttab – mounted file system table

SYNOPSIS

#include <sys/types.h>
#include <mnttab.h>

DESCRIPTION

Mnttab resides in directory **/etc** and contains a table of devices, mounted by the *mount*(1M) command, in the following structure as defined by **<mnttab.h>**:

```
        struct  mnttab {
                char      mt_dev[MNTLEN];
                char      mt_filsys[MNTLEN];
                short     mt_ro_flg;
                time_t    mt_time;
                };
```

Each entry is (2 x MNTLEN + 6) bytes in length (MNTLEN is defined in /usr/include/mnttab.h). The first MNTLEN bytes are the null-padded name of the place where the *special file* is mounted; the next MNTLEN bytes represent the null-padded root name of the mounted special file; the remaining 6 bytes contain the mounted *special file*'s read/write permissions and the date on which it was mounted.

The maximum number of entries in *mnttab* is based on the system parameter **NMOUNT** located in /usr/include/mnttab.h, which defines the number of allowable mounted special files.

CAVEATS

The table is present only for programs to return information about the mounted file systems. It does not matter to *mount* if there are duplicated entries nor to *umount* if a name cannot be found.

SEE ALSO

mount(1M), setmnt(1M).

NAME

model – HP-UX machine identification

SYNOPSIS

#include <model.h>

DESCRIPTION

There are some distinctions between the implementations of HP-UX due to hardware differences. Where such distinctions exist, conditional compilation or other definitions can be used to isolate the differences. Flags and typedefs to resolve these distinctions are collected in *model.h*. This file contains constants identifying various HP-UX implementations.

For example the header file *model.h* contains the following constants for the HP 9000 Series 200.

```
/* model.h for the HP 9000 Series 200 */

#define     HP_S_200     0x20A
#define     HP_S_500     0x208
#define     HP_S_800     0x20B
```

Other such constants will be added as HP-UX extends to other machines.

In addition, *model.h* has a statement defining the preprocessor constant *MYSYS* to represent the specific implementation for which compilation is desired. *MYSYS* will be equal to one of the constants above.

Conditional compilation may be used to adapt one file for execution on more than one HP-UX implementation, if it contains implementation- or architecture-dependent features. For instance,

```
#if MYSYS==HP_S_200
        <statements>
#endif
```

will cause the statements following the if statement to be compiled only for the HP 9000 Series 200.

Model.h also contains typedefs for several predefined types to enhance portability of certain types of code and of files.

int8, u_int8	Signed and unsigned 8-bit integers.
int16, u_int16	Signed and unsigned 16-bit integers.
int32, u_int32	Signed and unsigned 32-bit integers.
machptr, u_machptr	Signed and unsigned integers large enough to hold a pointer.

Certain C preprocessor conditional compilation variables are defined to aid in implementation-dependent code, see *cpp*(1).

SEE ALSO

cc(1), cpp(1), magic(4).

NAME

nlist – nlist structure format

SYNOPSIS

#include <nlist.h>

REMARKS

The exact content of the structure defined below can be best found by examining */usr/include/nlist.h*. It varies somewhat between the various implementations of HP-UX.

DESCRIPTION

Nlist(3C) can be used to extract information from a the symbol table in an object file. Because symbol tables are machine dependent (as defined in each implementation's copy of **<a.out.h>**) a header file, *nlist.h* is defined to encapsulate the differences.

The *nlist* function, when used with the *nlist* structure can be used to extract certain information about selected symbols in the symbol table. The data associated with each symbol is machine specific, thus only the name and position of the *n_name* field in the *nlist* structure is standardized by HP-UX. The rest of the structure includes at least the value and type of the symbol. The names and meanings of all fields not standardized will change no more than necessary.

```
struct nlist {
                char                 *n_name;
                /* other fields as needed; the following are suggested
                  if they apply */
                long                 n_value;
                unsigned char        n_type;
                unsigned char        n_length;
                short                n_unit;
                short                n_sdindex;
        };
```

SEE ALSO

nlist(3C), a.out(4).

NAME

passwd – password file, pwd.h

DESCRIPTION

Passwd contains for each user the following information:

login name
encrypted password
numerical user ID
numerical group ID
reserved field which will be used for identification
initial working directory
program to use as shell

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. Each user is separated from the next by a new-line. If the password field is null, no password is demanded. If the shell field is null, /bin/sh is used.

This file resides in directory **/etc**. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical user IDs to names.

The encrypted password consists of 13 characters chosen from a 64-character set of "digits" described below, except when the password is null, in which case the encrypted password is also null. Login can be prevented by entering in the password field a character that is not part of the set of digits(e.g. ∗).

The characters used to represent "digits" are **.** for 0, **/** for 1, **0** through **9** for 2–11, **A** through **Z** for 12–37, and **a** through **z** for 38–63.

Password aging is effected for a particular user if his encrypted password in the password file is followed by a comma and a non-null string of characters from the above alphabet. (Such a string must be introduced in the first instance by the super-user.) This string defines the "age" needed to implement password aging.

The first character of the age, M say, denotes the maximum number of weeks for which a password is valid. A user who attempts to login after his password has expired will be forced to supply a new one. The next character, m say, denotes the minimum period in weeks which must expire before the password may be changed. The remaining characters define the week (counted from the beginning of 1970) when the password was last changed. (A null string is equivalent to zero.) M and m have numerical values in the range 0–63 that correspond to the 64-character set of "digits" shown above. If $m = M = 0$ (derived from the string **.** or **..**) the user will be forced to change his password the next time he logs in (and the "age" will disappear from his entry in the password file). If $m > M$ (signified, e.g., by the string **./**) only the super-user will be able to change the password.

Pwd.h designates the broken out password file as obtained by *getpwent*(3C):

```
struct passwd {
      char    *pw_name;
      char    *pw_passwd;
      int     pw_uid;
      int     pw_gid;
      char    *pw_age;
      char    *pw_comment;
      char    *pw_gecos;
      char    *pw_dir;
      char    *pw_shell;
};
```

It is suggested that the range 0-99 not be used for user and group ID's (*pw_uid* and *pw_gid* in the above structure) so that IDs which may be assigned for system software do not conflict.

WARNINGS

The uid 17 is reserved for the Pascal Language operating system. The uid 18 is reserved for the BASIC Language operating system. These are operating systems for the Series 200 and Series 300 computers that can co-exist with HP-UX on the same disk. Using these uids for other purposes may inhibit file transfer and sharing.

HARDWARE DEPENDENCIES

Series 200, 300, 500:

The following fields have character limitations as noted:

the login name field can be no longer than 8 characters;

the initial working directory field can be no longer than 63 characters;

the program field can be no longer than 44 characters.

The results are unpredictable if these fields are longer than the limits specified above.

The reserved field, called *pw_gcos* in the data structures used by *getpwent*(3C), is reserved for future use. It currently may be used to contain any information the system manager desires, but such use may conflict with the use of future HP features. The correct operation of the system will never depend on this field, but some optional feature may specify its format and content.

FILES

/etc/passwd

SEE ALSO

login(1), passwd(1), a64l(3C), crypt(3C), getpwent(3C), group(4).

NAME

privgrp – format of privileged values

SYNOPSIS

#include <sys/privgrp.h>

DESCRIPTION

Setprivgrp(2) sets a mask of privileges, and *getprivgrp*(2) returns an array of structures giving privileged group assignments on a per group–id basis. *Privgrp.h* contains the constants and structures needed to deal with these system calls, and contains:

```
/*
 * Privileged group definitions --
 * the numeric values may vary between implementations.
 */
#define PRIV_RTPRIO     1
#define PRIV_MLOCK      2
#define PRIV_CHOWN      3

/* Maximum number of privileged groups in system */
#define PRIV_MAXGRPS  32

/*
 * Size of the privilege mask,
 * based on largest numbered privilege
 */
#define PRIV_MASKSIZ  1

/*
 * Structure defining the privilege mask
 */
struct privgrp_map {
   int    priv_groupno;
   unsigned int priv_mask[PRIV_MASKSIZ];
};
```

PRIV_RTPRIO allows access to the *rtprio*(2) system call.
PRIV_MLOCK allows access to the *plock*(2) system call.
PRIV_CHOWN allows access to the *chown*(2) system calls.

Privileges are described in a multi–word mask. The value of the **#define** for each privilege is interpreted as a bit index (counting from 1). Thus a group–id may have several different privileges associated with it by having different bits **or**'ed into the mask.

The system is configured with a maximum number of groups with special privileges. PRIV_MAXGRPS defines this maximum. Of this maximum, one is reserved for global privileges (granted to all processes), and the remainder can be assigned to actual group–ids.

PRIV_MASKSIZ defines the size of the multi–word mask used defining privileges associated with a group–id.

Privileges are returned to the user from the *getprivgrp*(2) system call in an array of structures of type **struct privgrp_map**. The structure associates a multi–word mask with a group–id.

SEE ALSO

getprivgrp(2)

NAME

profile – set up user's environment at login time

DESCRIPTION

If the file **/etc/profile** exists, it is executed by the shell for every user who logs in. The file **/etc/profile** should be set up to do only those things that are desirable for *every* user on the system, or to set reasonable defaults. If your login (home) directory contains a file named **.profile**, that file will be executed (via the shell's **exec .profile**) before your session begins. **.Profile** files are useful for setting various environment parameters, setting terminal modes, or overriding some or all of the results of executing **/etc/profile**.

The following example is typical (except for the comments):

```
# Make some environment variables global
export MAIL PATH TERM
# Set file creation mask
umask 22
# Tell me when new mail comes in
MAIL=/usr/mail/myname
# Add my /bin directory to the shell search sequence
PATH=$PATH:$HOME/bin
# Set terminal type
echo "terminal: \c"
read TERM
case $TERM in
        300)            stty cr2 nl0 tabs; tabs;;
        300s)           stty cr2 nl0 tabs; tabs;;
        450)            stty cr2 nl0 tabs; tabs;;
        hp)             stty cr0 nl0 tabs; tabs;;
        745|735)        stty cr1 nl] -tabs; TERM=745;;
        43)             stty cr1 nl0 -tabs;;
        4014|tek)       stty cr0 nl0 -tabs ffl; TERM=4014; echo "33;";;
        *)              echo "$TERM unknown";;
esac
```

A more complete model **.profile** may be found in **/etc/d.profile**.

FILES

$HOME/.profile
/etc/profile

HARDWARE DEPENDENCIES

Integral PC
The file **/etc/d.profile** is not supported.

SEE ALSO

env(1), login(1), mail(1), sh(1), stty(1), su(1), environ(5), term(5).

NAME

ranlib – archive symbol table format for object libraries

SYNOPSIS

#include <ranlib.h>

DESCRIPTION

Any archive containing object files also includes an archive symbol table, thus allowing the linker *ld* to scan libraries in random (rather than sequential) order.

The archive symbol table (if it exists) is always the first file in the archive, but it is never listed. It is automatically created and/or updated by *ar*.

The archive symbol table lists each externally known name in the archive, together with the offset of the archive element that defines that name. This offset is useful as an input argument to *lseek*(2) or *fseek*(3).

HARDWARE DEPENDENCIES

Series 500:

The archive symbol table file contains the symbol table and a name pool of strings (the names of external symbols). This allows for symbols with arbitrarily long names. The **rl_hdr** structure defines the layout of the file, and the **rl_ref** structure defines the contents of an archive symbol table entry. These structures have the following format:

```
struct rl_hdr {
        long int rl_tcbas;              /* offset of table */
        long int rl_tclen;              /* length of table */
        long int rl_nmbas;              /* offset of name pool */
        long int rl_nmlen;              /* length of name pool */
};

struct rl_ref {
        long int name_pos;              /* index into name pool */
        long int lib_pos;               /* offset of defining file */
};
```

Series 200/300:

The archive symbol table file contains a header, a name pool of strings (the names of external symbols), and the archive symbol table. This allows for symbols with arbitrarily long names. The header contains a short integer which specifies the number of entries, and a long integer which specifies the size of the string table. Following this is the name pool. The last section of the file contains the archive symbol table entries. The structure of these entries is defined below:

```
typedef long off_t;

struct      ranlib {
            union {
                    off_t ran_strx;         /* string table index */
                    char *ran_name;
            } ran_un;
            off_t   ran_off;                /* lib member offset */
};
```

SEE ALSO

ar(1), ld(1), ar(4).

NAME

sccsfile – format of SCCS file

DESCRIPTION

An SCCS file is an ASCII file. It consists of six logical parts: the ***checksum***, the ***delta table*** (contains information about each delta), ***user names*** (contains login names and/or numerical group IDs of users who may add deltas), ***flags*** (contains definitions of internal keywords), ***comments*** (contains arbitrary descriptive information about the file), and the ***body*** (contains the actual text lines intermixed with control lines).

Throughout an SCCS file there are lines which begin with the **ASCII SOH** (start of heading) character (octal 001). This character is hereafter referred to as ***the control character*** and will be represented graphically as @. Any line described below which is not depicted as beginning with the control character is prevented from beginning with the control character.

Entries of the form **DDDDD** represent a five-digit string (a number between 00000 and 99999).

Each logical part of an SCCS file is described in detail below.

Checksum

The checksum is the first line of an SCCS file. The form of the line is:

```
@hDDDDD
```

The value of the checksum is the sum of all characters, except those of the first line. The **@h** provides a *magic number* consisting of the two bytes 0x01 and 0x68. (Other versions of the UNIX operating system usually use this same value but it may be displayed or documented as a single number with a different byte order.)

Delta table

The delta table consists of a variable number of entries of the form:

```
@s DDDDD/DDDDD/DDDDD
@d <type> <SCCS ID> yr/mo/da hr:mi:se <pgmr> DDDDD DDDDD
@i DDDDD ...
@x DDDDD ...
@g DDDDD ...
@m <MR number>
  .
  .
  .
@c <comments> ...
  .
  .
  .
@e
```

The first line (**@s**) contains the number of lines inserted/deleted/unchanged, respectively. The second line (**@d**) contains the type of the delta (currently, normal: **D**, and removed: **R**), the SCCS ID of the delta, the date and time of creation of the delta, the login name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor, respectively.

The **@i**, **@x**, and **@g** lines contain the serial numbers of deltas included, excluded, and ignored, respectively. These lines are optional.

The **@m** lines (optional) each contain one **MR** number associated with the delta; the **@c** lines contain comments associated with the delta.

The @**e** line ends the delta table entry.

User names

The list of login names and/or numerical group IDs of users who may add deltas to the file, separated by new-lines. The lines containing these login names and/or numerical group IDs are surrounded by the bracketing lines @**u** and @**U**. An empty list allows anyone to make a delta. Any line starting with a **!** prohibits the succeeding group or user from making deltas.

Flags

Keywords used internally (see *admin*(1) for more information on their use). Each flag line takes the form:

```
@f <flag>        <optional text>
```

The following flags are defined:

```
@f t    <type of program>
@f v    <program name>
@f i    <keyword string>
@f b
@f m    <module name>
@f f    <floor>
@f c    <ceiling>
@f d    <default-sid>
@f n
@f j
@f l    <lock-releases>
@f q    <user defined>
@f z    <reserved for use in interfaces>
```

The **t** flag defines the replacement for the **%Y%** identification keyword. The **v** flag controls prompting for **MR** numbers in addition to comments; if the optional text is present it defines an **MR** number validity checking program. The **i** flag controls the warning/error aspect of the "No id keywords" message. When the **i** flag is not present, this message is only a warning; when the **i** flag is present, this message will cause a "fatal" error (the file will not be gotten, or the delta will not be made). When the **b** flag is present the **-b** keyletter may be used on the *get* command to cause a branch in the delta tree. The **m** flag defines the first choice for the replacement text of the **%M%** identification keyword. The **f** flag defines the "floor" release; the release below which no deltas may be added. The **c** flag defines the "ceiling" release; the release above which no deltas may be added. The **d** flag defines the default SID to be used when none is specified on a *get* command. The **n** flag causes *delta* to insert a "null" delta (a delta that applies *no* changes) in those releases that are skipped when a delta is made in a *new* release (e.g., when delta 5.1 is made after delta 2.7, releases 3 and 4 are skipped). The absence of the **n** flag causes skipped releases to be completely empty. The **j** flag causes *get* to allow concurrent edits of the same base SID. The **l** flag defines a *list* of releases that are *locked* against editing (*get*(1) with the **-e** keyletter). The **q** flag defines the replacement for the **%Q%** identification keyword. The **z** flag is used in certain specialized interface programs.

Comments

Arbitrary text is surrounded by the bracketing lines @**t** and @**T**. The comments section typically will contain a description of the file's purpose.

Body

The body consists of text lines and control lines. Text lines do not begin with the control character, control lines do. There are three kinds of control lines: ***insert***, ***delete***, and ***end***, represented by:

@I DDDDD
@D DDDDD
@E DDDDD

respectively. The digit string is the serial number corresponding to the delta for the control line.

SEE ALSO

admin(1), delta(1), get(1), prs(1).

NAME

sdf – structured directory format description

DESCRIPTION

SDF (Structured Directory Format) is the name given to the format of mounted media used by HP 9000 series 500 HP-UX. This format is based upon the format used in the series 500 Basic workstations.

The utilities listed under SEE ALSO below are provided for non-Series 500 access to the SDF media. These utilities read and write data to and from SDF volumes, as well as retrieve information from an SDF volume.

The SDF utilities listed below are the only HP-UX utilities that recognize the internal contents of an SDF volume. To the rest of HP-UX, an SDF volume is simply a file containing unspecified data. Therefore, to access SDF media on any HP-UX system other than the series 500, *mount*(1) cannot be used because the operating system cannot recognize it.

SDF file names are specified to the SDF utilities by concatenating the HP-UX path name for the SDF volume with the SDF file name, separating the two with a colon (:). For example,

/dev/rdsk/c5d1s2:/users/ivy specifies SDF file **/users/ivy** within HP-UX device special file **/dev/rdsk/c5d1s2**

Note that this file naming convention is applicable only for use as arguments to the SDF utilities and does not constitute a legal path name for any other use within HP-UX. The shell "wild-card" characters *, ?, and [...] do not work for specifying an arbitrary pattern for matching SDF file names when using the SDF utilities.

If the device name and a trailing colon are specified *without* a file or directory name following (for example **/dev/rdsk/c5d1s2:**), then the root (/) of the SDF file system is assumed by convention.

Files cannot be created with the SDF utilities unless there is at least one free block of storage on the device.

Although Shared Resource Management (SRM) storage media implement the SDF file system, Hewlett-Packard does not support the use of the SDF utilities on SRM workstation storage media.

WARNINGS

The SDF utilities are intended to be run on non-Series 500 HP UX systems. If the SDF utilities are executed on a Series 500, however, *you are cautioned to not run them on a disk that has a mounted file system on it.*

AUTHOR

Sdf(4) was developed by the Hewlett-Packard Company.

FILES

/tmp/SDF..LCK	lock file for single user access

SEE ALSO

sdfchmod(1), sdfchown(1), sdfcp(1), sdfdf(1M), sdffind(1), sdffsck(1M), sdffsdb(1M), sdfls(1), sdfmkdir(1), sdfrm(1).

NAME

term – format of compiled term file

SYNOPSIS

term

DESCRIPTION

Compiled terminfo descriptions are placed under the directory **/usr/lib/terminfo**. In order to avoid a linear search of a huge HP-UX system directory, a two-level scheme is used: **/usr/lib/terminfo/c/name** where *name* is the name of the terminal, and *c* is the first character of *name*. Thus, *act4* can be found in the file **/usr/lib/terminfo/a/act4**. Synonyms for the same terminal are implemented by multiple links to the same compiled file.

The format has been chosen so that it will be the same on all hardware. An 8 or more bit byte is assumed, but no assumptions about byte ordering or sign extension are made.

The compiled file is created with the *tic* program, and read by the routine *setupterm*. Both of these pieces of software are part of *curses*(3X). The file is divided into six parts: the header, terminal names, boolean flags, numbers, strings, and string table.

The header section begins the file. This section contains six short integers in the format described below. These integers are (1) the magic number (octal 0432); (2) the size, in bytes, of the names section; (3) the number of bytes in the boolean section; (4) the number of short integers in the numbers section; (5) the number of offsets (short integers) in the strings section; (6) the size, in bytes, of the string table.

Short integers are stored in two 8-bit bytes. The first byte contains the least significant 8 bits of the value, and the second byte contains the most significant 8 bits. (Thus, the value represented is 256*second+first.) The value –1 is represented by 0377, 0377, other negative value are illegal. The –1 generally means that a capability is missing from this terminal. Note that this format corresponds to the hardware of the VAX and PDP-11. Machines where this does not correspond to the hardware read the integers as two bytes and compute the result.

The terminal names section comes next. It contains the first line of the terminfo description, listing the various names for the terminal, separated by the '|' character. The section is terminated with an ASCII NUL character.

The boolean flags have one byte for each flag. This byte is either 0 or 1 as the flag is present or absent. The capabilities are in the same order as the file <term.h>.

Between the boolean section and the number section, a null byte will be inserted, if necessary, to ensure that the number section begins on an even byte. All short integers are aligned on a short word boundary.

The numbers section is similar to the flags section. Each capability takes up two bytes, and is stored as a short integer. If the value represented is –1, the capability is taken to be missing.

The strings section is also similar. Each capability is stored as a short integer, in the format above. A value of –1 means the capability is missing. Otherwise, the value is taken as an offset from the beginning of the string table. Special characters in ^X or \c notation are stored in their interpreted form, not the printing representation. Padding information $<nn> and parameter information %x are stored intact in uninterpreted form.

The final section is the string table. It contains all the values of string capabilities referenced in the string section. Each string is null terminated.

Note that it is possible for *setupterm* to expect a different set of capabilities than are actually present in the file. Either the database may have been updated since *setupterm* has been recompiled (resulting in extra unrecognized entries in the file) or the program may have been recompiled more recently than the database was updated (resulting in missing entries). The routine *setupterm* must be prepared for both possibilities – this is why the numbers and sizes are included.

Also, new capabilities must always be added at the end of the lists of boolean, number, and string capabilities.

As an example, an octal dump of the description for the Microterm ACT 4 is included:

```
microterm|act4|microterm act iv,
    cr=^M, cud1=^J, ind=^J, bel=^G, am, cub1=^H,
    ed=^_, el=^^, clear=^L, cup=^T%p1%c%p2%c,
    cols#80, lines#24, cuf1=^X, cuu1=^Z, home=^],
```

```
000  032 001      \0 025  \0  \b  \0 212  \0   "  \0   m   i   c   r
020    o   t   e   r   m   |   a   c   t   4   |   m   i   c   r   o
040    t   e   r   m       a   c   t       i   v  \0  \0 001  \0  \0
060   \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
100   \0  \0   P  \0 377 377 030  \0 377 377 377 377 377 377 377 377
120  377 377 377 377  \0  \0 002  \0 377 377 377 377 004  \0 006  \0
140   \b  \0 377 377 377 377  \n  \0 026  \0 030  \0 377 377 032  \0
160  377 377 377 377 034  \0 377 377 036  \0 377 377 377 377 377 377
200  377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
*
520  377 377 377 377      \0 377 377 377 377 377 377 377 377 377 377
540  377 377 377 377 377 377 007  \0  \r  \0  \f  \0 036  \0 037  \0
560  024   %   p   1   %   c   %   p   2   %   c  \0  \n  \0 035  \0
600   \b  \0 030  \0 032  \0  \n  \0
```

Some limitations: total compiled entries cannot exceed 4096 bytes. The name field cannot exceed 128 bytes.

WARNINGS

HP only supports terminals listed on the current list of supported devices. However, non-supported and supported terminals can be in the terminfo database. If you use such non-supported terminals, they may not work correctly.

FILES

/usr/lib/terminfo/?/*	compiled terminal capability data base

SEE ALSO

curses(3X), terminfo(4), tic(1), untic(1M).

NAME

terminfo – terminal capability data base

SYNOPSIS

/usr/lib/terminfo/?/*

DESCRIPTION

Terminfo is a data base describing terminals used by, for example, *vi*(1) and *curses*(3X). Terminals are described in *terminfo* by giving a set of capabilities which they have, and by describing how operations are performed. Padding requirements and initialization sequences are included in *terminfo*.

Entries in *terminfo* consist of a number of ',' separated fields. White space after each ',' is ignored. The first entry for each terminal gives the names which are known for the terminal, separated by '|' characters. The first name given is the most common abbreviation for the terminal, the last name given should be a long name fully identifying the terminal, and all others are understood as synonyms for the terminal name. All names but the last should be in lower case and contain no blanks; the last name may well contain upper case and blanks for readability.

Terminal names (except for the last, verbose entry) should be chosen using the following conventions. The particular piece of hardware making up the terminal should have a root name chosen, thus "hp2621". This name should not contain hyphens, except that synonyms may be chosen that do not conflict with other names. Modes that the hardware can be in, or user preferences, should be indicated by appending a hyphen and an indicator of the mode. Thus, a vt100 in 132 column mode would be vt100-w. The following suffixes should be used where possible:

Suffix	**Meaning**	**Example**
-w	Wide mode (more than 80 columns)	vt100-w
-am	With auto. margins (usually default)	vt100-am
-nam	Without automatic margins	vt100-nam
-*n*	Number of lines on the screen	aaa-60
-na	No arrow keys (leave them in local)	c100-na
-*n*p	Number of pages of memory	c100-4p
-rv	Reverse video	c100-rv

CAPABILITIES

The variable is the name by which the programmer (at the terminfo level) accesses the capability. The capname is the short name used in the text of the database, and is used by a person updating the database. The i.code is the two letter internal code used in the compiled database, and always corresponds to the old **termcap** capability name.

Capability names have no hard length limit, but an informal limit of 5 characters has been adopted to keep them short and to allow the tabs in the source file **caps** to line up nicely. Whenever possible, names are chosen to be the same as or similar to the ANSI X3.64-1979 standard. Semantics are also intended to match those of the specification.

(P) indicates that padding may be specified

(G) indicates that the string is passed through tparm withparms as given (#*i*).

(*) indicates that padding may be based on the number of lines affected

($\#_i$) indicates the i^{th} parameter.

Variable	**Cap-**	**I.**	**Description**
Booleans	**name**	**Code**	
auto_left_margin,	bw	bw	cub1 wraps from column 0 to last column
auto_right_margin,	am	am	Terminal has automatic margins
beehive_glitch,	xsb	xb	Beehive (f1=escape, f2=ctrl C)

ceol_standout_glitch,	xhp	xs	Standout not erased by overwriting (hp)
eat_newline_glitch,	xenl	xn	newline ignored after 80 cols (Concept)
erase_overstrike,	eo	eo	Can erase overstrikes with a blank
generic_type,	gn	gn	Generic line type (e.g.,, dialup, switch).
hard_copy,	hc	hc	Hardcopy terminal
has_meta_key,	km	km	Has a meta key (shift, sets parity bit)
has_status_line,	hs	hs	Has extra "status line"
insert_null_glitch,	in	in	Insert mode distinguishes nulls
memory_above,	da	da	Display may be retained above the screen
memory_below,	db	db	Display may be retained below the screen
move_insert_mode,	mir	mi	Safe to move while in insert mode
move_standout_mode,	msgr	ms	Safe to move in standout modes
over_strike,	os	os	Terminal overstrikes
status_line_esc_ok,	eslok	es	Escape can be used on the status line
teleray_glitch,	xt	xt	Tabs ruin, magic so char (Teleray 1061)
tilde_glitch,	hz	hz	Hazeltine; can not print ~'s
transparent_underline,	ul	ul	underline character overstrikes
xon_xoff,	xon	xo	Terminal uses xon/xoff handshaking
Numbers:			
columns,	cols	co	Number of columns in a line
init_tabs,	it	it	Tabs initially every # spaces
lines,	lines	li	Number of lines on screen or page
lines_of_memory,	lm	lm	Lines of memory if > lines. 0 means varies
magic_cookie_glitch,	xmc	sg	Number of blank chars left by smso or rmso
padding_baud_rate,	pb	pb	Lowest baud where cr/nl padding is needed
virtual_terminal,	vt	vt	Virtual terminal number (HP-UX system)
width_status_line,	wsl	ws	No. columns in status line
Strings:			
back_tab,	cbt	bt	Back tab (P)
bell,	bel	bl	Audible signal (bell) (P)
carriage_return,	cr	cr	Carriage return (P*)
change_scroll_region,	csr	cs	change to lines #1 through #2 (vt100) (PG)
clear_all_tabs,	tbc	ct	Clear all tab stops (P)
clear_screen,	clear	cl	Clear screen and home cursor (P*)
clr_eol,	el	ce	Clear to end of line (P)
clr_eos,	ed	cd	Clear to end of display (P*)
column_address,	hpa	ch	Set cursor column (PG)
command_character,	cmdch	CC	Term. settable cmd char in prototype
cursor_address,	cup	cm	Screen rel. cursor motion row #1 col #2 (PG)

cursor_down,	cud1	do	Down one line
cursor_home,	home	ho	Home cursor (if no cup)
cursor_invisible,	civis	vi	Make cursor invisible
cursor_left,	cub1	le	Move cursor left one space
cursor_mem_address,	mrcup	CM	Memory relative cursor addressing
cursor_normal,	cnorm	ve	Make cursor appear normal (undo vs/vi)
cursor_right,	cuf1	nd	Non-destructive space (cursor right)
cursor_to_ll,	ll	ll	Last line, first column (if no cup)
cursor_up,	cuu1	up	Upline (cursor up)
cursor_visible,	cvvis	vs	Make cursor very visible
delete_character,	dch1	dc	Delete character (P*)
delete_line,	dl1	dl	Delete line (P*)
dis_status_line,	dsl	ds	Disable status line
down_half_line,	hd	hd	Half-line down (forward 1/2 linefeed)
enter_alt_charset_mode,	smacs	as	Start alternate character set (P)
enter_blink_mode,	blink	mb	Turn on blinking
enter_bold_mode,	bold	md	Turn on bold (extra bright) mode
enter_ca_mode,	smcup	ti	String to begin programs that use cup
enter_delete_mode,	smdc	dm	Delete mode (enter)
enter_dim_mode,	dim	mh	Turn on half-bright mode
enter_insert_mode,	smir	im	Insert mode (enter);
enter_protected_mode,	prot	mp	Turn on protected mode
enter_reverse_mode,	rev	mr	Turn on reverse video mode
enter_secure_mode,	invis	mk	Turn on blank mode (chars invisible)
enter_standout_mode,	smso	so	Begin stand out mode
enter_underline_mode,	smul	us	Start underscore mode
erase_chars	ech	ec	Erase #1 characters (PG)
exit_alt_charset_mode,	rmacs	ae	End alternate character set (P)
exit_attribute_mode,	sgr0	me	Turn off all attributes
exit_ca_mode,	rmcup	te	String to end programs that use cup
exit_delete_mode,	rmdc	ed	End delete mode
exit_insert_mode,	rmir	ei	End insert mode
exit_standout_mode,	rmso	se	End stand out mode
exit_underline_mode,	rmul	ue	End underscore mode
flash_screen,	flash	vb	Visible bell (may not move cursor)
form_feed,	ff	ff	Hardcopy terminal page eject (P*)
from_status_line,	fsl	fs	Return from status line
init_1string,	is1	i1	Terminal initialization string
init_2string,	is2	i2	Terminal initialization string
init_3string,	is3	i3	Terminal initialization string
init_file,	if	if	Name of file containing is
insert_character,	ich1	ic	Insert character (P)
insert_line,	il1	al	Add new blank line (P*)
insert_padding,	ip	ip	Insert pad after character inserted (p*)
key_backspace,	kbs	kb	Sent by backspace key
key_catab,	ktbc	ka	Sent by clear-all-tabs key
key_clear,	kclr	kC	Sent by clear screen or erase key
key_ctab,	kctab	kt	Sent by clear-tab key
key_dc,	kdch1	kD	Sent by delete character key
key_dl,	kdl1	kL	Sent by delete line key
key_down,	kcud1	kd	Sent by terminal down arrow key
key_eic,	krmir	kM	Sent by rmir or smir in insert mode

key_eol,	kel	kE	Sent by clear-to-end-of-line key
key_eos,	ked	kS	Sent by clear-to-end-of-screen key
key_f0,	kf0	k0	Sent by function key f0
key_f1,	kf1	k1	Sent by function key f1
key_f10,	kf10	ka	Sent by function key f10
key_f2,	kf2	k2	Sent by function key f2
key_f3,	kf3	k3	Sent by function key f3
key_f4,	kf4	k4	Sent by function key f4
key_f5,	kf5	k5	Sent by function key f5
key_f6,	kf6	k6	Sent by function key f6
key_f7,	kf7	k7	Sent by function key f7
key_f8,	kf8	k8	Sent by function key f8
key_f9,	kf9	k9	Sent by function key f9
key_home,	khome	kh	Sent by home key
key_ic,	kich1	kI	Sent by ins char/enter ins mode key
key_il,	kil1	kA	Sent by insert line
key_left,	kcub1	kl	Sent by terminal left arrow key
key_ll,	kll	kH	Sent by home-down key
key_npage,	knp	kN	Sent by next-page key
key_ppage,	kpp	kP	Sent by previous-page key
key_right,	kcuf1	kr	Sent by terminal right arrow key
key_sf,	kind	kF	Sent by scroll-forward/down key
key_sr,	kri	kR	Sent by scroll-backward/up key
key_stab,	khts	kT	Sent by set-tab key
key_up,	kcuu1	ku	Sent by terminal up arrow key
keypad_local,	rmkx	ke	Out of "keypad transmit" mode
keypad_xmit,	smkx	ks	Put terminal in "keypad transmit" mode
lab_f0,	lf0	l0	Labels on function key f0 if not f0
lab_f1,	lf1	l1	Labels on function key f1 if not f1
lab_f10,	lf10	la	Labels on function key f10 if not f10
lab_f2,	lf2	l2	Labels on function key f2 if not f2
lab_f3,	lf3	l3	Labels on function key f3 if not f3
lab_f4,	lf4	l4	Labels on function key f4 if not f4
lab_f5,	lf5	l5	Labels on function key f5 if not f5
lab_f6,	lf6	l6	Labels on function key f6 if not f6
lab_f7,	lf7	l7	Labels on function key f7 if not f7
lab_f8,	lf8	l8	Labels on function key f8 if not f8
lab_f9,	lf9	l9	Labels on function key f9 if not f9
memory_lock,	meml	ml	Lock memory above cursor
memory_unlock,	memu	mu	Turn memory lock off
meta_on,	smm	mm	Turn on "meta mode" (8th bit)
meta_off,	rmm	mo	Turn off "meta mode"
newline,	nel	nw	Newline (behaves like cr followed by lf)
pad_char,	pad	pc	Pad character (rather than null)
parm_dch,	dch	DC	Delete #1 chars (PG*)
parm_delete_line,	dl	DL	Delete #1 lines (PG*)
parm_down_cursor,	cud	DO	Move cursor down #1 lines (PG*)
parm_ich,	ich	IC	Insert #1 blank chars (PG*)
parm_index,	indn	SF	Scroll forward #1 lines (PG)
parm_insert_line,	il	AL	Add #1 new blank lines (PG*)
parm_left_cursor,	cub	LE	Move cursor left #1 spaces (PG)
parm_right_cursor,	cuf	RI	Move cursor right #1 spaces (PG*)

parm_rindex,	rin	SR	Scroll backward #1 lines (PG)
parm_up_cursor,	cuu	UP	Move cursor up #1 lines (PG*)
pkey_key,	pfkey	pk	Prog funct key #1 to type string #2
pkey_local,	pfloc	pl	Prog funct key #1 to execute string #2
pkey_xmit,	pfx	px	Prog funct key #1 to xmit string #2
print_screen,	mc0	ps	Print contents of the screen
prtr_off,	mc4	pf	Turn off the printer
prtr_on,	mc5	po	Turn on the printer
repeat_char,	rep	rp	Repeat char #1 #2 times. (PG*)
reset_1string,	rs1	r1	Reset terminal completely to sane modes.
reset_2string,	rs2	r2	Reset terminal completely to sane modes.
reset_3string,	rs3	r3	Reset terminal completely to sane modes.
reset_file,	rf	rf	Name of file containing reset string
restore_cursor,	rc	rc	Restore cursor to position of last sc
row_address,	vpa	cv	Vertical position absolute (set row) (PG)
save_cursor,	sc	sc	Save cursor position (P)
scroll_forward,	ind	sf	Scroll text up (P)
scroll_reverse,	ri	sr	Scroll text down (P)
set_attributes,	sgr	sa	Define the video attributes (PG9)
set_tab,	hts	st	Set a tab in all rows, current column
set_window,	wind	wi	Current window is lines #1-#2 cols #3-#4
tab,	ht	ta	Tab to next 8 space hardware tab stop
to_status_line,	tsl	ts	Go to status line, column #1
underline_char,	uc	uc	Underscore one char and move past it
up_half_line,	hu	hu	Half-line up (reverse 1/2 linefeed)
init_prog,	iprog	iP	Path name of program for init
key_a1,	ka1	K1	Upper left of keypad
key_a3,	ka3	K3	Upper right of keypad
key_b2,	kb2	K2	Center of keypad
key_c1,	kc1	K4	Lower left of keypad
key_c3,	kc3	K5	Lower right of keypad
prtr_non,	mc5p	pO	Turn on the printer for #1 bytes

A Sample Entry

The following entry, which describes the Concept–100, is among the more complex entries in the *terminfo* file as of this writing.

```
concept100 | c100| concept | c104 | c100-4p | concept 100,
          am, bel=^G, blank=\EH, blink=\EC, clear=^L$<2*>, cnorm=\Ew,
          cols#80, cr=^M$<9>, cub1=^H, cud1=^J, cuf1=\E=,
          cup=\Ea%p1%' '%+%c%p2%' '%+%c,
          cuu1=\E;, cvvis=\EW, db, dch1=\E^A$<16*>, dim=\EE, dl1=\E^B$<3*>,
          ed=\E^C$<16*>, el=\E^U$<16>, eo, flash=\Ek$<20>\EK, ht=\t$<8>,
          il1=\E^R$<3*>, in, ind=^J, .ind=^J$<9>, ip=$<16*>,
          is2=\EU\Ef\E7\E5\E8\El\ENH\EK\E\200\Eo&\200\Eo\47\E,
          kbs=^h, kcub1=\E>, kcud1=\E<, kcuf1=\E=, kcuu1=\E;,
          kf1=\E5, kf2=\E6, kf3=\E7, khome=\E?,
          lines#24, mir, pb#9600, prot=\EI, rep=\Er%p1%c%p2%' '%+%c$<.2*>,
          rev=\ED, rmcup=\Ev    $<6>\Ep\r\n, rmir=\E\200, rmkx=\Ex,
          rmso=\Ed\Ee, rmul=\Eg, rmul=\Eg, sgr0=\EN\200,
          smcup=\EU\Ev  8p\Ep\r, smir=\E^P, smkx=\EX, smso=\EE\ED,
          smul=\EG, tabs, ul, vt#8, xenl,
```

Entries may continue onto multiple lines by placing white space at the beginning of each line except the first. Comments may be included on lines beginning with "#". Capabilities in *terminfo* are of three types: Boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or the size of particular delays, and string capabilities, which give a sequence which can be used to perform particular terminal operations.

Types of Capabilities

All capabilities have names. For instance, the fact that the Concept has *automatic margins* (i.e., an automatic return and linefeed when the end of a line is reached) is indicated by the capability **am**. Hence the description of the Concept includes **am**. Numeric capabilities are followed by the character '#' and then the value. Thus **cols**, which indicates the number of columns the terminal has, gives the value '80' for the Concept.

Finally, string valued capabilities, such as **el** (clear to end of line sequence) are given by the two-character code, an '=', and then a string ending at the next following ','. A delay in milliseconds may appear anywhere in such a capability, enclosed in $<..> brackets, as in **el**=\EK$<3>, and padding characters are supplied by *tputs* to provide this delay. The delay can be either a number, e.g., '20', or a number followed by an '*', i.e., '3*'. A '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. (In the case of insert character, the factor is still the number of *lines* affected. This is always one unless the terminal has **xenl** and the software uses it.) When a '*' is specified, it is sometimes useful to give a delay of the form '3.5' to specify a delay per unit to tenths of milliseconds. (Only one decimal place is allowed.)

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. Both **\E** and **\e** map to an ESCAPE character, **^x** maps to a control-x for any appropriate x, and the sequences **\n** **\l** **\r** **\t** **\b** **\f** **\s** give a newline, linefeed, return, tab, backspace, formfeed, and space. Other escapes include \^ for ^, \\ for \, \, for comma, \: for :, and \0 for null. (\0 will produce \200, which does not terminate a string but behaves as a null character on most terminals.) Finally, characters may be given as three octal digits after a ****.

Sometimes individual capabilities must be commented out. To do this, put a period before the capability name. For example, see the second **ind** in the example above.

Preparing Descriptions

We now outline how to prepare descriptions of terminals. The most effective way to prepare a terminal description is by imitating the description of a similar terminal in *terminfo* and to build up a description gradually, using partial descriptions with *vi* to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the *terminfo* file to describe it or bugs in *vi*. To easily test a new terminal description you can set the environment variable TERMINFO to a pathname of a directory containing the compiled description you are working on and programs will look there rather than in */usr/lib/terminfo*. To get the padding for insert line right (if the terminal manufacturer did not document it) a severe test is to edit /etc/passwd at 9600 baud, delete 16 or so lines from the middle of the screen, then hit the 'u' key several times quickly. If the terminal messes up, more padding is usually needed. A similar test can be used for insert character.

Basic Capabilities

The number of columns on each line for the terminal is given by the **cols** numeric capability. If the terminal is a CRT, then the number of lines on the screen is given by the **lines** capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it should have the **am** capability. If the terminal can clear its screen, leaving the cursor in the home position, then this is given by the **clear** string capability. If the terminal overstrikes (rather than clearing a position when a character is struck over) then it should have the **os** capability. If the terminal is a printing terminal, with no soft copy unit, give it both **hc** and **os**. (**os** applies to storage scope terminals, such as TEKTRONIX 4010 series, as well as hard copy and APL terminals.) If there is a code to move the cursor to the left edge of the current row, give this as **cr**. (Normally this will be carriage return, control M.) If there is a code to produce an audible signal (bell, beep, etc) give this as **bel**.

If there is a code to move the cursor one position to the left (such as backspace) that capability should be given as **cub1**. Similarly, codes to move to the right, up, and down should be given as **cuf1**, **cuu1**, and **cud1**. These local cursor motions should not alter the text they pass over, for example, you would not normally use '**cuf1=** ' because the space would erase the character moved over.

A very important point here is that the local cursor motions encoded in *terminfo* are undefined at the left and top edges of a CRT terminal. Programs should never attempt to backspace around the left edge, unless **bw** is given, and never attempt to go up locally off the top. In order to scroll text up, a program will go to the bottom left corner of the screen and send the **ind** (index) string.

To scroll text down, a program goes to the top left corner of the screen and sends the **ri** (reverse index) string. The strings **ind** and **ri** are undefined when not on their respective corners of the screen.

Parameterized versions of the scrolling sequences are **indn** and **rin** which have the same semantics as **ind** and **ri** except that they take one parameter, and scroll that many lines. They are also undefined except at the appropriate edge of the screen.

The **am** capability tells whether the cursor sticks at the right edge of the screen when text is output, but this does not necessarily apply to a **cuf1** from the last column. The only local motion which is defined from the left edge is if **bw** is given, then a **cub1** from the left edge will move to the right edge of the previous row. If **bw** is not given, the effect is undefined. This is useful for drawing a box around the edge of the screen, for example. If the terminal has switch selectable automatic margins, the *terminfo* file usually assumes that this is on; i.e., **am**. If the terminal has a command which moves to the first column of the next line, that command can be given as **nel** (newline). It does not matter if the command clears the remainder of the current line, so if the terminal has no **cr** and **lf** it may still be possible to craft a working **nel** out of one or both of them.

These capabilities suffice to describe hardcopy and glass-tty terminals. Thus the model 33 teletype is described as

```
33 | tty33 | tty | model 33 teletype,
bel=^G, cols#72, cr=^M, cud1=^J, hc, ind=^J, os,
```

while the Lear Siegler ADM-3 is described as

```
adm3 | 3 | lsi adm3,
am, bel=^G, clear=^Z, cols#80, cr=^M, cub1=^H, cud1=^J,
ind=^J, lines#24,
```

Parameterized Strings

Cursor addressing and other strings requiring parameters in the terminal are described by a parameterized string capability, with *printf*(3S) like escapes **%x** in it. For example, to address the cursor, the **cup** capability is given, using two parameters: the row and column to address to. (Rows and columns are numbered from zero and refer to the physical screen visible to the user, not to any unseen memory.) If the terminal has memory relative cursor addressing, that can be indicated by **mrcup**.

The parameter mechanism uses a stack and special **%** codes to manipulate it. Typically a sequence will push one of the parameters onto the stack and then print it in some format. Often more complex operations are necessary.

The **%** encodings have the following meanings:

%%	outputs '%'
%d	print pop() as in printf
%2d	print pop() like %2d
%3d	print pop() like %3d
%02d	
%03d	as in printf
%c	print pop() gives %c
%s	print pop() gives %s
%p[1-9]	push ith parm
%P[a-z]	set variable [a-z] to pop()
%g[a-z]	get variable [a-z] and push it
%'c'	char constant c
%{nn}	integer constant nn
%+ %- %* %/ %m	arithmetic (%m is mod): push(pop() op pop())
%& %| %^	bit operations: push(pop() op pop())
%= %> %<	logical operations: push(pop() op pop())
%! %~	unary operations push(op pop())
%i	add 1 to first two parms (for ANSI terminals)
%? expr %t thenpart %e elsepart %;	if-then-else, %e elsepart is optional. else-if's are possible ala Algol 68: %? c_1 %t b_1 %e c_2 %t b_2 %e c_3 %t b_3 %e c_4 %t b_4 %e %; c_i are conditions, b_i are bodies.

Binary operations are in postfix form with the operands in the usual order. That is, to get x-5 one would use "%gx%{5}%-".

Consider the HP2645, which, to get to row 3 and column 12, needs to be sent \E&a12c03Y padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are printed as two digits. Thus its **cup** capability is cup=6\E&%p2%2dc%p1%2dY.

The Microterm ACT-IV needs the current row and column sent preceded by a **^T**, with the row and column simply encoded in binary, cup=^T%p1%c%p2%c. Terminals which use %c need to be able to backspace the cursor (**cub1**), and to move the cursor up one line on the screen (**cuu1**). This is necessary because it is not always safe to transmit **\n** **^D** and **\r**, as the system may change or discard them. (The library routines dealing with terminfo set tty modes so that tabs are never expanded, so \t is safe to send. This turns out to be essential for the Ann Arbor 4080.)

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus cup=\E=%p1%' '%+%c%p2%' '%+%c. After sending '\E=', this pushes the first parameter, pushes the ASCII value for a space (32), adds them (pushing the sum on the stack in place of the two previous values) and outputs that value as a character. Then the same is done for the second parameter. More complex arithmetic is possible using the stack.

If the terminal has row or column absolute cursor addressing, these can be given as single parameter capabilities **hpa** (horizontal position absolute) and **vpa** (vertical position absolute). Sometimes these are shorter than the more general two parameter sequence (as with the hp2645) and can be used in preference to **cup** . If there are parameterized local motions (e.g., move *n* spaces to the right) these can be given as **cud**, **cub**, **cuf**, and **cuu** with a single parameter indicating how many spaces to move. These are primarily useful if the terminal does not have **cup**, such as the TEKTRONIX 4025.

Cursor Motions

If the terminal has a fast way to home the cursor (to very upper left corner of screen) then this can be given as **home**; similarly a fast way of getting to the lower left-hand corner can be given as **ll**; this may involve going up with **cuu1** from the home position, but a program should never do this itself (unless **ll** does) because it can make no assumption about the effect of moving up from the home position. Note that the home position is the same as addressing to (0,0): to the top left corner of the screen, not of memory. (Thus, the \EH sequence on HP terminals cannot be used for **home**.)

Area Clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **el**. If the terminal can clear from the current position to the end of the display, then this should be given as **ed**. **Ed** is only defined from the first column of a line. (Thus, it can be simulated by a request to delete a large number of lines, if a true **ed** is not available.)

Insert/delete line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **il1**; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as **dl1**; this is done only from the first position on the line to be deleted. Versions of **il1** and **dl1** which take a single parameter and insert or delete that many lines can be given as **il** and **dl**. If the terminal has a settable scrolling region (like the vt100) the command to set this can be described with the **csr** capability, which takes two parameters: the top and bottom lines of the scrolling region. The cursor position is, alas, undefined after using this command. It is possible to get the effect of insert or delete line using this command - the **sc** and **rc** (save and restore cursor) commands are also useful. Inserting lines at the top or bottom of the screen can also be done using **ri** or **ind** on many terminals without a true insert/delete line, and is often faster even on terminals with those features.

If the terminal has the ability to define a window as part of memory, which all commands affect, it should be given as the parameterized string **wind**. The four parameters are the starting and ending lines in memory and the starting and ending columns in memory, in that order.

If the terminal can retain display memory above, then the **da** capability should be given; if display memory can be retained below, then **db** should be given. These indicate that deleting a line or scrolling may bring non-blank lines up from below or that scrolling back with **ri** may bring down non-blank lines.

Insert/Delete Character

There are two basic kinds of intelligent terminals with respect to insert/delete character which can be described using *terminfo*. The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can determine the kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type abc   def using local cursor motions (not spaces) between the abc and the def. Then position the cursor before the abc and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the abc shifts over to the def which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for insert null. While these are two logically separate attributes (one line vs. multiline insert mode, and special treatment of untyped spaces) we have seen no terminals whose insert mode cannot be described with the single attribute.

Terminfo can describe both terminals which have an insert mode, and terminals which send a simple sequence to open a blank position on the current line. Give as **smir** the sequence to get into insert mode. Give as **rmir** the sequence to leave insert mode. Now give as **ich1** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ich1**; terminals which send a sequence to open a screen position should give it here. (If your terminal has both, insert mode is usually preferable to **ich1**. Do not give both unless the terminal actually requires both to be used in combination.) If post insert padding is needed, give this as a number of milliseconds in **ip** (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**. If your terminal needs both to be placed into an 'insert mode' and a special code to precede each inserted character, then both **smir/rmir** and **ich1** can be given, and both will be used. The **ich** capability, with one parameter, *n*, will repeat the effects of **ich1** *n* times.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g., if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability **mir** to speed up inserting in this case. Omitting **mir** will affect only speed. Some terminals (notably Datamedia's) must not have **mir** because of the way their insert mode works.

Finally, you can specify **dch1** to delete a single character, **dch** with one parameter, *n*, to delete *n characters,* and delete mode by giving **smdc** and **rmdc** to enter and exit delete mode (any mode the terminal needs to be placed in for **dch1** to work).

A command to erase *n* characters (equivalent to outputting *n* blanks without moving the cursor) can be given as **ech** with one parameter.

Highlighting, Underlining, and Visible Bells

If your terminal has one or more kinds of display attributes, these can be represented in a number of different ways. You should choose one display form as *standout mode*, representing a good, high contrast, easy-on-the-eyes, format for highlighting error messages and other attention getters. (If you have a choice, reverse video plus half-bright is good, or reverse video alone.) The sequences to enter and exit standout mode are given as **smso** and **rmso**, respectively. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, then **xmc** should be given to tell how many spaces are left.

Codes to begin underlining and end underlining can be given as **smul** and **rmul** respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Microterm Mime, this can be given as **uc**.

Other capabilities to enter various highlighting modes include **blink** (blinking) **bold** (bold or extra bright) **dim** (dim or half-bright) **invis** (blanking or invisible text) **prot** (protected) **rev** (reverse video) **sgr0** (turn off *all* attribute modes) **smacs** (enter alternate character set mode) and **rmacs** (exit alternate character set mode). Turning on any of these modes singly may or may not turn off other modes.

If there is a sequence to set arbitrary combinations of modes, this should be given as **sgr** (set attributes), taking 9 parameters. Each parameter is either 0 or 1, as the corresponding attribute is on or off. The 9 parameters are, in order: standout, underline, reverse, blink, dim, bold, blank, protect, alternate character set. Not all modes need be supported by **sgr**, only those for which corresponding separate attribute commands exist.

Terminals with the "magic cookie" glitch (**xmc**) deposit special "cookies" when they receive mode-setting sequences, which affect the display algorithm rather than having extra bits for each character. Some terminals, such as the HP 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit standout mode before moving the cursor or sending a newline, unless the **msgr** capability, asserting that it is safe to move in standout mode, is present.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement) then this can be given as **flash**; it must not move the cursor.

If the cursor needs to be made more visible than normal when it is not on the bottom line (to make, for example, a non-blinking underline into an easier to find block or blinking underline) give this sequence as **cvvis**. If there is a way to make the cursor completely invisible, give that as **civis**. The capability **cnorm** should be given which undoes the effects of both of these modes.

If the terminal needs to be in a special mode when running a program that uses these capabilities, the codes to enter and exit this mode can be given as **smcup** and **rmcup**. This arises, for example, from terminals like the Concept with more than one page of memory. If the terminal has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly. This is also used for the TEKTRONIX 4025, where **smcup** sets the command character to be the one used by terminfo.

If your terminal correctly generates underlined characters (with no special codes needed) even though it does not overstrike, then you should give the capability **ul**. If overstrikes are erasable with a blank, then this should be indicated by giving **eo**.

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted HP 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **smkx** and **rmkx**. Otherwise the keypad is assumed to always transmit. The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kcub1, kcuf1, kcuu1, kcud1,** and **khome** respectively. If there are function keys such as f0, f1, ..., f10, the codes they send can be given as **kf0, kf1, ..., kf10**. If these keys have labels other than the default f0 through f10, the labels can be given as **lf0, lf1, ..., lf10**. The codes transmitted by certain other special keys can be given: **kll** (home down), **kbs** (backspace), **ktbc** (clear all tabs), **kctab** (clear the tab stop in this column), **kclr** (clear screen or erase key), **kdch1** (delete character), **kdl1** (delete line), **krmir** (exit insert mode), **kel** (clear to end of line), **ked** (clear to end of screen), **kich1** (insert character or enter insert mode), **kil1** (insert line), **knp** (next page), **kpp** (previous page), **kind** (scroll forward/down), **kri** (scroll backward/up), **khts** (set a tab stop in this column). In addition, if the keypad has a 3 by 3 array of keys including the four arrow keys, the other five keys can be given as **ka1**, **ka3**, **kb2**, **kc1**, and **kc3**. These keys are useful when the effects of a 3 by 3 directional pad are needed.

Tabs and Initialization

If the terminal has hardware tabs, the command to advance to the next tab stop can be given as **ht** (usually control I). A "backtab" command which moves leftward to the next tab stop can be given as **cbt**. By convention, if the teletype modes indicate that tabs are being expanded by the computer rather than being sent to the terminal, programs should not use **ht** or **cbt** even if they are present, since the user may not have the tab stops properly set. If the terminal has hardware tabs which are initially set every *n* spaces when the terminal is powered up, the numeric parameter **it** is given, showing the number of spaces the tabs are set to. This is normally used by the *tset* command to determine whether to set the mode for hardware tab expansion, and whether to set the tab stops. If the terminal has tab stops that can be saved in nonvolatile memory, the terminfo description can assume that they are properly set.

Other capabilities include **is1**, **is2**, and **is3**, initialization strings for the terminal, **iprog**, the path name of a program to be run to initialize the terminal, and **if**, the name of a file containing long initialization strings. These strings are expected to set the terminal into modes consistent with the rest of the terminfo description. They are normally sent to the terminal, by the *tset* program, each time the user logs in. They will be printed in the following order: **is1**; **is2**; setting tabs using **tbc** and **hts**; **if**; running the program **iprog**; and finally **is3**. Most initialization is done with **is2**. Special terminal modes can be set up without duplicating strings by putting the common sequences in **is2** and special cases in **is1** and **is3**. A pair of sequences that does a harder reset from a totally unknown state can be analogously given as **rs1**, **rs2**, **rf**, and **rs3**, analogous to **is2** and **if**. These strings are output by the *reset* program, which is used when the terminal gets into a wedged state. Commands are normally placed in **rs2** and **rf** only if they produce annoying effects on the screen and are not necessary when logging in. For example, the command to set the vt100 into 80-column mode would normally be part of **is2**, but it causes an annoying glitch of the screen and is not normally needed since the terminal is usually already in 80 column mode.

If there are commands to set and clear tab stops, they can be given as **tbc** (clear all tab stops) and **hts** (set a tab stop in the current column of every row). If a more complex sequence is needed to set the tabs than can be described by this, the sequence can be placed in **is2** or **if**.

Delays

Certain capabilities control padding in the teletype driver. These are primarily needed by hard copy terminals, and are used by the *tset* program to set teletype modes appropriately. Delays embedded in the capabilities **cr**, **ind**, **cub1**, **ff**, and **tab** will cause the appropriate delay bits to be set in the teletype driver. If **pb** (padding baud rate) is given, these values can be ignored at baud rates below the value of **pb**.

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pad**. Only the first character of the **pad** string is used.

If the terminal has an extra "status line" that is not normally used by software, this fact can be indicated. If the status line is viewed as an extra line below the bottom line, into which one can cursor address normally (such as the Heathkit h19's 25th line, or the 24th line of a vt100 which is set to a 23-line scrolling region), the capability **hs** should be given. Special strings to go to the beginning of the status line and to return from the status line can be given as **tsl** and **fsl**. (**fsl** must leave the cursor position in the.same place it was before **tsl**. If necessary, the **sc** and **rc** strings can be included in **tsl** and **fsl** to get this effect.) The parameter **tsl** takes one parameter, which is the column number of the status line the cursor is to be moved to. If escape sequences and other special commands, such as tab, work while in the status line, the flag **eslok** can be given. A string which turns off the status line (or otherwise erases its contents) should be given as **dsl**. If the terminal has commands to save and restore the position of the cursor, give them as **sc** and **rc**. The status line is normally assumed to be the same width as the rest of the screen, e.g., **cols**. If the status line is a different width (possibly because the terminal does not allow an entire line to be loaded) the width, in columns, can be indicated with the numeric parameter **wsl**.

If the terminal can move up or down half a line, this can be indicated with **hu** (half-line up) and **hd** (half-line down). This is primarily useful for superscripts and subscripts on hardcopy terminals. If a hardcopy terminal can eject to the next page (form feed), give this as **ff** (usually control L).

If there is a command to repeat a given character a given number of times (to save time transmitting a large number of identical characters) this can be indicated with the parameterized string **rep**. The first parameter is the character to be repeated and the second is the number of times to repeat it. Thus, tparm(repeat_char, 'x', 10) is the same as 'xxxxxxxxxx'.

If the terminal has a settable command character, such as the TEKTRONIX 4025, this can be indicated with **cmdch**. A prototype command character is chosen which is used in all capabilities. This character is given in the **cmdch** capability to identify it. The following convention is supported on some HP-UX systems: The environment is to be searched for a **CC** variable, and if found, all occurrences of the prototype character are replaced with the character in the environment variable.

Terminal descriptions that do not represent a specific kind of known terminal, such as *switch*, *dialup*, *patch*, and *network*, should include the **gn** (generic) capability so that programs can complain that they do not know how to talk to the terminal. (This capability does not apply to *virtual* terminal descriptions for which the escape sequences are known.)

If the terminal uses xon/xoff handshaking for flow control, give **xon**. Padding information should still be included so that routines can make better decisions about costs, but actual pad characters will not be transmitted.

If the terminal has a "meta key" which acts as a shift key, setting the 8th bit of any character transmitted, this fact can be indicated with **km**. Otherwise, software will assume that the 8th bit is parity and it will usually be cleared. If strings exist to turn this "meta mode" on and off, they can be given as **smm** and **rmm**.

If the terminal has more lines of memory than will fit on the screen at once, the number of lines of memory can be indicated with **lm**. A value of **lm#0** indicates that the number of lines is not fixed, but that there is still more memory than fits on the screen.

If the terminal is one of those supported by the HP-UX virtual terminal protocol, the terminal number can be given as **vt**.

Media copy strings, which control an auxiliary printer connected to the terminal, can be given as **mc0**: print the contents of the screen, **mc4**: turn off the printer, and **mc5**: turn on the printer. When the printer is on, all text sent to the terminal will be sent to the printer. It is undefined whether the text is also displayed on the terminal screen when the printer is on. A variation **mc5p** takes one parameter, and leaves the printer on for as many characters as the value of the parameter, then turns the printer off. The parameter should not exceed 255. All text, including **mc4**, is transparently passed to the printer while an **mc5p** is in effect.

Strings to program function keys can be given as **pfkey**, **pfloc**, and **pfx**. Each of these strings takes two parameters: the function key number to program (from 0 to 10) and the string to program it with. Function key numbers out of this range may program undefined keys in a terminal dependent manner. The difference between the capabilities is that **pfkey** causes pressing the given key to be the same as the user typing the given string; **pfloc** causes the string to be executed by the terminal in local; and **pfx** causes the string to be transmitted to the computer.

Glitches

Hazeltine terminals, which do not allow '~' characters to be displayed should indicate **hz**.

Terminals which ignore a linefeed immediately after an **am** wrap, such as the Concept and vt100, should indicate **xenl**.

If **el** is required to get rid of standout (instead of merely writing normal text on top of it), **xhp** should be given.

Teleray terminals, where tabs turn all characters moved over to blanks, should indicate **xt** (destructive tabs). This glitch is also taken to mean that it is not possible to position the cursor on top of a "magic cookie", that to erase standout mode it is instead necessary to use delete and insert line.

The Beehive Superbee, which is unable to correctly transmit the escape or control C characters, has **xsb**, indicating that the f1 key is used for escape and f2 for control C. (Only certain Superbees have this problem, depending on the ROM.)

Other specific terminal problems may be corrected by adding more capabilities of the form *xx*.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **use** can be given with the name of the similar terminal. The capabilities given before **use** override those in the terminal type invoked by **use**. A capability can be cancelled by placing **xx@** to the left of the capability definition, where xx is the capability. For example, the entry

```
2621-nl, smkx@, rmkx@, use=2621,
```

defines a 2621-nl that does not have the **smkx** or **rmkx** capabilities, and hence does not turn on the function key labels when in visual mode. This is useful for different modes for a terminal, or for different user preferences.

WARNINGS

HP only supports terminals listed on the current list of supported devices. However, non-supported and supported terminals can be in the terminfo database. If you use such non-supported terminals, they may not work correctly.

FILES

/usr/lib/terminfo/?/*	files containing terminal descriptions

SEE ALSO

tic(1M), untic(1M), curses(3X), printf(3S), term(4).

NAME
ttytype – data base of terminal types by port

SYNOPSIS
/etc/ttytype

DESCRIPTION
Ttytype is a database containing, for each tty port on the system, the kind of terminal that is attached to that port. There is one line per port, containing the terminal kind (as a name listed in *terminfo*(4)), a space, and the name of the tty, less the initial "/dev/". For example, for an HP 2622 terminal on tty02:

```
2622 tty02
```

This information is read by *tset*(1) and by *login*(1) to initialize the TERM variable at login time.

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

SEE ALSO
login(1), tset(1).

BUGS
Some lines are merely known as "dialup" or "plugboard".

NAME

tztab – time zone adjustment table for date(1) and ctime(3C)

DESCRIPTION

The *tztab* file describes the differences between Greenwich Mean Time (GMT) and local time. Several local areas can be represented simultaneously with historical detail.

The file *tztab* consists of one or more time zone adjustment entries. The first line of the entry contains a unique string that may match the value of the TZ string in the user's environment. The format is **tzname***diff***dstzname** where **tzname** is the time zone name or abbreviation, *diff* is the difference in hours from GMT, and **dstzname** is the name or abbreviation of the "Daylight Savings" time zone. Fractional values of *diff* are expressed in minutes preceded by a colon. Each such string will start with an alphabetic character.

The second and subsequent lines of each entry will detail the time zone adjustments for that time zone. The lines contain seven fields each. The first six fields specify the first minute in which the time zone adjustment, specified in the seventh field, applies. The fields are separated by spaces or tabs. The first six are integer patterns that specify the minute (0-59), hour (0-23), day of the month (1-31), month of the year (1-12), year (1970-1999), and day of the week (0-6, with 0=Sunday). The minute, hour, and month of the year must contain a number in the (respective) range indicated above. The day of the month, year, and day of the week may contain a number as above or two numbers separated by a minus (indicating an inclusive range). Either the day of the month or the day of the week field must be a range, the other must be simple number.

The seventh field is a string that describes the time zone adjustment in its simplest form: **tzname***diff* where **tzname** is an alphabetic string giving the time zone name or abbreviation, and *diff* is the difference in hours from GMT. **Tzname** must match either the **tzname** field or the **dstzname** field in the first line of the time zone adjustment entry. Any fractional *diff* is shown in minutes.

Comments begin with # and include all characters up to a new-line. Comments are ignored.

If the value of the TZ string does not match any line in the table, it is interpreted according to the current American pattern.

EXAMPLES

The time zone adjustment table for the Eastern Time Zone in the United States is:

```
EST5EDT
0 3 6 1 1974 0-6 EDT4
0 3 22-28 2 1975 0 EDT4
0 3 24-30 4 1976-1986 0 EDT4
0 3 1-7 4 1987-1999 0 EDT4
0 1 24-30 11 1974 0 EST5
0 1 25-31 10 1975-1999 0 EST5
```

Normally (as indicated in the first line) Eastern Standard Time is five hours earlier than GMT. During Daylight Savings time, it changes to a 4 hour difference. The first time Daylight Savings Time took effect (second line) was on January 6, 1974 at 3:00 a.m. EDT. Note that the minute before was 1:59 a.m. EST. The change back to standard time took effect (sixth line) on the last Sunday in November of the same year. At that point, the time went from 1:59 a.m. EDT to 1:00 a.m. EST. The transition to Daylight Savings Time since then has gone from the last Sunday in February (third line) to the last Sunday in April (fourth line) to the first Sunday in April (fifth line). The return to standard time for the same period has remained at the last Sunday in October (seventh line).

AUTHOR

Tztab was developed by Hewlett-Packard Company.

FILES

/usr/lib/tztab

SEE ALSO

date(1), ctime(3C), environ(5).

INTERNATIONAL SUPPORT

8-bit data.

NAME

utmp, wtmp, btmp – utmp, wtmp, btmp entry format

SYNOPSIS

#include <sys/types.h>
#include <utmp.h>

DESCRIPTION

These files, which hold user and accounting information for such commands as *last*(1), *who*(1), *write*(1), and *login*(1), have the following structure as defined by **<utmp.h>**:

```
#define UTMP_FILE     "/etc/utmp"
#define WTMP_FILE     "/etc/wtmp"
#define BTMP_FILE     "/etc/btmp"
#define ut_name       ut_user

struct              utmp {
      char      ut_user[8];       /* User login name */
      char      ut_id[4];         /* /etc/inittab id (usually line #) */
      char      ut_line[12];      /* device name (console, lnxx) */
      short     ut_pid;           /* process id */
      short     ut_type;          /* type of entry */
      struct    exit_status {
                short             e_termination;                  /* Process termination status */
                short             e_exit;                         /* Process exit status */
                } ut_exit;        /* The exit status of a process
                                                                  /* marked as DEAD_PROCESS. */
      time_t    ut_time;          /* time entry was made */
};

/* Definitions for ut_type */
#define    EMPTY            0
#define    RUN_LVL          1
#define    BOOT_TIME        2
#define    OLD_TIME         3
#define    NEW_TIME         4
#define    INIT_PROCESS     5             /* Process spawned by "init" */
#define    LOGIN_PROCESS    6             /* A "getty" process waiting for login */
#define    USER_PROCESS     7             /* A user process */
#define    DEAD_PROCESS     8
#define    ACCOUNTING       9
#define    UTMAXTYPE        ACCOUNTING    /* Largest legal value of ut_type */

/* Special strings or formats used in the "ut_line" field when */
/* accounting for something other than a process */
/* No string for the ut_line field can be more than 11 chars + */
/* a NULL in length */
#define RUNLVL_MSG     "run-level %c"
#define BOOT_MSG       "system boot"
#define OTIME_MSG      "old time"
#define NTIME_MSG      "new time"
```

File **btmp** contains bad login entries for each invalid logon attempt.

Note that **wtmp** and **btmp** tend to grow without bound, and should be checked regularly. Information that is no longer useful should be removed periodically to prevent it from becoming too large.

FILES

/etc/utmp
/etc/wtmp
/etc/btmp

AUTHOR

Btmp was developed by the Hewlett-Packard Company, and the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

acctcon(1M), fwtmp(1M), last(1), login(1), who(1), write(1), getut(3C).

NAME

intro – introduction to miscellany

DESCRIPTION

This section describes miscellaneous facilities such as macro packages, character set tables, and the file system hierarchy.

SEE ALSO

The introduction to this manual.

NAME

ascii – map of ASCII character set

SYNOPSIS

cat /usr/pub/ascii

DESCRIPTION

Ascii is a map of the ASCII character set, giving both octal and hexadecimal equivalents of each character, to be printed as needed. It contains:

000 nul	001 soh	002 stx	003 etx	004 eot	005 enq	006 ack	007 bel
010 bs	011 ht	012 nl	013 vt	014 np	015 cr	016 so	017 si
020 dle	021 dc1	022 dc2	023 dc3	024 dc4	025 nak	026 syn	027 etb
030 can	031 em	032 sub	033 esc	034 fs	035 gs	036 rs	037 us
040 sp	041 !	042 ″	043 #	044 $	045 %	046 &	047 ´
050 (	051)	052 *	053 +	054 ,	055 -	056 .	057 /
060 0	061 1	062 2	063 3	064 4	065 5	066 6	067 7
070 8	071 9	072 :	073 ;	074 <	075 =	076 >	077 ?
100 @	101 A	102 B	103 C	104 D	105 E	106 F	107 G
110 H	111 I	112 J	113 K	114 L	115 M	116 N	117 O
120 P	121 Q	122 R	123 S	124 T	125 U	126 V	127 W
130 X	131 Y	132 Z	133 [	134 \	135]	136 ^	137 _
140 `	141 a	142 b	143 c	144 d	145 e	146 f	147 g
150 h	151 i	152 j	153 k	154 l	155 m	156 n	157 o
160 p	161 q	162 r	163 s	164 t	165 u	166 v	167 w
170 x	171 y	172 z	173 {	174 |	175 }	176 ~	177 del

00 nul	01 soh	02 stx	03 etx	04 eot	05 enq	06 ack	07 bel
08 bs	09 ht	0a nl	0b vt	0c np	0d cr	0e so	0f si
10 dle	11 dc1	12 dc2	13 dc3	14 dc4	15 nak	16 syn	17 etb
18 can	19 em	1a sub	1b esc	1c fs	1d gs	1e rs	1f us
20 sp	21 !	22 ″	23 #	24 $	25 %	26 &	27 ´
28 (	29)	2a *	2b +	2c ,	2d -	2e .	2f /
30 0	31 1	32 2	33 3	34 4	35 5	36 6	37 7
38 8	39 9	3a :	3b ;	3c <	3d =	3e >	3f ?
40 @	41 A	42 B	43 C	44 D	45 E	46 F	47 G
48 H	49 I	4a J	4b K	4c L	4d M	4e N	4f O
50 P	51 Q	52 R	53 S	54 T	55 U	56 V	57 W
58 X	59 Y	5a Z	5b [	5c \	5d]	5e ^	5f _
60 `	61 a	62 b	63 c	64 d	65 e	66 f	67 g
68 h	69 i	6a j	6b k	6c l	6d m	6e n	6f o
70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w
78 x	79 y	7a z	7b {	7c |	7d }	7e ~	7f del

FILES

/usr/pub/ascii

SEE ALSO

kana8(5), roman8(5).

NAME

environ – user environment

DESCRIPTION

An array of strings called the "environment" is made available by *exec*(2) when a process begins. By convention, these strings have the form "name=value". The following names are used by various commands:

PATH	The sequence of directory prefixes that *sh*(1), *time*(1), *nice*(1), *nohup*(1), etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by colons (:). *Login*(1) sets PATH=:/bin:/usr/bin .
HOME	Name of the user's login directory, set by *login*(1) From the password file, see *passwd*(4).
TERM	The kind of terminal for which output is to be prepared. This information is used by commands, such as *mm*(1) or *tplot*(1), that can exploit special capabilities of that terminal.
TZ	Time zone information. The minimum format is **tzname***diff* where **tzname** is an "alphabetic" string giving the time zone name or abbreviation, and *diff* is the difference in hours from GMT. *Diff* may be positive (west of Greenwich) or negative (east of Greenwich). Fractional hours are indicated as minutes preceded by a colon. If a summer time zone adjustment (such as Daylight Savings in the US) is to be applied the format is **tzname***diff***dstzname** where **dstzname** is the name of the "Daylight Savings" time zone. The entire string is compared with entries in the *tztab* file to determine the details of the time zone adjustment that should be applied, see *tztab*(4).
LANG	Language selection. This is one of the names listed in *langid*(5). It is used to select the character set, lexical order, up and down shift tables, and other information that varies from one area to another.

Further names may be placed in the environment by the *export* command and "name=value" arguments in *sh*(1), or by *exec*(2). It is unwise to conflict with certain shell variables that are frequently exported by **.profile** files: MAIL , PS1 , PS2 , IFS .

A process's environment is accessible from C via the global variable:

```
char **environ;
```

that points to an array of pointers to the strings which comprise the environment. The array is terminated by a null pointer.

AUTHOR

Environ was developed by AT&T and HP.

SEE ALSO

env(1), login(1), sh(1), exec(2), ctime(3C), getenv(3C), profile(4), term(5), tztab(4).

NAME

fcntl – file control options

SYNOPSIS

#include <fcntl.h>

DESCRIPTION

The *fcntl*(2) function provides for control over open files. This include file describes *requests* and *arguments* to *fcntl* and *open*(2).

```
/* Flag values accessible to open(2) and fcntl(2) */
/*  (The first three can only be set by open) */

#define O_RDONLY    0
#define O_WRONLY    1
#define O_RDWR      2
2568.if 1416<1416 .nr 50 1416
#define O_NDELAY    04         /* Non-blocking I/O */
#define O_APPEND    010        /* append (writes guaranteed at the end)
*/
#define O_SYNCIO    0100000    /* Do write through caching */

/* Flag values accessible only to open(2) */
#define O_CREAT     00400    /* Open with file create (uses third open arg)*/
#define O_TRUNC     01000    /* Open with truncation */
#define O_EXCL      02000    /* Exclusive open */

/* fcntl(2) requests */
#define F_DUPFD     0    /* Duplicate fildes */
#define F_GETFD     1    /* Get fildes flags */
#define F_SETFD     2    /* Set fildes flags */
#define F_GETFL     3    /* Get file flags */
#define F_SETFL     4    /* Set file flags */
#define F_GETLK     5    /* Get blocking file lock */
#define F_SETLK     6    /* Set or clear file locks and fail on busy */
#define F_SETLKW    7    /* Set or clear file locks and wait on busy
*/

/* file segment locking control structure */ struct flock {
    short    l_type;
    short    l_whence;
    long     l_start;
    long     l_len;
    int      l_pid;
};

/* file segment locking types */
#define F_RDLCK     01    /* Read lock */
#define F_WRLCK     02    /* Write lock */
#define F_UNLCK     03    /* Remove locks */
```

HARDWARE DEPENDENCIES

Series 200, 500

The F_GETLK, F_SETLK, and F_SETLKW commands are not supported.

SEE ALSO

fcntl(2), open(2).

NAME

hier – file system hierarchy

DESCRIPTION

The following outline gives a quick tour through a representative HP-UX directory hierarchy. Some of the directories listed only appear with HP-UX versions which support certain optional commands or packages which use those directories. Some HP-UX versions add special directories not shown here.

/	Root directory.
/bin	Frequently-used commands and those required to boot, restore, recover, and/or repair the system.
/dev	Special files (device files); see *mknod*(1).
/etc	
/etc/newconfig	New (updated) versions of customizable (localizable) configuration files and shell scripts. Shipped here so as not to overwrite current versions. Copied to regular locations for newly installed systems. Administrators may wish to keep them around for later reference.
/lib	Frequently-used object code libraries and related utilities.
/lost+found	For connecting detached files; for use by *fsck*(1).
/rbin	An analog to /bin for users in the restricted environment of *rsh*(1).
/tmp	Place to put temporary files (those normally with short lifetimes and which may be removed without notice).
/users	User home directories; sometimes immediate, sometimes at lower levels.
/users/guest	Default home directory for user "guest"; see *passwd*(4). Directory exists for novice users; you may wish to remove it.
/usr	Less-frequently-used commands and other miscellaneous things; historically, often a separate, mounted volume.
/usr/adm	
/usr/adm/sa	
/usr/bin	Less-frequently-used commands and those not required to boot, restore, recover, and/or repair the system.
/usr/bin/graph	*Gutil*(1) graphics commands.
/usr/contrib	User-contributed (unsupported, internal) commands, files, etc. Files under this directory come from outside the local site or organization, e.g. from users groups, HP service engineers, etc. See */usr/local* for local-site commands and files.
/usr/contrib/bin	User-contributed commands.
/usr/contrib/games	User-contributed games.
/usr/contrib/include	User-contributed include files. To include them, you must (in C) give a complete pathname, for example, #include "/usr/contrib/include/symtab.h".
/usr/contrib/lib	User-contributed libraries.
/usr/contrib/man/cat[1-8]	User-contributed manual entries, post-nroff form.

/usr/contrib/man/man[1-8]	User-contributed manual entries, pre-nroff form.
/usr/contrib/man/$LANG/cat[1-8]	User-contributed manual entries, formatted form for installed native languages. The LANG environment variable may take on values given in the **/usr/lib/nls/config** table.
/usr/contrib/man/$LANG/man[1-8]	User-contributed manual entries, unformatted form for installed native languages.
/usr/include	High-level C-language header files (shared definitions).
/usr/include/sys	Low-level (kernel-related) C-language header files.
/usr/lib	Less-frequently-used object code libraries, related utilities, miscellaneous data files, etc.
/usr/lib/acct	Certain system-administrative commands.
/usr/lib/cron	For *cron*(1M) and *at*(1) scheduling information.
/usr/lib/graphics/c	Device-independent Graphics Library (DGL) special C-language include files. Optional on some systems.
/usr/lib/graphics/demos	DGL demonstration software.
/usr/lib/graphics/fortran	DGL special FORTRAN-language include files.
/usr/lib/graphics/pascal	DGL special Pascal-language include files.
/usr/lib/help	Data files for *help*(1).
/usr/lib/lex	Data files for *lex*(1).
/usr/lib/macros	Macro definition packages for *nroff*(1) and *troff*.
/usr/lib/nls	native language support
/usr/lib/nls/config	correspondence between integer language id and name
/usr/lib/nls/$LANG	Language definition (Character Set Support, Local Customs, and Messages) for installed native languages. The LANG environment variable may take on values given in the **/usr/lib/nls/config** table.
/usr/lib/sa	
/usr/lib/spell	Data files for *spell*(1).
/usr/lib/tabset	Data files to set tabstops.
/usr/lib/term	Terminal initialization files.
/usr/lib/tmac	Macro definition packages for *nroff*(1) and *troff*.
/usr/lib/uucp[/*]	Commands, configuration files, and working directories for *uucp*(1).
/usr/local	Site-local commands, files, etc. Files under this directory come from inside the local site or organization. See */usr/contrib* for non-local unsupported commands and files.
/usr/local/bin	Site-local commands.

/usr/local/games	Site-local games.
/usr/local/include	Site-local include files. To include them, you must (in C) give a complete pathname, for example, #include "/usr/local/include/symtab.h".
/usr/local/lib	Site-local libraries.
/usr/local/man/cat[1-8]	Site-local manual entries, post-nroff form.
/usr/local/man/man[1-8]	Site-local manual entries, pre-nroff form.
/usr/local/man/$LANG/cat[1-8]	Site-local manual entries, unformatted form for installed native languages. The LANG environment variable may take on values given in the **/usr/lib/nls/config** table.
/usr/local/man/$LANG/man[1-8]	Site-local manual entries, formatted form for installed native languages.
/usr/mail	User mailboxes.
/usr/man	Online documentation.
/usr/man/cat[1-8]	Optional formatted (post-nroff) versions of online documentation for use by *man*(1).
/usr/man/man[1-8]	Unformatted (pre-nroff) versions of online documentation for use by *man*(1).
/usr/man/$LANG	Online documentation for installed native languages. The LANG environment variable may take on values given in the **/usr/lib/nls/config** table.
/usr/man/$LANG/cat[1-8]	Formatted native language versions of online documentation for use by *man*(1).
/usr/man/$LANG/man[1-8]	Unformatted native language versions of online documentation for use by *man*(1).
/usr/news	Local-system news articles for *news*(1).
/usr/preserve	Place where *ex*(1) and *vi*(1) save lost edit sessions until recovered.
/usr/rbin	An analog to /usr/bin for users in a restricted environment (as imposed by *rsh*(1)).
/usr/spool	Spooled (queued) files for various programs.
/usr/spool/cron	Spooled jobs for *cron*(1M) and *at*(1).
/usr/spool/cron/atjobs	Spooled jobs for *at*(1).
/usr/spool/lp	Control and working files for *lp*(1).
/usr/spool/lp/class	Printer class definition files.
/usr/spool/lp/interface	Printer interface shell scripts.
/usr/spool/lp/member	Printer class member definition files.

/usr/spool/lp/request	Spool directories for each logical destination.
/usr/spool/uucp	Queued work, lockfiles, logfiles, status files, and other files for *uucp*(1).
/usr/spool/uucppublic[/*]	Publicly-accessible directory for use with *uucp*(1).
/usr/src	Source files. Only present on HP-UX implementations which support source.
/usr/src/cmd/*	Source for commands. Simple command sources reside at the top level. Subdirectories are named after specific commands, e.g. */usr/src/cmd/cc*, and contain the source for multi-file or otherwise complicated commands. Directory structure below here depends on the individual command; see the associated makefiles.
/usr/src/games/*	Source for games. Simple game sources reside at the top level. Subdirectories are named after specific games, e.g. */usr/src/games/master*, and contain the source for multi-file or otherwise complicated games. Directory structure below here depends on the individual game; see the associated makefiles.
/usr/src/head	Include files which are copied into */usr/include/**.
/usr/src/lib	Source for libraries, in many subdirectories.
/usr/src/lib/libF77	Source for FORTRAN-77 miscellaneous (mostly math) libraries.
/usr/src/lib/libI77	Source for FORTRAN-77 I/O libraries.
/usr/src/lib/libPW	Source for Programmer's Workbench libraries.
/usr/src/lib/libc	Source for standard C libraries.
/usr/src/lib/libcurses/*	Source for curses (cursor control) libraries.
/usr/src/lib/libl	Source for lex(1) libraries.
/usr/src/lib/libm	Source for C math libraries.
/usr/src/lib/liby	Source for yacc(1) libraries.
/usr/tmp	Alternate place to put temporary files; usually used when there may be very many of them or if they will be large.

HARDWARE DEPENDENCIES

Series 500 systems support shared libraries loaded by the kernel at powerup time. They reside in the directory */etc/sslibs*.

Some directories include commands or files not supported on all HP-UX implementations.

SEE ALSO

find(1), grep(1), ls(1), whereis(1).

NAME

hpnls – HP Native Language Support (NLS) Model

SYNOPSIS

ls /usr/lib/nls/*

DESCRIPTION

The HP Native Language Support (NLS) model includes several capabilities that reduce or eliminate the barriers that would otherwise make HP–UX difficult to use in a non-English language. The three main categories, Character Set Support, Local Customs, and Messages, are subdivided into smaller categories in order to adequately reflect the extent of the Native Language Support.

CHARACTER SET SUPPORT –

A major NLS objective is to provide capabilities for adapting character sequences to local language needs.

CHARACTER CODE SIZE –

The length of the character code governs the number of distinct characters that can be included in the character set.

7–BIT –

The ASCII character set consists of 33 control characters including DEL, space, and 94 printable characters. (See *ascii*(5).) This is sufficient to span the Latin alphabet, upper and lowercase, plus punctuation and special symbols. Seven bits of information is sufficient to distinguish the characters in such a set.

8–BIT –

The use of an 8 bit character code allows 67 control codes, space, and 188 printable characters. In the case of European characters, this provides sufficient space for accented vowels, consonants with special forms, and other special symbols. (See *roman8*(5)). This is also sufficient to hold the phonetic Japanese character set Katakana. (See *kana8*(5).)

16–BIT –

A number of languages have very large character sets that require more than the 188 printable characters provided by the 8-bit character codes. Sixteen–bit character codes are available for these languages. To simplify processing, 16-bit printable characters are formed from pairs of 8-bit printable characters (neither byte may contain a control code or a space). This allows representation of up to 35344 characters.

CHARACTER TYPING –

Character processing which depends on character type must take into account the character type changes that vary with the character set being used. For example, an alphabetic character in the ROMAN8 character set may align with a punctuation character in the kana8 set.

SHIFTING –

While the ROMAN8 character set has uppercase and lowercase for most alphabetic characters, some languages discard accents when characters are shifted to uppercase. Other alphabetic characters may not be shifted at all, when there is no notion of "case" in the underlying language.

COLLATING –

The ASCII collation order, while generally tolerated, is not adequate for American dictionary usage. Different languages sort characters from the ROMAN8 set in different orders. Some languages require that character pairs, such as "ch" and "ll" in Spanish, be sorted as single characters. Ideographic character sets may have multiple orderings. For example, Japanese characters may be sorted in phonetic order; in a different order based on the number of strokes in the ideogram; or according, first, to the radical (root) of the character and, second, to the number of strokes added to the radical.

DIRECTIONALITY –

The assumption that displayed text goes from left to right does not hold for all languages. Some Middle Eastern languages go from right to left. Far Eastern languages usually use vertical columns, starting from the right.

CODING SCHEME CONSIDERATIONS –

Although most HP supported 8-bit character sets preserve the ASCII codes in the range of 0 to 127, 16-bit character sets may use these byte values in 2–byte characters. Software that assigns special meaning to bytes (metacharacters) in this range must distinguish between 1–byte and 2–byte characters. In multilingual environments, standard escape code sequences are used to indicate change to alternate character sets. Since these sequences are not usually printed or displayed, the number of characters output is usually less than the number of bytes in the sequence. Any software that must locate a character within a sequence must accommodate this.

LOCAL CUSTOMS –

Some aspects of Native Language Support relate more to local customs of a particular geographic location than to the characters used to write the language.

REPRESENTATION OF NUMBERS –

The character used to denote the radix of a decimal number varies for different regions. Similarly the use of a "thousands" indicator or grouping of (usually three) digits may vary with local custom.

CURRENCY REPRESENTATION –

The symbol for currency varies from country to country. The symbol may either precede or follow the numeric value. Some currencies allow decimal fractions while others use alternate methods of representing smaller monetary values.

DATE AND TIME REPRESENTATION –

Month and weekday names vary with language (if they are not omitted entirely). Abbreviations may be other than three characters, or may not be allowed at all. Even when a strictly numeric representation is used, the order of year, month, and day as well as the delimiters which separate them is not universal.

DATE AND TIME ADJUSTMENTS –

The HP–UX system clock runs on Greenwich Mean Time (GMT). Corrections to local time zones consist of adding or subtracting whole or fractional hours from GMT. The Gregorian calendar is most common, but some locales use different methods for determining meridian day and year; usually based on seasonal, astronomical, or historical events.

MESSAGES –

The need for messages to be readable by users is perhaps the most significant justification for implementing Native Language Support.

MESSAGE CONTENT –

Error messages, prompts, expected responses, and mnemonic command names should be based on the user's native language.

MESSAGE STRUCTURE –

Messages must often be built from segments. To accommodate grammatical differences, it may be necessary to change the order in which the fragments are connected.

EXAMPLE

A "fully localized" version of "pr" would

Never strip the 8th bit of a character code.

Properly format the date in each page header.

Account for non-printing escape sequences.

Use the message catalog system to select user error messages.

FILES

usr/lib/nls/*

AUTHOR

Hpnls was developed by the Hewlett-Packard Company.

SEE ALSO

date(1), sort(1), ctime(3C), ecvt(3C), nl_conv(3C), nl_ctype(3C), nl_string(3C), strtod(3C), ascii(5), kana8(5), roman8(5).

NAME

ioctl – generic device control commands

SYNOPSIS

#include <sys/ioctl.h>
ioctl(fildes, request, arg)
int fildes, request;

DESCRIPTION

The *ioctl*(2) system call provides for control over open devices. This include file describes *requests* and *arguments* used in *ioctl*(2) which are of a generic nature. For details about how individual requests will affect any particular device, see the corresponding device manual page section **(7)**. If a device does not support an ioctl request it will return **EINVAL**.

FIONREAD
: Returns in the long integer whose address is arg the number of characters immediately readable from the device file.

FIOSSAIOSTAT
: For those character device files which support this command, if the integer whose address is *arg* is non-zero, system asynchronous I/O is enabled. That is, enable SIGIO to be sent to the process currently designated with FIOSSAIOOWN (see below) whenever device-file dependent events occur. If no process has been designated with FIOSSAIOOWN, then enable SIGIO to be sent to the first process to open the device file.

If the designated process has exited, the SIGIO signal will not be sent to any process.

If the integer whose address is *arg* is 0, system asynchronous I/O is disabled.

FIOGSAIOSTAT
: For those character device files which support this command, the integer whose address is *arg* is set to 1, if system asynchronous I/O is enabled. Otherwise, the integer whose address is *arg* is set to 0.

FIOSSAIOOWN
: For those character device files which support this command, set process ID to receive the SIGIO signals with system asynchronous I/O to the value of the integer whose address is *arg*. The super-user may designate that any process receive the SIGIO signals. If the request is not made by the super-user, the calling process is only allowed to designate that itself or another process whose real or saved effective user ID matches its real or effective user ID, or a process which is a descendant of the calling process, receive the SIGIO signals. If no process can be found corresponding to that specified by the integer whose address is arg, the call will fail, with errno set to ESRCH. If the request is not made by the super-user, and the calling process attempts to designate a process other than itself or another process whose real or saved effective user ID matches its real or effective user ID, or a process which is not a descendant of the calling process, the call will fail, with errno set to EPERM.

If the designated process subsequently exits, the SIGIO signal will not be sent to any process.

The default on open of a device file is that the process performing the open is set to receive the SIGIO signals.

FIOGSAIOOWN

For those character device files which support this command, the integer whose address is *arg* is set to the process ID designated to receive SIGIO signals.

FIOSNBIO

For those character device files which support this command, if the integer whose address is *arg* is non-zero, non-blocking I/O is enabled. That is, subsequent reads and writes to the device file will be handled in a non-blocking manner (see below). If the integer whose address is *arg* is 0, non-blocking I/O is disabled.

For reads, non-blocking I/O will prevent all read requests to that device from blocking, whether the requests succeed or fail. Such a read request will complete in one of three ways: (1) If there is enough data available to satisfy the entire request, the read will complete successfully, having read all of the data, and return the number of bytes read; (2) If there is not enough data available to satisfy the entire request, the read will complete successfully, having read as much data as possible, and return the number of bytes it was able to read; (3) If there is no data available, the read will fail and *errno* will be set to EWOULDBLOCK.

For writes, non-blocking I/O will prevent all write requests to that device file from blocking, whether the requests succeed or fail. Such a write request will complete in one of three ways: (1) If there is enough space available in the system to buffer all the data, the write will complete successfully, having written out all of the data, and return the number of bytes written; (2) If there is not enough space in the buffer to write out the entire request, the write will complete successfully, having written as much data as possible, and return the number of bytes it was able to write; (3) If there is no space in the buffer, the write will fail and *errno* will be set to EWOULDBLOCK.

To prohibit non-blocking I/O from interfering with the O_NDELAY flag (see *open*(2) and *fcntl*(2)), the functionality of O_NDELAY always supercedes the functionality of non-blocking I/O. This means that if O_NDELAY is set, the driver will perform read requests in accordance with the definition of O_NDELAY. When O_NDELAY is not set, the definition of non-blocking I/O applies.

The default on open of a device file is that non-blocking I/O is disabled.

FIOGNBIO

For those character device files which support this command, the integer whose address is *arg* is set to 1, if non-blocking I/O is enabled. Otherwise, the integer whose address is *arg* is set to 0.

WARNINGS

FIOSSAIOSTAT is similar to 4.2 BSD FIOASYNC, with the addition of provisions for security. FIOGSAIOSTAT is of HP origin, complements FIOSSAIOSTAT, and allows saving and restoring system asynchronous I/O TTY state for BSD style job control. FIOSSAIOOWN is similar to 4.2 BSD FIOSETOWN, with the addition of provisions for security. FIOGSAIOOWN is similar to 4.2 BSD FIOGETOWN. Note also the difference that the 4.2 BSD version of this functionality used process groups, while the HP-UX version only uses processes. FIOSNBIO is the same as 4.2 BSD FIONBIO, except that it does not interfere with the AT&T O_NDELAY *open* and *fcntl* flag. FIOGNBIO is of HP origin, complements FIOSNBIO, and allows saving and restoring non-blocking

I/O TTY state for BSD style job control.

HARDWARE DEPENDENCIES

Series 200, 300, 500

Asynchronous I/O is not supported.

SEE ALSO

ioctl(2).

Section (7) of this manual.

NAME

kana8 – map of KANA8 character set used by NLS

SYNOPSIS

ls /usr/lib/nls/*

DESCRIPTION

Kana8 is a map of the KANA8 character set, giving the octal, decimal, and hexadecimal equivalents of each character, to be printed as needed. It contains:

000	0	00	nul	001	1	01	soh
002	2	02	stx	003	3	03	etx
004	4	04	eot	005	5	05	enq
006	6	06	ack	007	7	07	bel
010	8	08	bs	011	9	09	ht
012	10	0a	nl	013	11	0b	vt
014	12	0c	np	015	13	0d	cr
016	14	0e	so	017	15	0f	si
020	16	10	dle	021	17	11	dc1
022	18	12	dc2	023	19	13	dc3
024	20	14	dc4	025	21	15	nak
026	22	16	syn	027	23	17	etb
030	24	18	can	031	25	19	em
032	26	1a	sub	033	27	1b	esc
034	28	1c	fs	035	29	1d	gs
036	30	1e	rs	037	31	1f	us
040	32	20	sp	041	33	21	!
042	34	22	"	043	35	23	#
044	36	24	$	045	37	25	%
046	38	26	&	047	39	27	'
050	40	28	(	051	41	29	)
052	42	2a	*	053	43	2b	+
054	44	2c	,	055	45	2d	-
056	46	2e	.	057	47	2f	/
060	48	30	0	061	49	31	1
062	50	32	2	063	51	33	3
064	52	34	4	065	53	35	5
066	54	36	6	067	55	37	7
070	56	38	8	071	57	39	9
072	58	3a	:	073	59	3b	;
074	60	3c	<	075	61	3d	=
076	62	3e	>	077	63	3f	?
100	64	40	@	101	65	41	A
102	66	42	B	103	67	43	C
104	68	44	D	105	69	45	E
106	70	46	F	107	71	47	G
110	72	48	H	111	73	49	I
112	74	4a	J	113	75	4b	K
114	76	4c	L	115	77	4d	M
116	78	4e	N	117	79	4f	O

120	80	50	P		121	81	51	Q	
122	82	52	R		123	83	53	S	
124	84	54	T		125	85	55	U	
126	86	56	V		127	87	57	W	
130	88	58	X		131	89	59	Y	
132	90	5a	Z		133	91	5b	[	
134	92	5c	¥	yen	135	93	5d	]	
136	94	5e	^		137	95	5f	_	
140	96	60	`		141	97	61	a	
142	98	62	b		143	99	63	c	
144	100	64	d		145	101	65	e	
146	102	66	f		147	103	67	g	
150	104	68	h		151	105	69	i	
152	106	6a	j		153	107	6b	k	
154	108	6c	l		155	109	6d	m	
156	110	6e	n		157	111	6f	o	
160	112	70	p		161	113	71	q	
162	114	72	r		163	115	73	s	
164	116	74	t		165	117	75	u	
166	118	76	v		167	119	77	w	
170	120	78	x		171	121	79	y	
172	122	7a	z		173	123	7b	{	
174	124	7c	¦		175	125	7d	}	
176	126	7e	~		177	127	7f	del	
200	128	80			201	129	81		
202	130	82			203	131	83		
204	132	84			205	133	85		
206	134	86			207	135	87		
210	136	88			211	137	89		
212	138	8a			213	139	8b		
214	140	8c			215	141	8d		
216	142	8e	ss2		217	143	8f	ss3	
220	144	90			221	145	91		
222	146	92			223	147	93		
224	148	94			225	149	95		
226	150	96			227	151	97		
230	152	98			231	153	99		
232	154	9a			233	155	9b		
234	156	9c			235	157	9d		
236	158	9e			237	159	9f		
240	160	a0			241	161	a1	｡	ku-ten
242	162	a2	｢	hook	243	163	a3	｣	unhook
244	164	a4	､	to-ten	245	165	a5	･	dot
246	166	a6	ｦ	wo	247	167	a7	ｧ	small a
250	168	a8	ｨ	small i	251	169	a9	ｩ	small u
252	170	aa	ｪ	small e	253	171	ab	ｫ	small o
254	172	ac	ｬ	small ya	255	173	ad	ｭ	small yu
256	174	ae	ｮ	small yo	257	175	af	ｯ	small tsu
260	176	b0	ｰ	dash	261	177	b1	ｱ	a
262	178	b2	ｲ	i	263	179	b3	ｳ	u
264	180	b4	ｴ	e	265	181	b5	ｵ	o

266	182	b6	ｶ	ka	267	183	b7	ｷ	ki
270	184	b8	ｸ	ku	271	185	b9	ｹ	ke
272	186	ba	ｺ	ko	273	187	bb	ｻ	sa
274	188	bc	ｼ	shi	275	189	bd	ｽ	su
276	190	be	ｾ	se	277	191	bf	ｿ	so
300	192	c0	ﾀ	ta	301	193	c1	ﾁ	chi
302	194	c2	ﾂ	tsu	303	195	c3	ﾃ	te
304	196	c4	ﾄ	to	305	197	c5	ﾅ	na
306	198	c6	ﾆ	ni	307	199	c7	ﾇ	nu
310	200	c8	ﾈ	ne	311	201	c9	ﾉ	no
312	202	ca	ﾊ	ha	313	203	cb	ﾋ	hi
314	204	cc	ﾌ	fu	315	205	cd	ﾍ	he
316	206	ce	ﾎ	ho	317	207	cf	ﾏ	ma
320	208	d0	ﾐ	mi	321	209	d1	ﾑ	mu
322	210	d2	ﾒ	me	323	211	d3	ﾓ	mo
324	212	d4	ﾔ	ya	325	213	d5	ﾕ	yu
326	214	d6	ﾖ	yo	327	215	d7	ﾗ	ra
330	216	d8	ﾘ	ri	331	217	d9	ﾙ	ru
332	218	da	ﾚ	re	333	219	db	ﾛ	ro
334	220	dc	ﾜ	wa	335	221	dd	ﾝ	n
336	222	de	ﾞ	voiced	337	223	df	ﾟ	non-voiced plosive
340	224	e0			341	225	e1		
342	226	e2			343	227	e3		
344	228	e4			345	229	e5		
346	230	e6			347	231	e7		
350	232	e8			351	233	e9		
352	234	ea			353	235	eb		
354	236	ec			355	237	ed		
356	238	ee			357	239	ef		
360	240	f0			361	241	f1		
362	242	f2			363	243	f3		
364	244	f4			365	245	f5		
366	246	f6			367	247	f7		
370	248	f8			371	249	f9		
372	250	fa			373	251	fb		
374	252	fc			375	253	fd		
376	254	fe			377	255	ff		

WARNINGS

Peripheral or software limitations may garble this manual page. Many printers and terminals do not support the KANA8 character set.

AUTHOR

Kana8 was developed by HP.

FILES

/usr/lib/nls/*

SEE ALSO

ascii(5), hpnls(5), roman8(5).

NAME

langid – language identification variable

DESCRIPTION

This page defines integer values corresponding to values of the variable **LANG** in the user's environment. These are the values returned by *currlangid*(3C), and are passed as parameters into native language support library routines.

LANGUAGE NAMES

The following languages are currently supported by HP-UX.

Language Num	Abbreviation	Name
00	n-computer	native computer
01	american	american
02	c-french	canadian french
03	danish	danish
04	dutch	dutch
05	english	english
06	finnish	finnish
07	french	french
08	german	german
09	italian	italian
10	norwegian	norwegian
11	portuguese	portuguese
12	spanish	spanish
13	swedish	swedish
41	katakana	katakana
51	arabic	arabic
52	arabic-w	western arabic
61	greek	greek
71	hebrew	hebrew
81	turkish	turkish
221	japanese	japanese
901-999		available for user defined languages

WARNINGS

Right-to-left character processing is not yet supported by all HP-UX commands for the arabic, western arabic, and hebrew languages.

AUTHOR

Langid was developed by the Hewlett-Packard Company.

SEE ALSO

langinfo(3C), environ(5), hier(5), hpnls(5).

NAME

man – macros for formatting entries in this manual

DESCRIPTION

These *nroff* and *troff* macros are used to lay out the format of the entries of this manual. These macros are used by the *man*(1) command.

The default page size is 8.5×11, with a 6.5×10 text area; the **–rs1** option reduces these dimensions to 6×9 and 4.75×8.375, respectively. This option (which is *not* effective in *nroff*) also reduces the default type size from 10-point to 9-point, and the vertical line spacing from 12-point to 10-point. The **–rV2** option may be used to set certain parameters to values appropriate for certain Versatec printers: it sets the line length to 82 characters, the page length to 84 lines, and it inhibits underlining. This option should not be confused with the **–Tvp** option of the *man*(1) command, which is available on some UNIX operating systems.

Any *text* argument below may be one to six "words". Double quotes ("") may be used to include blanks in a "word". If *text* is empty, the special treatment is applied to the next line that contains text to be printed. For example, **.I** may be used to italicize a whole line, or **.SM** followed by **.B** to make small bold text. By default, hyphenation is turned off for *nroff*, but remains on for *troff*.

Type font and size are reset to default values before each paragraph and after processing font- and size-setting macros, e.g., **.I**, **.RB**, **.SM**. Tab stops are neither used nor set by any macro except **.DT** and **.TH**. **.TH** invokes **.DT** (see below).

Default units for indents *in* are ens. When *in* is omitted, the previous indent is used. This remembered indent is set to its default value (7.2 ens in *troff*, 5 ens in *nroff*–this corresponds to 0.5 inch in the default page size) by **.TH**, **.P**, and **.RS**, and restored by **.RE**.

.TH *t s c n o i*	Set the title and entry heading; *t* is the title, *s* is the section number, *c* is extra commentary, e.g., "local", *n* is new manual name, *o* and *i* are used in printing HP-UX manual pages. If *o* is "HP-UX", it is a standard page and HP-UX will be the center title, overriding the *c* and *n* options. If *o* is empty, the page is non-standard and there is no center title printed. The *i* option is used to set the second line of the header to reflect which HP-UX systems support that page. If *i* is empty, the page is supported by all the systems. If one or more systems do not support it, a quoted string is used to list the systems that do support the page. For example: **"Series 200, 300, 800 Only"**
.SH *text*	Place subhead *text*, e.g., **SYNOPSIS**, here.
.SS *text*	Place sub-subhead *text*, e.g., **Options**, here.
.B *text*	Make *text* bold.
.I *text*	Make *text* italic.
.SM *text*	Make *text* 1 point smaller than default point size.
.RI *a b*	Concatenate roman *a* with italic *b*, and alternate these two fonts for up to six arguments. Similar macros alternate between any two of roman, italic, and bold: **.IR .RB .BR .IB .BI**
.P	Begin a paragraph with normal font, point size, and indent. **.PP** is a synonym for **.P**.
.HP *in*	Begin paragraph with hanging indent.
.TP *in*	Begin indented paragraph with hanging tag. The next line that contains text to be printed is taken as the tag. If the tag does not fit, it is printed on a separate line.

.IP *t in*	Same as **.TP** *in* with tag *t*; often used to get an indented paragraph without a tag.
.RS *in*	Increase relative indent (initially zero). Indent all output an extra *in* units from the current left margin.
.RE *k*	Return to the *k*th relative indent level (initially, *k*=1; *k*=0 is equivalent to *k*=1); if *k* is omitted, return to the most recent lower indent level.
.PM *m*	Produces proprietary markings; where *m* may be **P** for **PRIVATE**, or **N** for **NOTICE**, **BP** for **BELL LABORATORIES PROPRIETARY**, or **BR** for **BELL LABORATORIES RESTRICTED**.
.DT	Restore default tab settings (every 7.2 ens in *troff*, 5 ens in *nroff*).
.PD *v*	Set the interparagraph distance to *v* vertical spaces. If *v* is omitted, set the interparagraph distance to the default value (0.4v in *troff*, 1v in *nroff*).

The following *strings* are defined:

\*R	**(Reg.)** in *nroff*, Registered Trademark symbol in *troff*, if available.
\*S	Change to default type size.
\*(Tm	Trademark indicator.

The following *number registers* are given default values by **.TH**:

IN	Left margin indent relative to subheads (default is 7.2 ens in *troff*, 5 ens in *nroff*).
LL	Line length including **IN**.
PD	Current interparagraph distance.

CAVEATS

In addition to the macros, strings, and number registers mentioned above, there are defined a number of *internal* macros, strings, and number registers. Except for names predefined by *troff* and number registers **d**, **m**, and **y**, all such internal names are of the form *XA*, where *X* is one of),], and }, and *A* stands for any alphanumeric character.

The programs that prepare the Table of Contents and the Permuted Index for this Manual assume the *NAME* section of each entry consists of a single line of input that has the following format:

```
name[, name, name ...] \- explanatory text
```

The macro package increases the inter-word spaces (to eliminate ambiguity) in the *SYNOPSIS* section of each entry.

The macro package itself uses only the roman font (so that one can replace, for example, the bold font by the constant-width font–see *cw*(1)). Of course, if the input text of an entry contains requests for other fonts (e.g., **.I**, **.RB**, **\fI**), the corresponding fonts must be mounted.

FILES

/usr/lib/macros/cmp.[nt].[dt].an
/usr/lib/tmac/tmac.an
/usr/lib/macros/ucmp.[nt].an

SEE ALSO

man(1), nroff(1).

WARNINGS

If the argument to **.TH** contains *any* blanks and is *not* enclosed by double quotes (""), the output can be incorrectly formatted.

NAME

math – math functions and constants

SYNOPSIS

#include <math.h>

DESCRIPTION

This file contains declarations of all the functions in the Math Library (described in Section (3M)), as well as various functions in the C Library (Section (3C)) that return floating-point values.

It defines the structure and constants used by the *matherr*(3M) error-handling mechanisms, including the following constant used as an error-return value:

HUGE	The maximum value of a single-precision floating-point number.

The following mathematical constants are defined for user convenience:

M_E	The base of natural logarithms (e).
M_LOG2E	The base-2 logarithm of e.
M_LOG10E	The base-10 logarithm of e.
M_LN2	The natural logarithm of 2.
M_LN10	The natural logarithm of 10.
M_PI	The ratio of the circumference of a circle to its diameter. (There are also several fractions of its reciprocal and its square root.)
M_SQRT2	The positive square root of 2.
M_SQRT1_2	The positive square root of 1/2.

For the definitions of various machine-dependent "constants," see the description of the <*values.h*> header file.

FILES

/usr/include/math.h

SEE ALSO

intro(3), matherr(3M), values(5).

NAME

mm – the MM macro package for formatting documents

SYNOPSIS

mm [options] [files]

nroff –mm [options] [files]

nroff –cm [options] [files]

DESCRIPTION

This package provides a formatting capability for a very wide variety of documents. The manner in which a document is typed in and edited is essentially independent of whether the document is to be eventually formatted at a terminal or is to be phototypeset. See the references below for further details.

The –**mm** option causes *nroff*(1) and *troff* to use the non-compacted version of the macro package, while the –**cm** option results in the use of the compacted version, thus speeding up the process of loading the macro package.

HARDWARE DEPENDENCIES

Series 500

Compacted macros are not supported.

FILES

/usr/lib/macros/cmp.n.[dt].m	compacted version of the package
/usr/lib/macros/mmn	non-compacted version of the package
/usr/lib/tmac/tmac.m	pointer to the non-compacted version of the package
/usr/lib/macros/ucmp.n.m	initializers for the compacted version of the package

SEE ALSO

mm(1), nroff(1).

NAME

prof – profile within a function

SYNOPSIS

#define MARK
#include <prof.h>

void MARK (name)

DESCRIPTION

MARK will introduce a mark called *name* that will be treated the same as a function entry point. Execution of the mark will add to a counter for that mark, and program-counter time spent will be accounted to the immediately preceding mark or to the function if there are no preceding marks within the active function.

Name may be any combination of up to six letters, numbers or underscores. Each *name* in a single compilation must be unique, but may be the same as any ordinary program symbol.

For marks to be effective, the symbol MARK must be defined before the header file <*prof.h*> is included. This may be defined by a preprocessor directive as in the synopsis, or by a command line argument, i.e:

```
cc -p -DMARK foo.c
```

If MARK is not defined, the *MARK*(name) statements may be left in the source files containing them and will be ignored.

EXAMPLE

In this example, marks can be used to determine how much time is spent in each loop. Unless this example is compiled with *MARK* defined on the command line, the marks are ignored.

```
#include <prof.h>

foo( )
{
        int i, j;

        .
        .
        .
        MARK(loop1);
        for (i = 0; i < 2000; i++) {
                . . .
        }
        MARK(loop2);
        for (j = 0; j < 2000; j++) {
                . . .
        }
}
```

SEE ALSO

prof(1), profil(2), monitor(3C).

NAME

INIT, GETC, PEEKC, UNGETC, RETURN, ERROR, compile, step, advance – regular expression compile and match routines

SYNOPSIS

```
#define INIT <declarations>
#define GETC( ) <getc code>
#define PEEKC( ) <peekc code>
#define UNGETC(c) <ungetc code>
#define RETURN(pointer) <return code>
#define ERROR(val) <error code>

#include <regexp.h>

char *compile (instring, expbuf, endbuf, eof)
char *instring, *expbuf, *endbuf;
int eof;

int step (string, expbuf)
char *string, *expbuf;

int advance (string, expbuf)
char *string, *expbuf;

extern char *loc1, *loc2, *locs;

extern int circf, sed, nbra;
```

DESCRIPTION

This page describes general-purpose regular expression matching routines in the form of *ed*(1), defined in **/usr/include/regexp.h**. Programs such as *ed*(1), *sed*(1), *grep*(1), *bs*(1), *expr*(1), etc., which perform regular expression matching use this source file. In this way, only this file need be changed to maintain regular expression compatibility.

The interface to this file is unpleasantly complex. Programs that include this file must have the following five macros declared before the "#include <regexp.h>" statement. These macros are used by the *compile* routine.

GETC( ) Return the value of the next character in the regular expression pattern. Successive calls to GETC( ) should return successive characters of the regular expression.

PEEKC( ) Return the next character in the regular expression. Successive calls to PEEKC( ) should return the same character (which should also be the next character returned by GETC( )).

UNGETC(*c*) Cause the argument *c* to be returned by the next call to GETC( ) (and PEEKC( )). No more than one character of pushback is ever needed and this character is guaranteed to be the last character read by GETC( ). The value of the macro UNGETC(*c*) is always ignored.

RETURN(*pointer*)
This macro is used on normal exit of the *compile* routine. The value of the argument *pointer* is a pointer to the character after the last character of the compiled regular expression. This is useful to programs which have memory allocation to manage.

ERROR(*val*) This is the abnormal return from the *compile* routine. The argument *val* is an error number (see table below for meanings). This call should never return.

ERROR	MEANING
11	Range endpoint too large.
16	Bad number.
25	"\digit" out of range.
36	Illegal or missing delimiter.
41	No remembered search string.
42	\(\) imbalance.
43	Too many \(.
44	More than 2 numbers given in \{ \}.
45	} expected after \.
46	First number exceeds second in \{ \}.
49	[ ] imbalance.
50	Regular expression overflow.

The syntax of the *compile* routine is as follows:

```
compile(instring, expbuf, endbuf, eof)
```

The first parameter *instring* is never used explicitly by the *compile* routine but is useful for programs that pass down different pointers to input characters. It is sometimes used in the INIT declaration (see below). Programs which call functions to input characters or have characters in an external array can pass down a value of ((char *) 0) for this parameter.

The next parameter *expbuf* is a character pointer. It points to the place where the compiled regular expression will be placed.

The parameter *endbuf* is one more than the highest address where the compiled regular expression may be placed. If the compiled expression cannot fit in (*endbuf−expbuf*) bytes, a call to ERROR(50) is made.

The parameter *eof* is the character which marks the end of the regular expression. For example, in *ed*(1), this character is usually a /.

Each program that includes this file must have a **#define** statement for INIT. This definition will be placed right after the declaration for the function *compile* and the opening curly brace ({). It is used for dependent declarations and initializations. Most often it is used to set a register variable to point to the beginning of the regular expression so that this register variable can be used in the declarations for GETC( ), PEEKC( ) and UNGETC( ). Otherwise it can be used to declare external variables that might be used by GETC( ), PEEKC( ) and UNGETC( ). See the example below of the declarations taken from *grep*(1).

There are other functions in this file which perform actual regular expression matching, one of which is the function *step*. The call to *step* is as follows:

```
step(string, expbuf)
```

The first parameter to *step* is a pointer to a string of characters to be checked for a match. This string should be null terminated.

The second parameter *expbuf* is the compiled regular expression which was obtained by a call of the function *compile*.

The function *step* returns non-zero if the given string matches the regular expression, and zero if the expressions do not match. If there is a match, two external character pointers are set as a side effect to the call to *step*. The variable set in *step* is *loc1*. This is a pointer to the first character that matched the regular expression. The variable *loc2*, which is set by the function *advance*, points to the character after the last character that matches the regular expression. Thus if the regular expression matches the entire line, *loc1* will point to the first character of *string* and *loc2* will point to the null at the end of *string*.

Step uses the external variable *circf* which is set by *compile* if the regular expression begins with ^. If this is set then *step* will try to match the regular expression to the beginning of the string only. If more than one regular expression is to be compiled before the first is executed the value of *circf* should be saved for each compiled expression and *circf* should be set to that saved value before each call to *step*.

The function *advance* is called from *step* with the same arguments as *step*. The purpose of *step* is to step through the *string* argument and call *advance* until *advance* returns non-zero indicating a match or until the end of *string* is reached. If one wants to constrain *string* to the beginning of the line in all cases, *step* need not be called; simply call *advance*.

When *advance* encounters a * or \{ \} sequence in the regular expression, it will advance its pointer to the string to be matched as far as possible and will recursively call itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, *advance* will back up along the string until it finds a match or reaches the point in the string that initially matched the * or \{ \}. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer *locs* is equal to the point in the string at sometime during the backing up process, *advance* will break out of the loop that backs up and will return zero. This is used by *ed*(1) and *sed*(1) for substitutions done globally (not just the first occurrence, but the whole line) so, for example, expressions like **s/y*//g** do not loop forever.

The additional external variables *sed* and *nbra* are used for special purposes.

EXAMPLES

The following is an example of how the regular expression macros and calls look from *grep*(1):

```
#define INIT            register char *sp = instring;
#define GETC()          (*sp++)
#define PEEKC()         (*sp)
#define UNGETC(c)       (—sp)
#define RETURN(c)       return;
#define ERROR(c)        regerr()

#include <regexp.h>
...
        (void) compile(*argv, expbuf, &expbuf[ESIZE], '\0');
...
        if (step(linebuf, expbuf))
                succeed();
```

FILES

/usr/include/regexp.h

SEE ALSO

bs(1), ed(1), expr(1), grep(1), sed(1).

BUGS

The handling of *circf* is poor.

NAME

roman8 – map of ROMAN8 character set used by NLS

SYNOPSIS

ls /usr/lib/nls/*

DESCRIPTION

Roman8 is a map of the ROMAN8 character set, giving the octal, decimal, and hexadecimal equivalents of each character, to be printed as needed. It contains:

000	0	00	nul	001	1	01	soh
002	2	02	stx	003	3	03	etx
004	4	04	eot	005	5	05	enq
006	6	06	ack	007	7	07	bel
010	8	08	bs	011	9	09	ht
012	10	0a	nl	013	11	0b	vt
014	12	0c	np	015	13	0d	cr
016	14	0e	so	017	15	0f	si
020	16	10	dle	021	17	11	dc1
022	18	12	dc2	023	19	13	dc3
024	20	14	dc4	025	21	15	nak
026	22	16	syn	027	23	17	etb
030	24	18	can	031	25	19	em
032	26	1a	sub	033	27	1b	esc
034	28	1c	fs	035	29	1d	gs
036	30	1e	rs	037	31	1f	us
040	32	20	sp	041	33	21	!
042	34	22	"	043	35	23	#
044	36	24	$	045	37	25	%
046	38	26	&	047	39	27	'
050	40	28	(	051	41	29	)
052	42	2a	*	053	43	2b	+
054	44	2c	,	055	45	2d	-
056	46	2e	.	057	47	2f	/
060	48	30	0	061	49	31	1
062	50	32	2	063	51	33	3
064	52	34	4	065	53	35	5
066	54	36	6	067	55	37	7
070	56	38	8	071	57	39	9
072	58	3a	:	073	59	3b	;
074	60	3c	<	075	61	3d	=
076	62	3e	>	077	63	3f	?
100	64	40	@	101	65	41	A

102	66	42	B	103	67	43	C
104	68	44	D	105	69	45	E
106	70	46	F	107	71	47	G
110	72	48	H	111	73	49	I
112	74	4a	J	113	75	4b	K
114	76	4c	L	115	77	4d	M
116	78	4e	N	117	79	4f	O
120	80	50	P	121	81	51	Q
122	82	52	R	123	83	53	S
124	84	54	T	125	85	55	U
126	86	56	V	127	87	57	W
130	88	58	X	131	89	59	Y
132	90	5a	Z	133	91	5b	[
134	92	5c	\	135	93	5d	]
136	94	5e	^	137	95	5f	_
140	96	60	'	141	97	61	a
142	98	62	b	143	99	63	c
144	100	64	d	145	101	65	e
146	102	66	f	147	103	67	g
150	104	68	h	151	105	69	i
152	106	6a	j	153	107	6b	k
154	108	6c	l	155	109	6d	m
156	110	6e	n	157	111	6f	o
160	112	70	p	161	113	71	q
162	114	72	r	163	115	73	s
164	116	74	t	165	117	75	u
166	118	76	v	167	119	77	w
170	120	78	x	171	121	79	y
172	122	7a	z	173	123	7b	{
174	124	7c	|	175	125	7d	}
176	126	7e	~	177	127	7f	del
200	128	80		201	129	81	
202	130	82		203	131	83	
204	132	84		205	133	85	
206	134	86		207	135	87	
210	136	88		211	137	89	
212	138	8a		213	139	8b	
214	140	8c		215	141	8d	
216	142	8e	ss2	217	143	8f	ss3
220	144	90		221	145	91	
222	146	92		223	147	93	
224	148	94		225	149	95	
226	150	96		227	151	97	
230	152	98		231	153	99	
232	154	9a		233	155	9b	
234	156	9c		235	157	9d	
236	158	9e		237	159	9f	
240	160	a0		241	161	a1	À A accent grave
242	162	a2	Â A circumflex	243	163	a3	È E accent grave
244	164	a4	Ê E circumflex	245	165	a5	Ë E umlaut
246	166	a6	Î I circumflex	247	167	a7	Ï I umlaut

250	168	a8	´	accent acute	251	169	a9	`	accent grave
252	170	aa	ˆ	circumflex	253	171	ab	¨	umlaut accent
254	172	ac	˜	tilde accent	255	173	ad	Ù	U accent grave
256	174	ae	Û	U circumflex	257	175	af	₤	Italian lira
260	176	b0	¯	over line	261	177	b1		
262	178	b2			263	179	b3	°	degree
264	180	b4	Ç	C cedilla	265	181	b5	ç	c cedilla
266	182	b6	Ñ	N tilde	267	183	b7	ñ	n tilde
270	184	b8	¡	inv. exclamation	271	185	b9	¿	inv. question
272	186	ba	¤	general currency	273	187	bb	£	British pound
274	188	bc	¥	Japanese yen	275	189	bd	§	section
276	190	be	ƒ	Dutch guilder	277	191	bf	¢	U.S. cent
300	192	c0	â	a circumflex	301	193	c1	ê	e circumflex
302	194	c2	ô	o circumflex	303	195	c3	û	u circumflex
304	196	c4	á	a accent acute	305	197	c5	é	e accent acute
306	198	c6	ó	o accent acute	307	199	c7	ú	u accent acute
310	200	c8	à	a accent grave	311	201	c9	è	e accent grave
312	202	ca	ò	o accent grave	313	203	cb	ù	u accent grave
314	204	cc	ä	a umlaut	315	205	cd	ë	e umlaut
316	206	ce	ö	o umlaut	317	207	cf	ü	u umlaut
320	208	d0	Å	A degree	321	209	d1	î	i circumflex
322	210	d2	Ø	O crossbar	323	211	d3	Æ	AE ligature
324	212	d4	å	a degree	325	213	d5	í	i accent acute
326	214	d6	ø	o crossbar	327	215	d7	æ	ae ligature
330	216	d8	Ä	A umlaut	331	217	d9	ì	i accent grave
332	218	da	Ö	O umlaut	333	219	db	Ü	U umlaut
334	220	dc	É	E accent acute	335	221	dd	ï	i umlaut
336	222	de	ß	sharp s	337	223	df	Ô	O circumflex
340	224	e0	Á	A accent acute	341	225	e1	Ã	A tilde
342	226	e2	ã	a tilde	343	227	e3	Đ	D stroke
344	228	e4	đ	d stroke	345	229	e5	Í	I accent acute
346	230	e6	Ì	I accent grave	347	231	e7	Ó	O accent acute
350	232	e8	Ò	O accent grave	351	233	e9	Õ	O tilde
352	234	ea	õ	o tilde	353	235	eb	Š	S caron
354	236	ec	š	s caron	355	237	ed	Ú	U accent acute
356	238	ee	Ÿ	Y umlaut	357	239	ef	ÿ	y umlaut
360	240	f0	Þ	THORN	361	241	f1	þ	thorn
362	242	f2			363	243	f3		
364	244	f4			365	245	f5		
366	246	f6	-	long dash	367	247	f7	¼	one fourth
370	248	f8	½	one half	371	249	f9	ª	femin. ordinal
372	250	fa	º	masc. ordinal	373	251	fb	«	open guillemets
374	252	fc	■	solid	375	253	fd	»	close guillemets
376	254	fe	±	plus/minus	377	255	ff		

WARNINGS

Peripheral or software limitations may garble this manual page. Some printers and terminals do not support the ROMAN8 character set.

AUTHOR

Roman8 was developed by HP.

FILES

/usr/lib/nls/*

SEE ALSO

ascii(5), hpnls(5), kana8(5).

NAME

stat – data returned by stat/fstat system call

SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>

DESCRIPTION

The system calls *stat* and *fstat* return data whose structure is defined by this include file. The encoding of the field *st_mode* is defined in this file also.

The contents of the *stat* structure include the following members:

```
dev_t   st_dev;          /* ID of device containing a */
                         /* directory entry for this file */
ino_t   st_ino;          /* Inode number */
ushort  st_mode;         /* File mode; see mknod(2) */
short   st_nlink;        /* Number of links */
ushort  st_uid;          /* User ID of file owner */
ushort  st_gid;          /* Group ID of file group */
dev_t   st_rdev;         /* Device ID; this entry defined */
                         /* only for char or blk spec files */
off_t   st_size;         /* File size (bytes) */
time_t  st_atime;        /* Time of last access */
time_t  st_mtime;        /* Last modification time */
time_t  st_ctime;        /* Last file status change time */
                         /* measured in seconds since */
                         /* 00:00:00 GMT, Jan 1, 1970 */
uint    st_remote:1;     /* Set if file is remote */
dev_t   st_netdev;       /* ID of device containing */
                         /* network special file */
ino_t   st_netino;       /* Inode number of network special file */

#define S_IFMT   0170000 /* type of file */
#define S_IFDIR  0040000 /* directory */
#define S_IFCHR  0020000 /* character special */
#define S_IFBLK  0060000 /* block special */
#define S_IFREG  0100000 /* regular */
#define S_IFIFO  0010000 /* fifo */
#define S_ISUID  04000   /* set user id on execution */
#define S_ISGID  02000   /* set group id on execution */
#define S_ISVTX  01000   /* save swapped text even after use */
#define S_IREAD  00400   /* read permission, owner */
#define S_IWRITE 00200   /* write permission, owner */
#define S_IEXEC  00100   /* execute/search permission, owner */
#define S_ENFMT  02000   /* set file-locking mode to enforced */
```

FILES

/usr/include/sys/stat.h
/usr/include/sys/types.h

SEE ALSO

stat(2), types(5).

HARDWARE DEPENDENCIES

Series 500:

The following inode types are defined only for the Series 500:

```
#define    S_IFSRM    0150000
#define    S_IFNWK    0110000
```

The fields st_netdev and st_netino are not supported.

NAME

term – conventional names for terminals

DESCRIPTION

The environment variable TERM is used by certain commands (e.g., *tabs*(1), and is maintained as part of the shell environment (see *profile*(4), and *environ*(5)) The *tset*(1) command can be used to set the TERM variable. The name to which TERM is set usually exists as a compiled *terminfo* data base (see *terminfo*(4)). The following names are always available in the terminfo data base

hp	Minimal subset of the capabilities of all Hewlett-Packard terminals and terminal emulators supported on HP-UX systems. Note that entries for specific models of terminals are generally available, and that they often provide better use of the features of those terminals.
dumb	Generic name for terminals that lack reverse line-feed and other special escape sequences.
dialup	Generic name for dial-in ports connected to unknown terminals.

The TERM variable is also used by certain commands (e.g. *nroff*(1), *man*(1), *tabs*(1)), some of which use terminal and printer description files from the */usr/lib/terms* directory. One TERM name which has a file in this directory is

lp	Generic name for a line printer.

A basic terminal name can be up to eight characters chosen from A-Z, a-z, 0-9, and –. Terminal sub-models and operational modes are distinguished by suffixes beginning with a –. Names should generally be based on original vendors, rather than local distributors. A terminal acquired from one vendor should not have more than one distinct basic name.

Commands whose behavior depends on the type of terminal should accept arguments of the form –**T***term* where *term* is one of the names given above; if no such argument is present, such commands should obtain the terminal type from the environment variable **$TERM**, which, in turn, should contain *term*.

NOTES

The inclusion of other names in the terminfo data base or the /usr/lib/terms directory does not imply support of these devices.

AUTHOR

Term was developed by AT&T Bell Laboratories and the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

man(1), mm(1), nroff(1), sh(1), stty(1), tabs(1), tplot(1), tset(1), ul(1), curses(3X), profile(4), terminfo(4), ttytype(4), environ(5).

BUGS

The TERM variable is used differently by commands which originated from UCB code (such as *vi*(1) and *more*(1)) and commands which originated from Bell System III code (such as *nroff*(1) and *tabs*(1)). These different usages of TERM can be confusing.

NAME

types – primitive system data types

SYNOPSIS

#include <sys/types.h>

REMARKS

The example given on this page is a typical version; the type names are in general expected to be present, although exceptions can be described in HARDWARE DEPENDENCIES. In most cases the fundamental type which implements each typedef is implementation dependent, as long as source code which uses those typedefs need not be changed. In some cases the typedef is actually a shorthand for a commonly used type, and it will not vary.

DESCRIPTION

The data types defined in the include file are used in HP-UX system code; some data of these types are accessible to user code:

```
typedef struct { int r[1]; }   *physadr;
typedef long                   daddr_t;
typedef char                   *caddr_t;
typedef unsigned int           uint;
typedef unsigned short         ushort;
typedef ushort                 ino_t;
typedef short                  cnt_t;
typedef long                   time_t;
typedef long                   dev_t;
typedef long                   off_t;
typedef long                   paddr_t;
typedef long                   key_t;
```

Note that the defined names above are standardized, but the actual type to which they are defined may vary between HP-UX implementations.

The meanings of the types are:

physadr	used as a pointer to memory; the pointer is aligned to follow hardware-dependent instruction addressing conventions.
daddr_t	used for disk addresses except in an inode on disk, see *fs*(4).
caddr_t	used as an untyped pointer or a pointer to untyped memory.
uint	shorthand for *unsigned integer*.
ushort	shorthand for *unsigned short*.
ino_t	used to specify I-numbers.
cnt_t	used in some implementations to hold reference counts for some kernel data structures.
time_t	time encoded in seconds since 00:00:00 GMT, January 1, 1970.
dev_t	specifies kind and unit number of a device, encoded in two parts known as major and minor.
off_t	offsets measured in bytes from the beginning of a file.
paddr_t	used as an integer type which is properly sized to hold a pointer.
key_t	the type of a key used to obtain a message queue, semaphore, or shared memory identifier, see *stdipc*(3C).

SEE ALSO

fs(4), stdipc(3C).

NAME

values – machine-dependent values

SYNOPSIS

#include <values.h>

DESCRIPTION

This file contains a set of manifest constants, conditionally defined for particular processor architectures.

The model assumed for integers is binary representation (one's or two's complement), where the sign is represented by the value of the high-order bit.

BITS(*type*)	The number of bits in a specified type (e.g., int).
HIBITS	The value of a short integer with only the high-order bit set (in most implementations, 0x8000).
HIBITL	The value of a long integer with only the high-order bit set (in most implementations, 0x80000000).
HIBITI	The value of a regular integer with only the high-order bit set (usually the same as HIBITS or HIBITL).
MAXSHORT	The maximum value of a signed short integer (in most implementations, 0x7FFF ≡ 32767).
MAXLONG	The maximum value of a signed long integer (in most implementations, 0x7FFFFFFF ≡ 2147483647).
MAXINT	The maximum value of a signed regular integer (usually the same as MAXSHORT or MAXLONG).
MAXFLOAT, LN_MAXFLOAT	The maximum value of a single-precision floating-point number, and its natural logarithm.
MAXDOUBLE, LN_MAXDOUBLE	The maximum value of a double-precision floating-point number, and its natural logarithm.
MINFLOAT, LN_MINFLOAT	The minimum positive value of a single-precision floating-point number, and its natural logarithm.
MINDOUBLE, LN_MINDOUBLE	The minimum positive value of a double-precision floating-point number, and its natural logarithm.
FSIGNIF	The number of significant bits in the mantissa of a single-precision floating-point number.
DSIGNIF	The number of significant bits in the mantissa of a double-precision floating-point number.

FILES

/usr/include/values.h

SEE ALSO

intro(3), math(5).

NAME

varargs – handle variable argument list

SYNOPSIS

#include <varargs.h>

va_alist

va_dcl

void va_start(pvar)
va_list pvar;

type **va_arg(pvar,** *type***)**
va_list pvar;

void va_end(pvar)
va_list pvar;

DESCRIPTION

This set of macros allows portable procedures that accept variable argument lists to be written. Routines that have variable argument lists (such as *printf*(3S)) but do not use *varargs* are inherently nonportable, as different machines use different argument-passing conventions.

va_alist is used as the parameter list in a function header.

va_dcl is a declaration for *va_alist*. No semicolon should follow *va_dcl*.

va_list is a type defined for the variable used to traverse the list.

va_start is called to initialize *pvar* to the beginning of the list.

va_arg will return the next argument in the list pointed to by *pvar*. *Type* is the type the argument is expected to be. Different types can be mixed, but it is up to the routine to know what type of argument is expected, as it cannot be determined at runtime.

va_end is used to clean up.

Multiple traversals, each bracketed by *va_start* ... *va_end,* are possible.

EXAMPLE

This example is a possible implementation of *execl* (on *exec*(2):

```
#include <varargs.h>
#define MAXARGS     100

/*      execl is called by
                execl(file, arg1, arg2, ..., (char *)0);
*/
execl(va_alist)
va_dcl
{
        va_list ap;
        char *file;
        char *args[MAXARGS];
        int argno = 0;

        va_start(ap);
        file = va_arg(ap, char *);
        while ((args[argno++] = va_arg(ap, char *)) != (char *)0)
                ;
        va_end(ap);
```

```
                return execv(file, args);
        }
```

SEE ALSO

exec(2), vprintf(3S).

BUGS

It is up to the calling routine to specify how many arguments there are, since it is not always possible to determine this from the stack frame. For example, *execl* is passed a zero pointer to signal the end of the list. *Printf* can tell how many arguments are there by the format.

It is non-portable to specify a second argument of *char*, *short*, or *float* to *va_arg*, since arguments seen by the called function are not *char*, *short*, or *float*. C converts *char* and *short* arguments to *int* and converts *float* arguments to *double* before passing them to a function.

NAME

intro – introduction to special files

DESCRIPTION

This section describes various special files that refer to specific HP peripherals and device drivers. The names of the entries are generally derived from the type of device being described (disk, plotter, etc.), not the names of the special files themselves. Characteristics of both the hardware device and the corresponding HP-UX device driver are discussed where applicable.

The devices are divided into two categories, **unblocked** and **blocked.** An unblocked device is also called a **raw** or a character mode device. An unblocked device, such as a line printer, uses a character special file.

Blocked devices, as the name implies, transfer data in blocks via the systems normal buffering mechanism. Block devices use block special files.

For specific details about the default special files shipped with your system, consult the System Administrator Manual for your system.

You associate the name you want with a specific device when you create a special file for that device using the *mkdev*(1M) and *mknod*(1M) commands. When creating special files, it is recommended that the following naming convention be followed. For disk and tape, it is identical with that used on other UNIX systems, and is independent of the hardware.

The following format is for 9 track tape device file names:

/dev/{r}mt/(c#d)#[hml]{n}

where **r** indicates a raw device, **c#d** indicates the controller number (which is optionally specified by the system administrator), # is the device number, **hml** indicates the density (**h** (high) for 6250 bpi, **m** (medium) for 1600 bpi, and l (low density) for 800 bpi), and **n** indicates no rewind on close, e.g., **/dev/mt/2mn**.

The following format is for hard disk device file names:

/dev/{r}dsk/(r)(c#d)#s#

where **r** indicates a raw interface to the disk, the second **r** indicates that this disk is on a remote system, the **c#d** indicates the controller number (which is optionally specified by the system administrator), and **#s#** indicates the drive and section numbers, respectively.

HARDWARE DEPENDENCIES

Series 500:

You cannot open a block special file for reading or writing.

The IBM format is not supported for the HP 9895A.

WARNINGS

There have been several other naming conventions in the past for similar devices. Using *ln* (on *cp*(1)) to create a link between the old name and the new standard name is useful as a temporary expedient until all the programs using the old naming convention have been converted.

In general, device drivers are not portable across systems; however, every effort has been made to make their behavior portable. Due to variation in hardware, this is not always possible. Programs which use these drivers directly are at higher than average risk of not being portable.

SEE ALSO

hier(5).

The introduction to this manual.

The System Administrator Manual for your system.

NAME

console – system console interface

DESCRIPTION

/dev/console is a generic name given to the system console. It is usually linked to a particular machine dependent special file. It provides a basic I/O interface to the system console through the *termio* interface.

SEE ALSO

termio(7).

NAME

ct – cartridge tape access

DESCRIPTION

This page describes the actions of the general HP-UX cartridge tape drivers when referring to a cartridge tape as either a block- or character-special (raw) device.

Cartridge tapes are designed to work optimally as "streaming" devices, and are not designed to start and stop frequently. Technically, they are "random access" devices, like disks, but such access is both less efficient and more stressful than streaming mode. Thus it is possible to use a cartridge tape as a file system, or in general access it randomly, but such use will more rapidly wear either or both of the tape drive and the media.

Cartridge tape units in either Command-Set 80 disk drives or in stand-alone devices can be accessed as blocked or raw devices.

Block special files access cartridge tapes via the system's normal buffering mechanism. Buffering is done in such a way that concurrent access through multiple opens or a mount of the same physical device do not get out of phase. Block special files may be read and written without regard to physical cartridge tape records. Each I/O operation results in one or more logical block transactions. In general, this mode is not recommended as it stresses the hardware.

There is also a *raw* interface via a character special file which provides for direct transmission between the cartridge tape and the user's read or write buffer. A single read or write operation results in exactly one transaction. Therefore raw I/O is considerably more efficient when many bytes are transmitted in a single operation because blocked cartridge tape access requires potentially several transactions and does not transmit directly to user space.

In raw I/O, there may be implementation dependent restrictions on the alignment of the user buffer in memory and its maximum size. Also, each transfer must occur on a record boundary and must read a whole number of records. The record size is a hardware dependent value.

Selecting the proper buffer size when accessing a cartridge tape device through the raw interface is critical to the performance of the cartridge tape device and other devices connected on the same HPIB. A large buffer in certain situations can increase performance but has the potential to block other devices on the HPIB until all the data for a request has been transferred. On the other hand when a small buffer is used and the application is unable to keep the cartridge tape device streaming, performance and the wear and tear of the device suffer because of tape repositioning. The optimal solution is to keep the tape streaming while using a small buffer. To select the proper buffer size, consider two factors: the cartridge tape device being accessed and the application which is accessing the cartridge tape device.

Some cartridge tape units (see HARDWARE DEPENDENCIES) support a feature called immediate report mode. During writing, this mode enables the drive to complete a write transaction with the host before the data has actually been written to the tape from the drive's buffer. This allows the host to start gathering data for the next write request while the data for the previous request is still in the process of being written. During reading, this mode enables the drive to read ahead after completing a host read request. This allows the drive to gather data for future read requests while the host is still processing data from the previous read request. When data is requested or supplied at a sufficient rate, immediate report mode allows the drive to stream the tape continuously across multiple read/write requests, as opposed to having to reposition the tape between each read/write request. Repositioning adds to the wear and tear of the cartridge tape device and decreases the performance. Some cartridge tape devices (see HARDWARE DEPENDENCIES) do not support immediate report mode and as such cannot stream across multiple requests.

If the cartridge tape device being accessed supports immediate report mode and the application can maintain a data rate that allows the cartridge tape device to stream multiple requests, a small buffer (1 Kbyte to 12 Kbytes) is suggested so that the HPIB is not blocked for a significant amount of time. For cartridge tape devices that do not support the immediate report mode or applications that cannot maintain a data rate that allows the cartridge tape device to stream multiple requests, a large buffer (64 Kbytes) is suggested so that the number of tape repositions is reduced.

Each raw access is independent of other raw accesses and of block accesses to the same physical device. Thus, transfers are not guaranteed to occur in any particular order. Having multiple programs access the cartridge tape is in effect random access, and is subject to the warnings above.

In raw I/O, each operation is completed to the device before the call returns. For block-mode writes, the data may be cached until it is convenient for the system to write it. In addition, block-mode reads potentially do a one (or more) block read-ahead. The interaction of block-mode and raw access to the same cartridge tape is not specified, and in general is unpredictable. Because block-mode writes can be delayed, it is possible for a program to generate requests much more rapidly than the drive can actually process them. Flushing a large number of requests could take several minutes, and during that time the system will not have use of the buffers taken by these requests, and thus will suffer a possibly severe performance degradation. If the tape is integral with the system disk, very little disk activity may be possible until the buffers are flushed.

Cartridge tape device file names are in the following format:

/dev/(r)ct/(r)c#(d#)(s#)

where the first **r** indicates a raw interface to the cartridge tape, the second **r** is reserved to indicate that this cartridge tape is on a remote system, the **c#** indicates the controller number, the d# optionally indicates the drive, and the **s#** optionally indicates a section number. The assignment of controller, drive, and section numbers is described in the System Administrator's manual for your system.

WARNINGS

Like disks, the cartridge tape units in Command Set 80 (CS/80) disk drives can be accessed as blocked or raw devices. However, using a cartridge tape as a file system will severely limit the life expectancy of the tape drive. Tapes should be used only for system back-up and other needs where data must be stored on tape for transport or other purposes.

HARDWARE DEPENDENCIES

Series 500

Block-special devices cannot be opened for I/O.

HP7941CT/HP9144A/HP35401

These cartridge tape devices support the immediate report mode.

HP7942/HP7946

These cartridge tape devices support the immediate report mode. The use of a small buffer size is not recommended with these shared controller devices when there is simultaneous access to the disk, because the disk accesses will prevent proper tape streaming.

HP7908/HP7911/HP7912/HP7914

These cartridge tape devices do not support the immediate report mode.

AUTHOR

Ct was developed by HP and AT&T.

SEE ALSO

mkdev(1M), mknod(1M), tcio(1), disk(7), intro(7), mt(7).

NAME

diag0 – diagnostic interface to I/O subsystem

SYNOPSIS

pseudo-device diag0

DESCRIPTION

Diag0 is the diagnostic global port for the system. All diagnostic events are sent to it on subqueue one.

The only commands that *diag0* supports are *open*(2), *close*(2), and *read*(2). The *open*(2)command is exclusive and the opening process must have an effective super-user user ID. A read of the diagnostic device file returns a diagnostic event message. If a diagnostic event has been logged since the last read, the next read call will be satisfied with that event message and will return immediately. If no diagnostic events are queued and waiting, the process that made the read call is put to sleep waiting for an event to be sent to the diagnostic port. When an event finally arrives, the sleeping process is awakened and the read is satisfied. If during the read an error is encountered, the read will be terminated and errno will be set.

RETURNS

Upon successful completion, a zero is returned. Otherwise, a value of -1 is returned and errno is set to indicate the error.

Diag0 does not support any *ioctl*(2) calls or *write*(2) commands. If either command is issued then the following error is returned:

[ENODEV]	ioctl not supported.
[EBADF]	write not supported.

When opening *diag0* the following errors may be returned:

[EPERM]	permission denied, not superuser.
[EBUSY]	device file already opened.

When attempting a read of *diag0* the following errors may be returned:

[EIO]	error occurred while retrieving message.
[ENXIO]	unknown message type received.

AUTHOR

Diag0 was developed by HP.

FILES

/dev/diag0
/hp-ux

SEE ALSO

open(2), close(2), read(2).

NAME

disk – direct disk access

DESCRIPTION

This page describes the actions of the general HP-UX disk drivers when referring to a disk as either a block-special or character-special (raw) device.

Block special files access disks via the system's normal buffering mechanism. Buffering is done in such a way that concurrent access through multiple opens or a mount of the same physical device do not get out of phase. Block special files may be read and written without regard to physical disk records. Each I/O operation results in one or more logical block transactions. The size of the logical block request can be found in f_blksize as returned by *ustat*(2).

There is also a *raw* interface via a character special file which provides for direct transmission between the disk and the user's read or write buffer. A single read or write operation results in exactly one transaction. Therefore raw I/O is considerably more efficient when many bytes are transmitted in a single operation because blocked disk access requires potentially several transactions and does not transmit directly to user space.

In raw I/O, there may be implementation dependent restrictions on the alignment of the user buffer in memory. Also, each transfer must occur on a sector boundary and must read a whole number of sectors. The sector size is a hardware dependent value.

Each raw access is independent of other raw accesses and of block accesses to the same physical device. Thus, transfers are not guaranteed to occur in any particular order.

In raw I/O, each operation is completed to the device before the call returns. For block-mode writes, the data may be cached until it is convenient for the system to write it. In addition, block-mode reads potentially do a one (or more) block read-ahead. The interaction of block-mode and raw access to the same disk is not specified, and in general is unpredictable.

Disk device file names are in the following format:

/dev/{r}dsk/(r)(c#d)#s#

where **r** indicates a raw interface to the disk, the second **r** should not be used and is reserved for future use, the **c#d** indicates the controller number (which is optionally specified by the system administrator), and **#s#** indicates the drive and section numbers, respectively. The assignment of controller, drive, and section numbers is described in the System Administrator's manual for your system.

WARNING

On some systems, having both a mounted file system and a block special file open on the same device is asking for trouble; this should be avoided if possible. This is because it may be possible for some files to have private buffers in some systems.

HARDWARE DEPENDENCIES

Series 500
: Block-special devices cannot be opened for I/O.

Series 800
: All transfers must begin on a DEV_BSIZE boundary.

AUTHOR

Disk was developed by HP and AT&T.

SEE ALSO

mkdev(1M), mknod(1M), ct(7), intro(7).

HP-UX System Administrator's Manual included with your system.

NAME

afi – asynchronous FIFO interface

SYNOPSIS

#include <ioctl.h>
#include <sys/gpio.h>

ioctl(fildes, IO_CONTROL, gpio_control)
int fildes;
struct io_ctl_status *gpio_control;

ioctl(fildes, GPIO_COMMAND, gpio_cmd)
int fildes;
struct gpio_command *gpio_cmd;

ioctl(fildes, IO_STATUS, gpio_status)
int fildes;
struct io_ctl_status *gpio_status;

ioctl(fildes, IO_ENVIRONMENT, gpio_env)
int fildes;
struct io_environment *gpio_env;

DESCRIPTION

AFI is a general purpose I/O interface supporting high speed parallel communication with an arbitrary peripheral. It includes sixteen data lines, two handshake lines for transfer protocol, a peripheral-controlled interrupt line, a user-controlled peripheral reset line, and several lines for application-dependent control and status. This section describes the use of the GPIO driver in the HP-UX system.

Transfer Requests: The standard *read*(2) and *write*(2) requests (q.v.) are used for data transfer over AFI.

Control Requests: A user can configure the AFI driver via *ioctl*(2) calls:

```
struct io_ctl_status {
  int type;
  int arg[3];
} gpio_control;

ioctl (filedes, IO_CONTROL, &gpio_control);
```

In the *io_ctl_status* structure, the *type* field specifies the type of control function to be done, while the *arg* array holds any associated arguments. The allowable values for *type* are:

GPIO_TIMEOUT
: Set the timeout. If any transaction for this file takes longer than *arg[0]* microseconds, it will be aborted with a status of ETIMEDOUT returned to the user; this is mainly used for detecting device failure. A timeout of 0 is equivalent to infinity; i.e., no transaction will time out.

GPIO_WIDTH
: Set the width of the interface. This request specifies the number of valid data lines on transfers; *arg[0]* holds the desired interface width in bits. All future read requests will inspect only the least significant *arg[0]* data lines, and all future writes will present data on only those lines. The state of all other data lines is indeterminate.

GPIO_LOOPBACK
: Initiate a loopback test on the card.

GPIO_EDGE
: Select the triggering edge for the peripherals. *arg[0]* has one of the following values:

LDG_EDG	Trigger off the leading edge.
FAL_EDG	Trigger off the trailing edge.

GPIO_SIGNAL_MASK
: Define signaling events. This request allows the calling process to receive a signal when some event occurs on the GPIO. In *arg[0]* is an event mask, constructed by *or*-ing flags from the list below. Each request overwrites the previous mask for the file; therefore events can be disabled by using a zero. Currently any non-zero value will enable interrupt mask.

  When the interrupt mask is enabled and interrupt line is asserted, the process will be sent SIGEMT; the user should set up a handler to trap this signal (see *signal*(2)). The reason for interrupt can be obtained via the **IO_STATUS** request GPIO_SIGNAL_MASK.

GPIO_LOCK
: Lock/unlock the GPIO interface. Locking is done by opening the device file with the O_EXCL flag set. See *open*(2).

GPIO_RESET
: Reset the device/bus. This request restores a device or bus to a known state; *arg[0]* has one of the following values:

DEVICE_CLR	Reset the state machine on the card and put the device in a known state. Assert Device Clear(DCL).
HW_CLR	Reset card interface hardware

Status Requests: These requests are used to obtain information about the state of a device or the GPIO in general. They use a calling sequence similar to that of control requests:

```
struct io_ctl_status   gpio_status;

ioctl (filedes, IO_STATUS, &gpio_status);
```

Type specifies the type of information to obtain, while the *arg* array holds clarification data; the allowable status requests for GPIO are:

GPIO_TIMEOUT
: Return the interface's timeout in microseconds in *arg[0]*.

GPIO_WIDTH
: Return the interface's path width in bits in *arg[0]*.

GPIO_LOCK
: If the device is locked to a process, return that process id in *arg[0]*. If the device is not locked, *arg[0]* holds a -1.

GPIO_INTERFACE_TYPE
: Return the type of interface. This will return one of two values in *arg[0]*:

  GPIO_INTERFACE
  : the open file is a GPIO *raw bus* file

  GPIO_DEVICE
  : the open file is a GPIO *device* file

Extended Status Request: In the event that the user wishes to obtain several of the status variables in one request, the following request can be used:

```
struct io_environment {
  int interface_type;
  int timeout;
  int status;
  int term_reason;
  int read_pattern;
  int signal_mask;
  int width;
  int speed;
  int locking_pid;
} gpio;

ioctl (filedes, IO_ENVIRONMENT, &gpio_env);
```

Default Configuration: The default configuration of any GPIO file is:

Timeout	Infinite
Path Width	16 bits
Enabled Signals	None
Locking	Unlocked

RETURNS

A −1 return value for a driver request indicates an error occurred; *errno* is set to indicate the reason. In addition to those errors defined in *open*(2), *close*(2), *read*(2), *write*(2), and *ioctl*(2), a driver request can fail if any of the following are true:

[EACCES]	The interface is not active controller or system controller, and this request requires it.
[EBUSY]	Either the interface is currently locked (see GPIO_LOCK), or the driver has no software resources available.
[EINTR]	An interface power failure occurred during the processing of this request; the device might have lost state.
[EIO]	Some unclassified error occurred.
[ENXIO]	There is no bus interface associated with the device file.
[ETIMEDOUT]	The transaction did not complete within the timeout specified.

In addition, the following messages can appear on the system console as a result of errors:

gpio0 unit %d: device adapter failure.

The bus hardware is no longer functioning. This is a fatal error.

gpio0 unit %d: unexpected message (message type = %d, from port %d).

The driver received an unclassified message.

AUTHOR

GPIO was developed by HP.

FILES

/dev/gpio/*

SEE ALSO

ioctl(2), signal(2), particular device documentation.

NAME

CRT graphics – information for CRT graphics devices

Remarks:

This information is valid for Series 200 and 300 only.

DESCRIPTION

CRT graphics devices are frame-buffer based raster displays. These devices use memory-mapped I/O to obtain much higher performance than is possible with tty-based graphics terminals. CRT graphics devices should only be accessed through the STARBASE libraries. They cannot be piped or redirected to because they are not serial devices.

Special (device) files for CRT graphics devices are character special files with major number 12.

The minor number for CRT graphics devices is of the form:

```
0xSSTTXX
```

where SS is a one-byte select code number, TT is a one-byte type specifier, and XX is zero or contains device-specific information as defined in the appropriate Starbase Device Driver manual.

The type field in the minor number is defined as follows:

0 auto-configures to one of the following:

a) low resolution graphics device at physical address 0x520000 (if present).

b) high resolution graphics device at physical address 0x560000 if low resolution device at 0x520000 not present.

1 high resolution graphics device at physical address 0x560000 (unless there is no low resolution device at 0x520000, in which case type 1 is invalid).

2 high or low resolution graphics device at the select code specified by the select code field in the minor number.

Communication with a CRT graphics device is begun with an *open* system call. Multiple processes may concurrently have the graphics device open.

Close shuts down the file descriptor associated with the graphics device. If the close is for the last system wide open on the device then the graphics device is also unmapped from the user address space; otherwise it is left mapped into the user address space (see GCUNMAP below).

Read and *write* system calls are undefined and will always return ENXIO.

Ioctl is used to control the graphics device. The valid ioctl commands (see **<sys/graphics.h>**) are:

GCID	Return the identity of the CRT graphics device. Possible identities are: 1 = 98204A 2 = 9826A 3 = 9836A 4 = 9836C 5 = 98627A 6 = 98204B 7 = 9837 8 = 98700 9 = hp9000s300 displays 10 = 98720
GCON, GCOFF	Turn graphics "on" or "off". May be a no-op for some devices.
GCAON, GCAOFF	turn alpha "on" or "off". May be a no-op for some devices.

GCMAP	map the CRT graphics device into the user address space at the address specified in the ioctl argument. The argument is 'char **arg'. The value *arg is used as a requested address. The actual mapping address is then returned in *arg. If *arg is 0 then the system selects the first available address (see GCLOCK/GCUNLOCK below).
GCUNMAP	remove the mapping of the CRT graphics device from the user address space.
GCLOCK	ensure exclusive use of the CRT graphics device.
GCUNLOCK	relinquish exclusive use of the CRT graphics device.

For all frame buffers the data bytes scan from left to right and from top to bottom. Some displays map in control areas which must be skipped over to reach the frame buffer. Some devices map individual bits to pixels, (dots on the screen.) Some map bytes or parts of bytes to pixels. Lsb stands for least significant bit; msb stands for most significant bit.

HP 98204A	There are 300 lines of 100 bytes each. Only the odd numbered bytes are used. There is a one bit per pixel, with msb left, and lsb right.
HP 98204B	There are 390 lines of 64 bytes each. There is a one bit per pixel, with msb left, and lsb right.
HP 98627A	Starting 0x8000 bytes from the base address, there are 3 buffers of 0x8000 bytes each. The 3 buffers are the data for red, green, and blue. There is one bit per pixel, with msb left, and lsb right. There are 64 bytes per line. The number of lines depends on the setting of control registers.
Model 237	Starting 0x10000 bytes from the base address, there are 768 lines of 1024 bytes each. There is one pixel per byte. The lsb of each byte corresponds to a pixel.
HP 98700	Starting 0x10000 bytes from the base address, there are 768 lines of 1024 bytes each. There is one pixel per byte. Each byte corresponds to the color map index of a pixel.
HP 98720	Starting 0x20000 bytes from the base address, there are 1024 lines of 2048 bytes each (1280 bytes by 1024 lines are displayable); one pixel per byte.

Series 300 Displays:

These displays have registers describing the display size. The following code computes frame buffer width and height and determines what portion of the frame buffer is being displayed:

```
/* unsigned char *base = <base address for display mapping>; */
buffer_width = (base[5] << 8) + base[7];
buffer_height = (base[9] << 8) + base[11];
displayed_width = (base[13] << 8) + base[15];
displayed_height = (base[17] << 8) + base[19];
not_square = ((base[23] & 1) == 1);
```

Starting 0x10000 bytes from the base address, there are <buffer_height> lines of <buffer_width> bytes each. There is one pixel per byte. Each byte corresponds to the color map index of a pixel. On a monochrome display, the byte value is either 0, (black), or 1 (white). If ((base[23] & 1) == 1) then pixels are twice as high as they are wide, and may be used in pairs to produce square double pixels.

One shared memory descriptor (see *shmget*(2)) is used for each graphics device. Each shared memory descriptor is accessible only through its graphics interface. Thus, any attempt to access them through *shmat*(2)), *shmctl*(2)), *shmdt*(2)), etc. results in EACCESS errors.

ERRORS

[ENXIO]	no such device or read/write not supported.
[ENOSPC]	cannot allocate required resources for mapping.
[ENOMEM]	cannot allocate sufficient memory for mapping.
[ENOTTY]	bad ioctl command, or an ioctl was attempted on an open file.

SEE ALSO

mknod(1M).

NAME

hpib – Hewlett-Packard Interface Bus driver

SYNOPSIS

#include <ioctl.h>
#include <sys/hpibio.h>

ioctl (fildes, IO_CONTROL, hpib_control)
int fildes;
struct io_ctl_status *hpib_control;

ioctl (fildes, HPIB_COMMAND, hpib_cmd)
int fildes;
struct hpib_command *hpib_cmd;

ioctl (fildes, IO_STATUS, hpib_status)
int fildes;
struct io_ctl_status *hpib_status;

ioctl (fildes, IO_ENVIRONMENT, hpib_env)
int fildes;
struct io_environment *hpib_env;

DESCRIPTION

HP-IB is Hewlett-Packard's implementation of the Institute of Electrical and Electronic Engineers Standard Digital Interface for Programmable Instrumentation (IEEE Std 488-1978). This section describes the use of the HP-IB driver in the HP-UX system.

Auto-addressed Files vs. Raw Bus Files

A major distinction is made in the HP-UX driver between "auto-addressed" files and "raw bus" files. An "auto-addressed" file is associated with a specified address on the HP-IB. The user need not be concerned with any HP-IB addressing or commands; the driver handles device addressing and unaddressing during data transfers. However, the user is limited to transactions to and from a single device. A "raw bus" file, on the other hand, gives the user access to the entire HP-IB; responsibility for all commands and addressing lies with the user. The raw bus file is typically used to access multiple devices on the same bus, as well as provide universal device commands, such as interface clear and parallel poll.

Even though differences exist between auto-addressed and raw bus files, the user/driver interface is consistent across both types. Therefore, each category of requests will be presented with separate subsections for auto-addressed and raw bus files.

Naming Convention

HP-IB device file names are in the following format:

/dev/hpib/#(a#)

where the first # specifies the bus number (which is specified by the system administrator) and the second # specifies the address on that bus. Device files without an address suffix denote the raw bus. Files with the address suffix are auto-addressed.

Transfer Requests

The standard *read*(2) and *write*(2) requests are used for data transfer over HP-IB. However, their actions are slightly different for each type of file. Raw bus files place data directly onto the bus. No addressing or unaddressing of devices is done; this is the user's responsibility.

On the other hand, transactions with auto-addressed files have all addressing done by the driver. The actual sequence of events is:

UNL, <device addressing>, <data>, <terminator>

Write requests end on count. Optionally, the End or Identify (EOI) line can be asserted on the last byte written, via the HPIB_EOI request. All reads end on count or device assertion of EOI. In addition, a single character can be defined to end the read buffer via the HPIB_READ_PATTERN request.

Control Requests

This form of request takes some action on the bus. All requests have the same format:

```
struct io_ctl_status {
  int type;
  int arg[3];
} hpib_control;

ioctl (fildes, IO_CONTROL, &hpib_control);
```

In the *io_ctl_status* structure, the *type* field specifies the type of control function to be done, while the *arg* array holds any associated arguments. The allowable values for *type* are:

HPIB_TIMEOUT
: Set the timeout. If any transaction for this file takes longer than *arg[0]* microseconds, it will be aborted with a status of ETIMEDOUT returned to the user. This is mainly used for detecting device failure. A timeout of **0** is equivalent to infinity, i.e., no transaction will time out.

HPIB_WIDTH
: Set the width of the interface. This request specifies the number of valid data lines on transfers; *arg[0]* holds the desired interface width in bits. All future read requests will inspect only the least significant *arg[0]* data lines, and all future writes will present data on only those lines. The state of all other data lines is indeterminate.

HPIB_SPEED
: Set the transfer speed of the interface. The desired data transfer speed in kilobytes per second is specified in *arg[0]*. Note that this value is ADVISORY only, and is typically used by the driver to determine the method of data transfer (e.g., DMA or non-DMA).

HPIB_EOI
: Enable/disable EOI assertion on writes. If *arg[0]* is nonzero, all subsequent writes will end with EOI asserted on the last byte transferred. A zero *arg[0]* disables EOI assertion.

HPIB_READ_PATTERN
: Enable/disable pattern matching on reads. If *arg[0]* is nonzero, all subsequent reads will terminate when the pattern specified in *arg[1]* is encountered in the input stream. This termination condition is subject to all other termination conditions in effect for the file. Only the *n* least significant bits of the pattern are used in the match, where *n* is the interface's current width, set via HPIB_WIDTH. A zero *arg[0]* disables read pattern matching.

HPIB_SIGNAL_MASK
: Define signaling events. This request allows the calling process to receive a signal when some event occurs on the HP-IB. The event(s) are specified by *or*-ing together bits from the list below, and placing the mask in *arg[0]*. All these events can be enabled on raw bus files, but only ST_SRQ and ST_PPOLL apply to auto-addressed files.

ST_SRQ	Signal on assertion of Service Request (SRQ).
ST_PPOLL	Signal when device responds to Parallel Poll.

ST_REN Signal when interface enters remote state.

ST_ACTIVE_CTLR
Signal when interface becomes active controller.

ST_TALK Signal when interface is addressed to talk.

ST_LISTEN Signal when interface is addressed to listen.

ST_IFC Signal on assertion of Interface Clear (IFC).

ST_DCL Signal on receipt of Device Clear (DCL).

ST_GET Signal on receipt of Group Execute Trigger (GET).

When any flagged event occurs, the process will be sent SIGEMT. The user should set up a handler to trap this signal, via *signal*(2) or *sigvector*(2). The reason for interrupt can be obtained via the IO_STATUS request HPIB_SIGNAL_MASK. Each request overwrites the previous mask for the file, therefore events can be disabled by using a zero *arg[0]*.

If ST_PPOLL is flagged, the user supplies additional information in the *arg* array. For raw bus files, the low order bytes of *arg[1]* and *arg[2]* contain eight-bit masks with each bit corresponding to a Data I/O (DIO) line, with the least significant bit mapped to DIO line 0. When a device responds to parallel poll, it asserts the appropriate line; *arg[1]*'s bits indicate the sense of this assertion. Set bits in *arg[2]* indicate that the corresponding address is capable of response to polling. For auto-addressed files, *arg[1]* specifies the sense of the assigned device's response to parallel poll.

HPIB_LOCK Lock/unlock the HP-IB interface. Note that the semantics for this request have not yet been defined. Check the HARDWARE DEPENDENCIES section.

HPIB_ADDRESS
Set the HP-IB address to which the interface will respond when it is not active controller. The bus address is set via *arg[0]*, and must be between 0 to 30 decimal. This request is applicable only to raw bus files.

HPIB_RESET Reset the device/bus. This request restores a device or bus to a known state, depending on which of the following values is in *arg[0]*:

DEVICE_CLR Address the device and send a selective device clear (SDC). This applies only to auto-addressed files.

BUS_CLR Assert Interface Clear (IFC) and Remote Enable (REN), and clear Attention (ATN).

HW_CLR Reset bus interface hardware . This applies only to raw bus files.

HPIB_PPOLL_RESP
Control interface's response to parallel poll. When the interface is not acting as active controller, it can be enabled to respond to parallel polling by the current active controller. If *arg[0]* is nonzero, the interface will respond to parallel poll. The DIO line on which the card is to respond is specified in *arg[1]*; it has a value between **0** and **7**. The sense of the response is determined by *arg[2]*. An *arg[0]* of zero disables the interface's response to parallel poll.

For auto-addressed files, the file's associated device address is configured, rather than the interface.

HPIB_PPOLL_IST
Enable/disable response to parallel poll. If *arg[0]* is nonzero, the interface will

respond to parallel poll. An *arg[0]* of zero disables the interface's response. This differs from the previous request, in that the sense and address of the interface's response is unchanged. This request applies only to raw bus files.

HPIB_REN
: Place a device into/out of the remote state. For a raw bus file, this request merely asserts or deasserts the Remote Enable line, if *arg[0]* is nonzero or zero respectively. For auto-addressed files, a nonzero *arg[0]* asserts the Remote Enable line and addresses the device. If *arg[0]* is zero, the device is removed from the remote state by sending it a Go To Local command (GTL).

HPIB_SRQ
: Request service. This request causes the interface to assert the Service Request line (SRQ), until it is serially polled. At that time it will respond with the status byte given in *arg[1]*. This request applies only to raw bus files.

HPIB_PASS_CONTROL
: Pass active control of the bus. If the interface is currently active controller, this request relinquishes control of the bus, passing it instead to the device at the bus address in *arg[0]*. Passing control should be done with care, since it is not possible to detect if the named device can indeed assume bus control. This request applies only to raw bus files.

HPIB_GET_CONTROL
: Become active controller. This request causes the interface to assert Interface Clear (IFC) and Remote Enable (REN) as a means of regaining control of the HP-IB. It applies only to raw bus files.

Transparent Bus Request

This request allows a user to send direct commands over the HP-IB; it should be used with care, since without proper precautions the bus can be placed in an unusable state.

The format of the transparent bus request is:

```
struct hpib_command {
  int length;
  char buffer[MAX_HPIB_COMMANDS];
} hpib_cmd;

ioctl (fildes, HPIB_COMMAND, &hpib_cmd);
```

The effect of this call is to transmit the *length* bytes of data in *buffer* over the HP-IB with Attention (ATN) asserted. On completion of the request, ATN will remain asserted.

For commands sent through an auto-addressed file, *buffer* is surrounded with the appropriate device addressing. What appears on the bus is:

UNL, TALK CIC, LISTEN device, <buffer>

This differs from the approach toward a raw bus file. For such files, the *buffer* is merely placed on the bus with ATN asserted, with no addressing or unaddressing.

Status Requests

These requests are used to obtain information about the state of a device or the HP-IB in general. They use a calling sequence similar to that of control requests:

```
struct io_ctl_status hpib_status;

ioctl (fildes, IO_STATUS, &hpib_status);
```

Type specifies the type of information to obtain, while the *arg* array holds clarification data. The allowable status requests for HP-IB are:

HPIB_ADDRESS
: Return the bus address associated with the file in *arg[0]*.

HPIB_TIMEOUT
: Return the interface's timeout in microseconds in *arg[0]*.

HPIB_WIDTH
: Return the interface's path width in bits in *arg[0]*.

HPIB_SPEED
: Return the interface's data transfer rate in K-bytes per second in *arg[0]*.

HPIB_READ_PATTERN
: Return the interface's read termination pattern in *arg[0]*; if pattern matching is not enabled, *arg[0]* will hold a **-1**.

HPIB_SIGNAL_MASK
: Return the reason for the last signal. This request returns a mask in *arg[0]*, with bits set indicating the reason(s) for the last SIGEMT sent to the user process. Bit definitions are the same as for the corresponding IO_CONTROL request.

HPIB_LOCK
: Return lock status. If the device is locked to a process, return that process id in *arg[0]*. If the device is not locked, *arg[0]* will hold a **-1**.

HPIB_TERM_REASON
: Return end conditions for the last read from this device/bus. This request returns a byte in *arg[0]*, with a mask of reason(s) for the completion of the last read from the device or raw bus. Applicable bits are:

TR_COUNT	Read requested number of bytes.
TR_MATCH	Detected specified match pattern.
TR_TIMEOUT	Timed out.
TR_END	Device asserted EOI.
TR_ERROR	Detected bus error.
TR_NOTERM	No read done since open.

HPIB_PPOLL
: Conduct a parallel poll. This request returns the bus response to parallel poll in the least significant byte of *arg[0]*. DIO line 0 corresponds to the least significant bit in the response byte. This request applies equally to auto-addressed and raw bus files.

HPIB_SPOLL
: Conduct a serial poll. For raw bus files, this request will conduct a serial poll of the device address in *arg[1]*; the status byte returned by the device will be available in *arg[0]*. Auto-addressed files will ignore any address in *arg[1]*, polling instead the device's predefined address.

HPIB_BUS_STATUS
: Return the status of the HP-IB. This request, applicable to both types of files, returns information related to the current bus state. On return, *arg[0]* holds a value with bits set indicating:

ST_NDAC	NDAC is being asserted.
ST_SRQ	SRQ is being asserted.
ST_REN	interface is in the remote state.
ST_ACTIVE_CTLR	interface is active controller.
ST_SYSTEM_CTLR	interface is system controller.

ST_TALK	interface is addressed to talk.
ST_LISTEN	interface is addressed to listen.

HPIB_WAIT_ON_PPOLL

Wait (sleep) until a given device responds to parallel poll. This request will block the user until either the user's device responds to parallel poll (for auto-addressed files), or until any enabled devices respond (for raw bus files).

For a raw bus file, *arg[1]* and *arg[2]* contain eight-bit masks as defined in the HPIB_SIGNAL_MASK request. The return value of the request, in *arg[0]*, shows which devices responded to parallel poll.

For an auto-addressed file, *arg[1]* specifies the sense of the particular device's assertion. Successful completion of the request implies that the device responded.

HPIB_WAIT_ON_STATUS

Wait (sleep) until any of a set of given states is entered. The event(s) to await are specified by *or*-ing together bits from the list below, and placing the mask in *arg[0]*. Applicable bits are:

ST_SRQ	wait until SRQ is asserted.
ST_ACTIVE_CTLR	wait until user is active controller.
ST_TALK	wait until user is addressed to talk.
ST_LISTEN	wait until user is addressed to listen.

Note that more than one bit can be set, thereby waiting for any of the events to occur. The return value in *arg[0]* will be modified to show the actual event(s) that ended the wait. This is only applicable to raw bus files.

HPIB_INTERFACE_TYPE

Return the type of interface. This will return one of two values in *arg[0]*:

HPIB_INTERFACE	the open file is a HP-IB raw bus file
HPIB_DEVICE	the open file is a HP-IB auto-addressed file

Extended Status Request

If the user wishes to obtain several status variables in one request, the following request can be used:

```
struct io_environment {
  int interface_type;
  int timeout;
  int status;
  int term_reason;
  int read_pattern;
  int signal_mask;
  int width;
  int speed;
  int locking_pid;
} hpib_env;

ioctl (fildes, IO_ENVIRONMENT, &hpib_env);
```

Default Configuration

The default configuration of any HP-IB file is:

Timeout	Infinite
Path Width	8 bits
Transfer Speed	DMA
EOI Assertion	Enabled
Pattern Match	Disabled
Enabled Signals	None
Locking	Unlocked
Bus Address	Raw Bus: hardware switch settings
Termination Reason	TR_NOTERM

RETURNS

A −1 return value for a driver request indicates an error occurred; *errno* is set to specify the reason. In addition to those errors defined in *open*(2), *close*(2), *read*(2), *write*(2), and *ioctl*(2), a driver request can fail if any of the following are true:

[EACCES]	The interface is not active controller or system controller, and this request requires it.
[EBUSY]	Either the interface is currently locked via HPIB_LOCK, or the driver has no software resources available.
[EINTR]	An interface power failure occurred during the processing of this request; the device might have lost state.
[EIO]	Some unclassified error occurred.
[ENXIO]	There is no bus interface associated with the device file.
[ETIMEDOUT]	The transaction did not complete within the timeout specified.

In addition, the following messages can appear on the system console as a result of errors:

instr0 unit %d: device adapter failure. The bus hardware is no longer functioning.

instr0 unit %d: unexpected message (message type = %d, from port %d). The driver received an unclassifiable message.

WARNINGS

It is possible to circumvent the bus protection mechanisms afforded by the auto-addressed and raw bus dichotomy. Specifically, an auto-addressed file user can send commands to any or all devices on the bus with the HPIB_COMMAND request, if the proper device addressing is done within the data buffer.

By default, some HP-IB peripherals respond to parallel poll on DIO line *n*, where *n* is (7 - the device's bus address). That is, a device at address 6 can respond on DIO line 1. Therefore, the results of an HPIB_PPOLL request can be misleading, if some devices have not been remotely configured.

It is impossible to send a secondary address with a data transfer in a single driver request.

HARDWARE DEPENDENCIES

Series 800

The following IO_CONTROL requests are not supported: HPIB_SPEED and HPIB_LOCK.

The following IO_STATUS requests are not supported: HPIB_SPEED and HPIB_LOCK. Locking is done by opening the device file with the O_EXCL flag set.

The HPIB_SRQ request can only affect the RQS bit of the serial poll response byte; all other bits are masked to zero by the hardware.

AUTHOR

Hpib was developed by HP.

FILES

/dev/hpib/*

SEE ALSO

ioctl(2), signal(2), sigvector(2), particular device documentation.

NAME

iomap -- physical address mapping

Remarks:

This information is valid for Series 200 and 300 only.

DESCRIPTION

The iomap mechanism allows the mapping (thus direct access) of physical addresses into the user process address space. For Series 200 and Series 300 Models 310 and 320 computers, the physical address space begins at 0x000000 and extends to 0xffffff.

The special (device) files for iomap devices are character special files with major number 10.

The minor number for iomap devices is of the form:

    0xAAAANN

where AAAA is a two-byte address, and NN is a one-byte field.

The address portion of the minor number is formed by dividing the physical address by 65536. NN*65536 is the size of the region to be mapped. For example, the minor number for a device at 0x720000 and 128k in size is 0x007202.

Access to the iomap devices is controlled by the file permissions set on the character special file.

Multiple processes may concurrently have iomap devices opened and mapped. It is the responsibility of the processes to synchronize their accesses.

Read and *write* system calls are not supported.

Ioctl is used to control the iomap device. The valid ioctl commands (see **<iomap.h>**) are:

IOMAPMAP

map the iomap device into user address space at the location specified in the ioctl argument. If the user address specified in the ioctl argument is 0, the system selects an appropriate address. The ioctl then returns the user address where the device was mapped, storing it in the original ioctl argument (see EXAMPLES below). Multiple processes may concurrently have the iomap device mapped.

IOMAPUNMAP

unmap the iomap device from the user address space.

Close shuts down the file descriptor associated with the iomap device. If the close is for the last system wide open on the device then the iomap device is also unmapped from the user address space; otherwise it is left mapped into the user address space (see IOMAPUNMAP above).

One shared memory descriptor (see *shmget*(2)) is used for each iomap device. Shared memory descriptors are accessible only through the iomap interface. Consequently, attempts to access them through *shmat*(2), *shmctl*(2), *shmdt*(2), etc. result in EACCESS errors.

WARNING

Iomap devices should be created and used with extreme caution. Inappropriate accesses to io devices or ram may result in a system crash.

ERRORS

[ENINVAL]	address field out of range, ioctl command invalid.
[ENOMEM]	cannot allocate required memory for mapping.
[ENODEV]	read/write unsupported.
[ENXIO]	no such address.
[ENOSPC]	cannot allocate required resources for mapping.

[ENOTTY] bad ioctl command, or an ioctl was attempted on an open file.

EXAMPLES

Consider the following code fragment:

```
#include <iomap.h>
...
int fildes;
char* addr;
   addr=REQUESTED_ADDRESS;
   ioctl(fildes,IOMAPMAP,&addr);
   printf("actual address=0x%x/n,addr);
```

where **fildes** is the device special file descriptor and **addr** is a pointer to a character variable.

If **addr** is zero, the system selects a suitable address then returns the selected address, in **addr**.

If the value in **addr** is non-zero, it is used as a specified address for allocating memory. If the specified address cannot be used, an error is returned (see ERRORS).

SEE ALSO

mknod(1M).

NAME

lp – line printer

DESCRIPTION

All file names in **/dev** containing the mnemonic *lp* are special files providing the interface to a particular line printer. A line printer is a character special device which may optionally have an interpretation applied to the data.

If the *lp* mnemonic is preceded by the character **r**, then data is sent to the printer in *raw mode.* (This could assume, for example, a graphic printer operation.) In raw mode, no interpretation is done on the data to be printed, and no page formatting is performed. The bytes are simply sent to the printer and printed as is.

If the *lp* mnemonic is **not** preceded by the character **r**, then the data is interpreted according to rules discussed below. The driver understands the concept of a printer page in that it has a page length (in lines), line length (in characters), and offset from the left margin (in characters). The default line length, indent, lines per page, open and close page eject, and handling of backspace are set to defaults determined when the printer is opened and recognized by the system the first time. If the printer is not recognized, the default line length is 132 characters, indent is 4 characters, lines per page is 66, one page is ejected on close and none on open, and backspace is handled for a character printer.

The following rules describe the interpretation of the data stream:

> A form feed causes a page eject and resets the line counter to zero.
>
> Multiple consecutive form-feeds are treated as a single form-feed.
>
> The new-line character is mapped into a carriage-return/line-feed sequence, and if an offset is specified a number of blanks are inserted after the carriage-return/line-feed sequence.
>
> A new-line that extends over the end of a page is turned into a form-feed.
>
> Tab characters are expanded into the appropriate number of blanks (tab stops are assumed to occur every eight character positions).
>
> Backspaces are interpreted to yield the appropriate overstrike either for a character printer or a line printer.
>
> Lines longer than the line length minus the indent (i.e., 128 characters, using the above defaults) are truncated.
>
> Carriage-return characters cause the line to be overstruck.
>
> When it is opened or closed, a suitable number of page ejects is generated.

Two *ioctl*(2) system calls are available to control the lines per page, characters per line, indent, handling of backspaces, and number of pages to be ejected at open and close times. At either open or close time, if no page eject is requested the paper will not be moved. For opens, line and page counting will start assuming a top-of-form condition.

```
#include <sys/lprio.h>
ioctl (fildes, command, arg)
struct lprio *arg;
```

The *commands* are:

LPRGET	Get the current printer status information and store in the *lprio* structure referenced by **arg**.
LPRSET	Set the current printer status information from the structure referenced by **arg**.

These are remembered across opens, and thus, indent, page width and page length can be set with an external program. If the columns field is set to zero, the defaults are restored at the next open.

The structure passed to the LPRGET and LPRSET *ioctl* calls, as defined in *<sys/lprio.h>*, is:

```
struct lprio {
        short   ind;        /* indent */
        short   col;        /* columns per page */
        short   line;       /* lines per page */
        short   bksp;       /* backspace handling flag */
        short   open_ej;    /* pages to eject on open */
        short   close_ej;   /* pages to eject on close */
    };
```

If the backspace handling flag is 0, a character printer is assumed and backspaces are passed through the driver unchanged. If the flag is a 1, a line printer is assumed, and sufficient print operations are generated to generate the appropriate overstruck characters.

HARDWARE DEPENDENCIES

Integral PC

This version of *lp* is not supported on the Integral PC. Refer to the *Integral Personal Computer Programmer's Guide* for more information about the *lp* implementation on the Intregral PC.

Series 200, Series 300

The uppercase-only flag, the no-overprint flag, the raw-mode flag, and no-page-eject-on-open-or-close flag can be selected (enabled) by appropriate use of the minor number in the *mknod*(1M) command. See the *HP-UX System Administrator's Manual* for details.

See also *slp*(1).

Series 500

The number of characters per line (80 or 132) and wrap-around can be selected/enabled via the *minor* number in the *mknod*(1M) command. See the System Administrator Manual for details.

The LPRGET and LPRSET ioctl commands are not supported and the include file **/usr/include/sys/lprio.h** is not available.

AUTHOR

Lp was developed by HP and AT&T.

FILES

/dev/lp	default or standard printer used by some HP-UX commands;
/dev/[r]lp*	special files for printers

SEE ALSO

lp(1), ioctl(2), intro(7).

NAME

mem, kmem – main memory

DESCRIPTION

Mem is a special file that is an image of the main memory of the computer. It may be used, for example, to examine and patch the system.

Byte addresses in *mem* are interpreted as physical memory addresses. References to non-existent locations cause errors to be returned.

The file *kmem* is the same as *mem* except that kernel virtual memory rather than physical memory is accessed.

WARNINGS

Examining and patching device registers is likely to lead to unexpected results when read-only or write-only bits are present.

FILES

/dev/mem
/dev/kmem

NAME

modem – asynchronous serial modem line control

DESCRIPTION

This section describes the two modes of modem line control and the three types of terminal port access. It also discusses the effect of the bits of the *termio* structure that affect modem line control. The modem related *ioctl*(2) system calls are discussed at the end of the document.

Definitions

There are several terms that are used within the following discussion which will be defined here for reference. "Modem control lines" (CONTROL) are generally defined as those outgoing modem lines that are automatically controlled by the driver. "Modem status lines" (STATUS) are generally defined as those incoming modem lines that are automatically monitored by the driver. CONTROL and STATUS for a terminal file vary according to the modem line control mode of the file (see **Modem line control modes** below). An *open*(2) to a port will be considered to be BLOCKED if it is waiting for another file on the same port to be closed. An *open* to a port will be considered to be PENDING if it is waiting for the STATUS to be raised. An *open* to a port will be considered to be SUCCESSFUL if the *open* system call has returned to the calling process without error.

Open flag bits

Currently, the only *open* flag bit recognized by the driver is the O_NDELAY bit. When this bit is set, an *open* call to the driver will never become blocked. If possible, the *open* will be returned immediately as SUCCESSFUL, and the driver will continue the process of opening the tty file. If it is not possible, then the *open* will be returned immediately with the appropriate error code as described in the appropriate section.

Termio bits

When set, the CLOCAL bit in the *termio* structure (see *termio*(7)) is used to remove the driver's automatic monitoring of the modem lines. However, the user's ability to control the modem lines is determined only by the mode in effect and does not depend on the state of CLOCAL. Normally, the driver will monitor and require the STATUS to be raised. An *open* system call will raise the CONTROL and wait for the STATUS before completing unless the CLOCAL bit is set. (If the O_NDELAY bit is set, the *open* will be returned immediately, but the driver will otherwise continue to monitor the modem lines as normal based on the state of the CLOCAL bit.) Normally, loss of the STATUS will cause the driver to break the modem connection and lower the CONTROL. However, if CLOCAL is set, any changes in the STATUS will be ignored. A connection is required before any data may be read or written, unless CLOCAL is set. Any timers that would normally be in effect (see **Modem line control modes** and **Modem timers** below) will be stopped while CLOCAL is set.

When the CLOCAL bit is changed from clear to set, the driver will assume the existence of an active device (such as a modem) on the port regardless of the STATUS. If any of the CONTROL are raised at that point in time, they will continue in that state. The STATUS will no longer be actively monitored. When the CLOCAL bit is changed from set to clear, the driver will resume actively monitoring the STATUS. If all of the CONTROL and STATUS are raised at that point in time, the driver will continue the modem connection. If any of the STATUS are not raised, the driver will act as though those signals were lost (as described in **Modem line control modes** below) and, if the device is a controlling terminal, a *hangup* signal will be sent to the process group. If any of the CONTROL are not raised, the driver will break the modem connection by lowering all the CONTROL.

The HUPCL bit in the *termio* structure determines the action of the driver regarding the CONTROL when the last *close* system call is issued to a terminal file. If the HUPCL bit is set, the driver will lower the CONTROL at *close* time and the modem connection will be broken. If

HUPCL is not set and a modem connection exists, it will continue to exist, even after the *close* is issued. The driver will not change the CONTROL.

Terminal port access types

There are three types of modem access: call-in connections, call-out connections, and direct (no modem control) connections. A given port may be accessed through all three types of connection by accessing different files. The modem access type of a terminal file is determined by the file's major and/or minor device numbers.

The call-in type of access is used when the connection is expected to be established by an incoming call. This is the type that would be used by *getty*(1M) to accept logins over a modem. When an *open* is issued to such a file, the driver may wait for an incoming call and will then raise the CONTROL based on the current mode (see below) of the port. When the port is closed, the driver may or may not lower the CONTROL depending on the HUPCL bit.

The call-out type of access is used when the connection is expected to be established by an outgoing call. This would be used by programs such as *uucp*(1). When an *open* is issued to such a file, the driver will immediately raise the CONTROL and wait for a connection based on the mode currently in effect. When the port is closed, the driver may or may not lower the CONTROL depending on the HUPCL bit.

The direct type of access is used when no driver modem control is desired. This could then be used for directly connected terminals that use a three-wire connection, or to talk to a modem before a connection has been established. The second case allows a program to give dialing instructions to the modem. Neither the CLOCAL nor the HUPCL bits have any effect on a port accessed through a direct file. (However, both bits may be inherited by other types of files; see **Terminal port access interlock** below.) An *open* to a direct file does not affect the CONTROL and does not depend on any particular state of the STATUS to succeed. When the file is closed, the driver will not affect the state of the CONTROL. If a modem connection has been established, it will continue to exist. Setting the speed of a direct file to B0 (see *termio*(7)) will be considered an impossible speed change and will be ignored. It will not affect the CONTROL.

Modem line control modes

There are two modes of modem line control: CCITT mode and simple mode. A given port may have only one of these two modes in effect at any given point in time. An attempt to *open* a port with a mode other than the one in effect (from a PENDING or SUCCESSFUL *open* on a different file) will cause the *open* to be returned with an ENXIO error. The modem access type of a terminal file is determined by the file's major and/or minor device numbers.

CCITT mode is used for connections to switched line modems. The CONTROL for CCITT mode are Data Terminal Ready (DTR) and Request to Send (RTS). The STATUS are Data Set Ready (DSR), Data Carrier Detect (DCD), and Clear to Send (CTS). Additionally, the Ring Indicate (RI) signal indicates the presence of an incoming call. When a connection is begun (an incoming call for a call-in file or an *open* issued to a call-out file), the CONTROL are raised and a connection timer (see **Modem timers** below) is started. If the STATUS become raised before the time period has elapsed, a connection is established and the *open* request is returned successfully. If the time period expires, the CONTROL are lowered and the connection is aborted. For a call-in file, the driver will wait for another incoming call; for a call-out file, the *open* will be returned with an EIO error. Once a connection is established, loss of either DSR or CTS will cause the CONTROL to be lowered and a *hangup* signal to be generated if the device is a controlling terminal.

If DCD is lost, a timer is started. If DCD resumes before the time period has expired, the connection will be maintained. However, no data transfer will occur during this time. The driver will stop transmitting characters, and any characters received by the driver will be discarded. (However, on some implementations data transmission cannot be stopped. See **HARDWARE DEPENDENCIES**.) If DCD is not restored within the allotted time, the connection will be broken as described above for DSR and CTS.

If the modem connection is to be broken when the *close* system call is issued (i.e. HUPCL is set), then the CONTROL will be lowered and the *close* will be returned as successful. However, no further *opens* will be allowed until after both DSR and CTS have been lowered by the modem, and the hangup timer (see **Modem timers** below) has expired. The action taken in response to an *open* during this time will be the same as if the port were still open. (See **Terminal port access interlock** below.)

When a port is in CCITT mode, the driver has complete control of the modem lines and the user is not allowed to change the setting of the CONTROL or affect which STATUS are actively monitored by the driver (see **Modem ioctls** below). This is to provide strict adherence with the CCITT recommendations.

Simple mode is used for connections to devices which require only a simple method of modem line control. This can include devices such as black boxes, data switches, or for system-to-system connections. It can also be used with modems which cannot operate under the CCITT recommendations. The CONTROL for simple mode consists of only DTR. The STATUS consists of only DCD. When an *open* is issued, the CONTROL is raised but no connection timer is started. When the STATUS becomes raised, a connection is established and the *open* request is returned as SUCCESSFUL. Once a connection is established, loss of the STATUS will cause the CONTROL to be lowered and a *hangup* signal to be generated if the device is a controlling terminal.

When a port is in simple mode, the driver will normally control the modem lines. However, the user is allowed to change the setting of the CONTROL (see **Modem ioctls** below).

Terminal port access interlock

An interlock mechanism is provided between the three access types of terminal files. It prevents more than one file from being successfully opened at a time, but allows certain *opens* to succeed while others are PENDING so that a port can be opened through a call-out connection while *getty* has a pending *open* at a call-in connection. The three access types are given a priority that determines which *open* will succeed if more than one file has an *open* issued against it. The three access types are ordered from lowest priority to highest as follows: call-in, call-out, and direct.

If an *open* is issued to a port which already has a SUCCESSFUL *open* on it of a lower priority type, the new *open* will be returned with an EBUSY error. (EBUSY will also be returned by an attempted *open* on a CCITT call-out file if an incoming call indication is currently being received. In this case, if there is a PENDING *open* on the corresponding CCITT call-in file, this PENDING *open* will complete.) If the lower priority *open* is PENDING, the new *open* will succeed if possible, or will be left PENDING if waiting for the STATUS and the lower priority *open* will become BLOCKED. If a higher priority *open* has succeeded or is PENDING, the new *open* will be BLOCKED, unless the new *open* has the O_NDELAY flag bit set, in which case the *open* will be returned with an EBUSY error. Once an *open* on one type of file is SUCCESSFUL, any PENDING *opens* on lower priority files will become BLOCKED.

When a file of one priority is closed, a BLOCKED *open* on the next lower priority type file will become active. If all of the STATUS are raised, the *open* will be SUCCESSFUL, otherwise the *open* will become PENDING waiting for the STATUS. If the lower priority *open* is SUCCESSFUL (because the connection was maintained when the higher priority file was closed), the port characteristics (speed, parity, etc.) that were set by the higher priority file will be inherited by the lower priority file. If the connection is not maintained through the *close*, the port characteristics will be set to default values.

Modem timers

There are four timers currently defined for use with modem connections. The first three of the timers are applicable only to CCITT mode connections. In general, the effect of changing a timer value while the timer is running is system dependent. However, setting the timer value to zero is guaranteed to disable the timer even if it is running.

The connect timer is used to limit the amount of time to wait for a connection to be established once it has been begun. This timer is started when an incoming call has been received on a call-in file, or when an *open* has been issued on a call-out file for which no *opens* are already pending. If the connection is completed in time, the timer is aborted. If the time period expires, the connection is aborted. For a call-in file, the driver will again wait for an incoming call and the *open* will remain pending. For a call-out file, the *open* will be returned with an EIO error.

The carrier detect timer is used to limit the amount of time to wait before causing a disconnect if DCD drops. If carrier is not re-established in this time, a disconnect will occur. If carrier is re-established before the timeout, the timer will be aborted and the connection maintained. During the period when carrier is not raised, no data will be transferred across the line.

The no activity timer is used to limit the amount of time a connection will remain open with no data transfer across the line. When the data line becomes quiescent with no data transfer, this timer will be started. If data is again transferred over the line in either direction before the time limit, the timer will be aborted. If no activity occurs before the timeout has occurred, the driver will disconnect the line. This can be used to avoid long and costly telephone connections when data transfer has been stopped either normally or abnormally.

The last timer defined, the hangup timer, is used for both CCITT and simple modes. This timer controls the amount of time to wait after disconnecting a modem line before allowing another *open* to be allowed. This time period should be made long enough to guarantee that the connection has been terminated by the telephone switching equipment. If this period is not long enough, the telephone connection may not be broken and a succeeding *open* may complete with the old connection.

Modem ioctls

Several *ioctl* system calls apply to manipulation of modem lines. They use the following information defined in **<sys/modem.h>**.

```
#define NMTIMER    6
typedef unsigned long  mflag;
struct  mtimer   {
        unsigned short m_timers[NMTIMER];
        };
```

Each bit of the *mflag* long corresponds to one of the modem lines as follows:

MRTS	Request to Send	outbound
MCTS	Clear to Send	inbound
MDSR	Data Set Ready	inbound
MDCD	Data Carrier Detect	inbound
MDTR	Data Terminal Ready	outbound
MRI	Ring Indicator	inbound
MDRS	Data Rate Select	outbound

The timer values are defined in the array m_timers. The relative position of the timer and default initial values and units for each timer are as follows:

0	MTCONNECT	25 s
1	MTCARRIER	400 ms
2	MTNOACTIVITY	0 min

3	MTHANGUP	250 ms
4	Reserved	
5	Reserved	

A value of zero for any timer will disable that timer.

The modem line *ioctl* system calls have the form:

ioctl (fildes, command, arg)
mflag *arg;

The commands using this form are:

MCGETA Get the current state of both inbound and outbound modem lines and store in the *mflag* long referenced by **arg**. A raised line will be indicated by a one bit in the appropriate position.

MCSETA Set the outbound modem lines from the *mflag* long referenced by **arg**. Setting an outbound bit to one causes that line to be raised and zero to be lowered. Setting bits for inbound lines has no effect. Setting any bits while in CCITT mode has no effect. The change to the modem lines is immediate and using this form while characters are still being output may cause unpredictable results.

MCSETAW Wait for the output to drain and set the new parameters as described above.

MCSETAF Wait for the output to drain, then flush the input queue and set the new parameters as described above.

The timer value *ioctl* system calls have the form:

ioctl (fildes, command, arg)
struct mtimer *arg;

The commands using this form are:

MCGETT Get the current timer value settings and store in the *mtimer* structure referenced by **arg**.

MCSETT Set the timer values from the structure referenced by **arg**.

For any timer, setting the timer value to its previous value has no effect.

WARNING

Occasionally it is possible that a process may open a call-out file at approximately the same time as an incoming call is received. In some cases, the call-out connection may be satisfied by the incoming call. In general, however, the results are indeterminate. If necessary, the situation can be avoided by the use of two modems and ports, one for call-out connections and the other for receiving incoming calls.

HARDWARE DEPENDENCIES

Some hardware implementations may not have access to all modem lines supported by MCSETA. If a particular hardware does not support a given line, attempts to set the value of a line will be ignored, and reading the current state of the line will return zero. The I/O card manual should be referenced to determine the lines supported by the hardware installed.

Some hardware implementations may not have access to all timers supported by MCSETT. Also, the granularity of the individual timers may vary depending on the hardware and system in use. The effect of setting a timer out of range or with a granularity outside the capability of a particular system should be documented by that system. The effect of changing the value for a timer while that timer is running is system dependent and should be documented by each system.

Setting the CLOCAL bit while a timer is running will cause the timer to be stopped. It is a system dependency whether or not the timer is restarted, and if so, the value at which it is restarted when the CLOCAL bit is subsequently cleared.

On those implementations supporting the HP27140A 6-Channel Multiplexer, transmission of characters cannot be stopped during loss of DCD. The driver cannot detect loss of DCD until the connection is broken. Also, the I/O card may still have characters in its internal buffers and will still try to transmit them.

Series 500

For the HP27140A 6-port modem multiplexor, the ranges and resolutions of the timers are as follows:

MTCONNECT	0-255 sec, 1 sec resolution
MTCARRIER	0-2550 msec, 10 msec resolution
MTNOACTIVITY	0-1092 min, 1 min resolution
MTHANGUP	0-65535 msec, 10 msec resolution

If a timer is set out of its range, the maximum value that timer can assume is used instead.

For the HP27128A Asynchronous Serial Interface, the ioctl requests described above are not supported. The timers have fixed values as follows:

MTCONNECT	25 sec
MTCARRIER	400 msec
MTNOACTIVITY	0 min
MTHANGUP	500 msec

This interface only supports the call-in and call-out port access types, and does not support the direct access type.

It is not possible to change the state of the CLOCAL bit when using CCITT mode.

Simultaneous call-in and call-out open attempts in CCITT mode are not allowed.

The default state of the CLOCAL bit upon first open is determined by the state of switch on the interface (see your system administration manual).

FILES

/dev/cua*
/dev/cul*
/dev/tty*
/dev/ttyd*

AUTHOR

Modem was developed by HP and AT&T.

SEE ALSO

mknod(1M), stty(1), ioctl(2), termio(7).

NAME

mt – magnetic tape interface and controls

DESCRIPTION

The files **/dev/mt/*** and **/dev/rmt/*** refer to specific tape drives; the behavior of the specific unit is specified in the major and minor numbers of the device special file which describes the tape unit.

The following naming convention is recommended for tape devices, and serves to connect most of the mode flags with the device name:

/dev/{r}mt/(c#d)#[hml]{n}

where **r** indicates a raw device, **c#d** indicates the controller number (which is optionally specified by the system administrator), # is the device number, **hml** indicates the density (**h** (high) for 6250 bpi, **m** (medium) for 1600 bpi, and **l** (low density) for 800 bpi), and **n** indicates no rewind on close. For example, **/dev/rmt/2mn** is raw device 2 at 1600 bpi with no rewind. Blocked mag tapes are used only for special situations and are supported only in some implementations. See HARDWARE DEPENDENCIES below for details. The selection of controller and unit numbers is system dependent, and is discussed in the appropriate System Administrator's Guide.

The operation of a tape drive is controlled by mode flags, which are usually encoded as bits in the *minor* number of the device special file.

no-rewind	Unless this mode is requested, the tape is automatically rewound upon close. When a rewind on close is not desired, the **n** flag should be used in the device name.
style	When this mode is requested, the tape drive behaves as on Berkeley systems; when not requested, the drive behaves as on AT&T UNIX operating systems. The details are described below. The *ioctl* operations described below work in both modes on *raw* tapes only. The *mt*(1) tape movement utility requires that the Berkeley mode be specified.
density	This may be used to select the density of the tape being written. Possible values that may be selected may include 6250, 1600, and 800 bpi, depending on the capabilities of the specific tape drive. This corresponds to the **h, m** and **l** flags in the recommended device name.

Refer to the System Administrator Manual for your computer for more specific details of how to select the modes for a given device.

The special files associated with a *raw* tape interface are named **rmt/***. Unless otherwise stated, the following discussion refers to raw magnetic tapes.

When opened for reading or writing, the tape is assumed to be positioned as desired.

When a file opened for writing is closed, two consecutive EOF marks are written if, and only if, one or more writes to the file have occurred. The tape is rewound unless the no-rewind mode has been specified, in which case the tape is positioned before the second EOF just written.

When a file open only for reading is closed, and the no-rewind bit is not set, the tape is rewound. If the no-rewind bit is set, the behavior depends on the *style* mode. For AT&T-style devices, the tape is positioned after the EOF following the data just read. For Berkeley-style devices, the tape is not re-positioned in any way.

Each *read* or *write* call reads or writes the next record on the tape. In the write case the record has the same length as the buffer given (within the limits of the hardware).

During a read, the record size is passed back as the number of bytes read, up to the buffer size specified. The number of bytes ignored is available in the *mt_resid* field of the *mtget* structure via the MTIOCGET call of *ioctl.* The buffer and size may have implementation dependent alignment restrictions.

Reading an EOF mark is returned as a zero-length read, i.e., the data count returned will be zero, and the tape is positioned after the EOF, so that the next read will return the next record.

Seeks on a raw magnetic tape device are ignored. Instead, the *ioctl* operations below can be used to position the tape and determine its status.

The following is included from <*sys/mtio.h*> and describes the possible operations:

```
/* mag tape I/O control requests */

#define     MTIOCTOP   _IOW(m,1,struct mtop)  /* do mag tape op */
#define     MTIOCGET   _IOR(m,2,struct mtget) /* get tape status */

/* structure for MTIOCTOP - mag tape op request */

struct      mtop {
        short      mt_op;      /* operations defined below */
        daddr_t    mt_count;   /* how many of them */
};

/* operations */

#define MTWEOF    0   /* write end-of-file record */
#define MTFSF     1   /* forward space file */
#define MTBSF     2   /* backward space file */
#define MTFSR     3   /* forward space record */
#define MTBSR     4   /* backward space record */
#define MTREW     5   /* rewind */
#define MTOFFL    6   /* rewind, put drive offline */
#define MTNOP     7   /* no-op, may set status */

/* structure for MTIOCGET - mag tape get status command */

struct            mtget {
          long    mt_type;
          long    mt_resid;

/* The following two registers are device dependent */

        long    mt_dsreg1;
        long    mt_dsreg2;

/* The following is a device independent status word */

         long    mt_gstat;
         long    mt_erreg;

/* The following are used only when block devices are supported */

        daddr_t   mt_fileno;
        daddr_t   mt_blkno;
};
```

```
/*
* Constants for mt_type; the first three are historical
*/
#define MT_ISTS         01
#define MT_ISHT         02
#define MT_ISTM         03
#define MT_IS7970E      04
#define MT_ISSTREAM     05
```

HARDWARE DEPENDENCIES

Series 200, 300, and 500:

Block magnetic tape is not supported.

Series 800:

The MTNOP operation does not set the device independent status word.

The files **/dev/mt/*** refer to block magnetic tapes. They should only be used for system installation or for treating a pre-written mag tape as a read-only block file system. A read-only block tape can best be created with *dd*(1) using raw mode and a record size of 512 bytes.

Although the size of records on a block tape is always 512 bytes, the block I/O system deals with block sizes that are a multiple of DEV_BSIZE (param.h) with the tape driver making the translation. For this reason, if a user is attempting to write 512 bytes in block mode, the block I/O system will attempt to pre-read from the block tape prior to merging in the new data and writing it to the tape. Since a pre-read of a blank tape or a tape of unknown format will terminate with an error, it is strongly suggested that *dd*(1) be used to create the tape as described above, and that block magnetic tape be used only for seeking and reading. Alternatively, always writing a byte count which is an even multiple of BLKDEV_IOSIZE (param.h) bytes will avoid the hazards of a block pre-read.

A tape treated as a block-special device consists of several 512-byte records terminated by an EOF.

The system makes it possible to treat a pre-written block tape like an ordinary file, with the exception that writing in the "middle" of a file truncates the file at that point. Seeks have their usual meaning and it is possible to read or write a byte at a time.

The efficient use of streaming tape drives with large internal buffers and *immediate-reporting* require the following end-of-tape procedures:

All writes near the EOT foil (which is not on the recording surface) will complete without error if actually written to the tape. When the tape drive determines the foil has been passed, subsequent writes will not occur and an error will be returned.

Since some applications require the writing of a trailer for multiple tape operations, a user request for mag tape status, which will reflect the EOT condition, signals the driver to drop all write barriers. Caution must be exercised in order to keep the tape on the reel.

When reading near the end-of-tape, the user will not be made aware of the EOT foil marker. Instead, the typical double EOF marks or a pre-arranged trailer will signal the logical end-of-tape.

The EOT description above applies in the default case when *immediate-reporting* mode is allowed in the *minor*. When it is specifically dis-allowed in the minor, the EOT operation attempts to emulate *compatibility-mode* on other HP-UX machines. In this mode, the write that encounters the EOT foil returns an error with the tape automatically backing up over that record. The read that encounters the EOT foil returns an error.

Since there are differences in EOT sensing among various types of mag tape drives because of the physical placement of sensors, any application (such as multiple tape *cpio*(1) backups) requiring that data be continued from the EOT area of one tape to another tape must be restricted. Therefore, the tape drive type and mode should be the same for the creation and reading of the tapes.

The following macros are defined in <sys/mtio.h> for decoding the generic status of the tape drive (returned in the mt_gstat field):

```
GMT_EOF(x)            /* At an EOF mark */
GMT_BOT(x)            /* At beginning of tape */
GMT_EOT(x)            /* At end of tape */
GMT_WR_PROT(x)        /* Tape is write protected */
GMT_ONLINE(x)         /* Drive is online */
GMT_D_6250(x)         /* Density is 6520 bpi */
GMT_D_1600(x)         /* Density is 1600 bpi */
GMT_D_800(x)          /* Density is 800 bpi */
GMT_DR_OPEN(x)        /* Drive door is open */
GMT_IM_REP_EN(x)     /* Immediate reporting on */
(which means that the drive will report completion immediately)
```

FILES

/dev/mt/*
/dev/rmt/*

AUTHOR

Mt was developed by HP, and the University of California, Berkeley.

BUGS

It is impossible to write a program that will leave a tape positioned at the beginning on an AT&T-style raw device with the no-rewind bit set, because closing the device file upon the program's termination will reposition the tape after the first EOF mark.

SEE ALSO

ct(7), dd(1), mt(1).

NAME

null – null file

DESCRIPTION

Data written on a null special file is discarded.

Reads from a null special file always return 0 bytes.

FILES

/dev/null

NAME

pty – pseudo terminal driver

SYNOPSIS

pseudo-device pty

DESCRIPTION

The *pty* driver provides support for a device-pair termed a *pseudo terminal.* A pseudo terminal is a pair of character devices, a *master* device and a *slave* device. The slave device provides processes an interface identical to that described in *termio*(7). However, whereas all other devices which provide the interface described in *termio*(7) have a hardware device of some sort behind them, the slave device has, instead, another process manipulating it through the master half of the pseudo terminal. That is, anything written on the master device is given to the slave device as input and anything written on the slave device is presented as input on the master device.

HP-UX termio(7)		Slave : pty : Master		Server
Application	<-->	Side : : Side	<-->	Process
Processes				

The following *ioctl* requests, defined in **<sys/ptyio.h>**, apply only to the master side of pty:

TIOCBREAK Causes a break operation to be done on the slave side of the pty. This action is the same as if a user had hit the break key on a real terminal. Takes no parameter.

TIOCSIGSEND Causes a signal to be sent on the slave side of the pty to the current tty process group of the slave side. The value of the parameter is taken to be the signal number to be sent. An EINVAL error will be returned and no signal sent if the specified signal number does not refer to a legal signal (see *signal*(2)). Note that this request allows the server process to send signals to processes that are not owned by the same user id.

TIOCSTOP Stops data flowing from the slave side of the pty to the master side (e.g. like typing ^S). Takes no parameter.

TIOCSTART Restarts output (stopped by TIOCSTOP or by typing ^S). Takes no parameter.

TIOCPKT Enable/disable *packet* mode. Packet mode is enabled by specifying (by reference) a nonzero int parameter and disabled by specifying (by reference) a zero int parameter. When applied to the master side of a pseudo terminal, each subsequent *read* from the master side will return data written on the slave part of the pseudo terminal preceded by a zero byte (symbolically defined as TIOCPKT_DATA), or a single byte reflecting control status information. In the latter case, the byte is an inclusive-or of zero or more of the bits:

TIOCPKT_FLUSHREAD
Whenever the read queue for the slave side is flushed.

TIOCPKT_FLUSHWRITE
Whenever the write queue for the slave side is flushed.

TIOCPKT_STOP
Whenever data flowing from the slave side of the pty to the master side is stopped by means of ^S, TIOCSTOP, or TCXONC.

TIOCPKT_START
Whenever data flowing from the slave side of the pty to the master side is restarted.

TIOCPKT_DOSTOP
: Whenever the stop and start characters get set to ^S/^Q.

TIOCPKT_NOSTOP
: Whenever the stop and start characters get set to something other than ^S/^Q.

TIOCREMOTE
A mode for the master half of a pseudo terminal, independent of TIOCPKT. This mode causes input to the pseudo terminal to be flow controlled and not input edited (regardless of the terminal mode). Each write to the master side produces a record boundary for the process reading the slave side. In normal usage, a write of data is like the data typed as a line on the terminal; a write of 0 bytes is like typing an end-of-file character (the EOF character as defined in *termio*(7)). The data read by the slave side is identical to the data written on the master side. Data written on the slave side and read on the master side with TIOCREMOTE enabled is still subject to the normal *termio*(7) processing. TIOCREMOTE can be used when doing remote line editing in a window manager, or whenever flow controlled input is required. The request takes one int sized parameter, passed by value. When zero, it disables TIOCREMOTE; when one it enables TIOCREMOTE. TIOCREMOTE is only effective when TIOCTTY (explained below) is also enabled, and all data buffered in the pseudo terminal will be flushed when this request is made.

TIOCTTY
Enable or disable all *termio*(7) processing by pty. When disabled, all data is passed through the pty with no modification. *Termio*(7) processing (of input and output such as tab expansion) is enabled by specifying (by reference) a nonzero int parameter and disabled by specifying (by reference) a zero int parameter. Default is to be enabled. When TIOCTTY is disabled, the following pty modes are also inoperable: TIOCBREAK, TIOCSTOP, TIOCSTART, TIOCPKT, TIOCREMOTE, and TIOCMONITOR. Issuing a TIOCTTY ioctl request will also flush all data buffered in the pseudo terminal, and release any processes currently blocked waiting for data.

When TIOCTTY is enabled (the default case), all *termio*(7) ioctl requests are handled by the pty driver itself. When TIOCTTY is disabled, slave side *termio*(7) ioctl requests are either ignored completely or passed to the master side depending upon the state of TIOCTRAP below. Slave side non-termio ioctl requests are not affected by the state of TIOCTTY. They are always ignored completely or passed to the master side depending upon the state of TIOCTRAP below.

Data being written through a pseudo terminal with TIOCTTY disabled will be handled in a manner similar to the way data flows through a pipe. A write request will block in the pty until all of its data has been written into the pty. A read request will block if there is no data available unless the O_NDELAY flag is set (see *fcntl*(2)). When data is available to be read, the read request will return whatever is available, and will not wait for the number of bytes requested to be satisfied. The number of bytes a pty can contain in its internal memory is implementation dependent, but will always be at least 256 bytes in each direction. For example, a write on the slave side of a pty of 1024 bytes might be read on the master side by four read requests returning 256 bytes each. The size of the chunks of data that are read is not guaranteed to be consistent, but no data will be lost.

Opening and closing of the master side acts as a modem connection/disconnection on a real terminal as far as the slave side is concerned. Having no server on the master side will cause opens on the slave side to hang until there is a server. (*Termio*(7) description of O_NDELAY interaction with pty is also supported.) Opens to the master side are exclusive. Attempts to open an already open master side of a pty will return *errno*(2) error EBUSY. (Attempts to open a non-existent pty will return *errno*(2) ENXIO.) Closing the master side of a pty sends a SIGHUP hangup signal to the tty process group number of the corresponding slave side and flushes pending input and output.

Any *termio*(7) ioctl request can also be applied to the master side of the pty, unless TIOCTTY has been disabled.

IOCTL/OPEN/CLOSE TRAPPING

The capabilities that follow give additional flexibility and control for servers connected to the master side.

When trapping of ioctl/open/close is enabled, *ioctl*(2), *open*(2), and *close*(2) requests made to the slave side will notify the server on the master side of each request. The close request will only notify the server and continue to completion, while the open and ioctl requests will not complete until the master side has had a chance to handle them. The master side acknowledges completion via an ioctl to the master side. If the pty is not enabled to pass *ioctl*(2), *open*(2), and *close*(2) from the slave to the master, then they will be ignored (except for *termio*(7) related processing).

The following ioctl calls apply only to the master side of a pty and pertain to trapping open, close, and ioctl. They are also defined in **<sys/ptyio.h>**:

TIOCTRAP
: Enable or disable trapping of ioctl, open, and close from the slave side. Trapping is enabled by specifying (by reference) a nonzero int parameter and disabled by specifying (by reference) a zero int parameter. Default is to be disabled. (*termio*(7) ioctl requests will not be trapped, unless TIOCTTY is also disabled or TIOCMONITOR is enabled.)

TIOCTRAPSTATUS
: Find out if any ioctl/open/close traps are pending. The argument points to an int, that will be set to one if anything is pending and zero if nothing is pending. This ioctl request is used when the preferred method of a *select*(2) "exceptional condition" is not available.

TIOCREQGET
: In response to a *select*(2) "exceptional condition" on the master side, this ioctl request will read the pending ioctl, open, or close information into memory pointed to by the argument in the form:

```
struct request_info {
        int request;
        int argget;
        int argset;
        short pgrp;
        short pid;
        int errno_error;
        int return_value;
};
```

All elements of request_info refer to the slave side of the pty. Enumerating the elements:

request	is the ioctl command received.
argget	is the ioctl request to apply to master side to receive the trapped ioctl structure if there is one to receive, (a zero value means there is none). (When nonzero, argget is a TIOCARGGET request with the size field precomputed.)
argset	is the ioctl request to apply to master side to send back the resulting ioctl structure if there is one to send back, (a zero value means there is none). (When nonzero, argset is a TIOCARGSET request with the size field precomputed.)

pgrp	is the process group number of the process doing the operation.
pid	is the process id of the process doing the operation.
errno_error	is the *errno*(2) error code (initialized to zero) to be returned by ioctl on the slave side.
return_value	(initialized to zero) is the success value to be returned by ioctl on the slave side when ***errno_error*** is not set.

For the case that the ioctl argument received on the slave side is not a pointer, its value is stored as four bytes that can be retrieved with an ioctl request to the master side equal to *argget*.

When an open or close is being passed, ***request*** will be set to TIOCOPEN or TIOCCLOSE, respectively. For TIOCOPEN and TIOCCLOSE, both *argget* and *argset* will be of zero because there is no ioctl structure. When TIOCTTY is enabled, the *termio*(7) definition of open/close will be executed first, before being passed to the master side. Note, while all opens are trapped, only the last close on a particular inode for a pty slave side is trapped by the pty.

If a TIOCREQGET is done before anything has been trapped, this master side ioctl will block until a slave side ioctl, open, or close is trapped.

TIOCREQSET Done to complete the handshake started by a previous TIOCREQGET. The argument should point to the request_info structure as defined by the TIOCREQGET.

Before doing this ioctl, to complete the handshake, the server should set *errno_error* to an *errno*(2) error value to be passed back to the slave side. If there is no error, *errno_error* can be left alone because the pty will have initialized it to zero. Also, when there is no error, ***return_value*** should be set, if other than a zero result is desired. It should be noted that the ability to determine the return value and error code for a request to the slave side is only available for trapped ioctl requests. The server will not be able to set these values if the trapped request is an open or a close.

If the TIOCREQSET request is made and the request value in the passed request_info structure does not equal the trapped value, *error*(2) EINVAL will be returned. (EINVAL is also returned if there is no trapped ioctl/open/close.)

If the trapped slave-side request has been interrupted by a signal between the time that the server has done the TIOCREQGET and the TIOCREQSET, an EINVAL error will be returned by the TIOCREQSET request.

TIOCMONITOR Enable or disable read only trapping of *termio* ioctl requests when TIOCTTY is also enabled. (When TIOCTTY is disabled, TIOCMONITOR has no effect. Also TIOCMONITOR is independent of TIOCTRAP.) Trapping is enabled by specifying (by reference) a nonzero int parameter and disabled by specifying (by reference) a zero int parameter. Default is to be disabled.

This allows a server process attached to the master side of the pty to know when characteristics of the line discipline in the pty are changed by an application on the slave side. The mechanism for handshaking trapped *termio*(7) requests (when TIOCTTY is enabled) is the same as that for non-termio ioctl requests; except that any changes or error conditions set by the server on the master side will have no effect. (It is recommended that *termio*(7) ioctl requests be used on the master side to interrogate the configured state of the line discipline in the pty. One reason for this is to compensate for the window of time before TIOCMONITOR is enabled, when *termio*(7) ioctls were not trapped.)

When using *select*(2) on the master side of a pty, the "exceptional condition" refers to an open, close, or ioctl pending on the slave side. Ready for reading or writing refers to a read, or write pending respectively, from the point of view of the master side.

Of the ioctls that are subject to being trapped, only one per pty may be handled at one time. This means that when an application does a non-termio ioctl to the slave side, all other ioctls to the same pty slave side will be blocked until the first one is handshaked back by the master side. (Ioctls that are not trapped, such as *termio*(7) when TIOCTTY is enabled and TIOCMONITOR is disabled, will not be blocked.) This permits the implementation of indivisible operations by an ioctl call on the slave side that is passed to the server process.

In summary, handshaking of an ioctl/open/close on the master side is done using the following steps:

Slave Side open/close/ioctl Trapped.
: This is indicated via a *select*(2) exceptional condition or via the TIOC-TRAPSTATUS ioctl request.

TIOCREQGET ioctl request.
: This is done to find out what slave open/close/ioctl is trapped.

argget ioctl request.
: This optional ioctl is done if argget is nonzero and the server wants to do more than just reject the trapped slave ioctl.

argset ioctl request.
: This optional ioctl is done if argset is nonzero and the server wants to pass back a modified ioctl structure. It is done after the trapped ioctl is processed via the server on the master side.

TIOCREQSET ioctl request.
: This is done to complete the trapped slave open/close/ioctl. In case the trapped request is an ioctl, errno_error should be set appropriately. return_value should be set for trapped slave ioctls if errno_error is set to zero.

While a process is waiting in the slave side of the pty for the server to complete a handshake, it is susceptible to receiving signals. The following master side ioctl allows the server process to control how the pty will respond when a signal attempts to interrupt a trapped open or ioctl request.

TIOCSIGMODE Sets the signal handling state of the pty to the mode specified as the argument. The mode can have three values, which are TIOCSIGBLOCK, TIOCSIGA-BORT, and TIOCSIGNORMAL.

TIOCSIGBLOCK Causes some signals that are destined for the process whose open/ioctl is trapped to be postponed. The signals that are blocked are those which would otherwise cause the process to jump to an installed signal handler. Signals that are currently being ignored or would cause the slave-side process to be aborted will not be held off. When the server process completes the handshake by means of the TIOCREQSET ioctl request, the slave-side process will return to the calling program, and any pending signals will then be acted upon. Any signals that the user has blocked by means of *sigblock*(2) will continue to be blocked.

TIOCSIGABORT Forces all signals that interrupt a trapped open/ioctl request to not be restartable. The server process will set this mode when it wants the interrupted requests to return to the calling program with an EINTR error.

TIOCSIGNORMAL
: This is the default mode of the pty. If a signal interrupts a trapped open/ioctl request, the user's signal handler routine has the option of specifying whether the request is to be restarted. If the request is to be restarted, it will be executed again from the beginning, and the server will have to do another TIOCREQGET to start the handshake over again. If the user's signal handler routine specifies that the interrupted request is not to be restarted, then the request will return to the calling program with EINTR upon completion of the signal handler. Note that it is not guaranteed that the restarted request will be the very next one to be trapped.

WARNINGS

It is not possible for the slave side to indicate an End Of File condition to the master side.

When using TIOCREMOTE, a single write to the master side of greater than 256 bytes may result in multiple smaller records being read from the slave side, instead of only one record.

HARDWARE DEPENDENCIES

Series 200, 300 and 500
: The largest ioctl argument passable between master and slave sides is currently limited to 128 bytes.

Series 500
: The TIOCREMOTE mode is not currently implemented.

AUTHOR

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

FILES

/dev/pty[pqr]*	master pseudo terminals
/dev/tty[pqr]*	slave pseudo terminals
/dev/ptym/pty[pqr]*	master pseudo terminals
/dev/pty/tty[pqr]*	slave pseudo terminals

SEE ALSO

ioctl(2), select(2), signal(2), termio(7).

NAME

stty – terminal interface for Version 6/PWB compatibility

REMARKS

These facilities are included to aid in conversion of old programs, and should not be used in new code. Use the interface described in *termio*(7). Note that these conversions do **not** work for programs ported from UNIX Time-Sharing System, Seventh Edition (Version 7), since some V7 flags are defined differently.

DESCRIPTION

These routines attempt to map the UNIX Time-Sharing System, Sixth Edition (Version 6), and PWB *stty* and *gtty* calls into the current ioctls that perform the same functions. The mapping cannot be perfect. The way the features are translated is described below. The reader should be familiar with *termio*(7) before studying this page.

The following data structure is defined in the include file **sgtty.h**:

```
struct sgttyb {
        char    sg_ispeed;      /* input speed */
        char    sg_ospeed;      /* output speed */
        char    sg_erase;       /* erase character */
        char    sg_kill;        /* kill character */
        int     sg_flags;       /* mode flags */
}
```

The flags, as defined in **sgtty.h**, are:

```
#define HUPCL      01
#define XTABS      02
#define LCASE      04
#define ECHO       010
#define CRMOD      020
#define RAW        040
#define ODDP       0100
#define EVENP      0200
#define ANYP       0300
#define NLDELAY    001400
#define TBDELAY    002000
#define CRDELAY    030000
#define VTDELAY    040000
#define BSDELAY    0100000

#define CR0        0
#define CR1        010000
#define CR2        020000
#define CR3        030000
#define NL0        0
#define NL1        000400
#define NL2        001000
#define NL3        001400
#define TAB0       0
#define TAB1       002000
#define NOAL       004000
#define FF0        0
#define FF1        040000
#define BS0        0
```

```
#define BS1          0100000
```

When the *stty*(2) command (*ioctl* **TIOCSETP**) is executed, the flags in the old **sgttyb** structure are mapped into their new equivalents in the **termio** structure. Then the **TCSETA** command is executed.

The following table shows the mapping between the old **sgttyb** flags and the current **termio** flags. Note that flags contained in the **termio** structure that are not mentioned below are cleared.

HUPCL	(if set) sets the **termio** HUPCL flag;
HUPCL	(if clear) clears the **termio** HUPCL flag;
XTABS	(if set) sets the **termio** TAB3 flag;
XTABS	(if clear) clears the **termio** TAB3 flag;
TBDELAY	(if set) sets the **termio** TAB1 flag;
TBDELAY	(if clear) clears the **termio** TAB1 flag;
LCASE	(if set) sets the **termio** IUCLC, OLCUC, and XCASE flags;
LCASE	(if clear) clears the **termio** IUCLC, OLCUC, and XCASE flags;
ECHO	(if set) sets the **termio** ECHO flag;
ECHO	(if clear) clears the **termio** ECHO flag;
NOAL	(if set) sets the **termio** ECHOK flag;
NOAL	(if clear) clears the **termio** ECHOK flag;
CRMOD	(if set) sets the **termio** ICRNL and ONLCR flags; also, if CR1 is set, the **termio** CR1 flag is set, and if CR2 is set, the **termio** ONOCR and CR2 flags are set;
CRMOD	(if clear) sets the **termio** ONLRET flag; also, if NL1 is set, the **termio** CR1 flag is set, and if NL2 is set, the **termio** CR2 flag is set;
RAW	(if set) sets the **termio** CS8 flag, and clears the **termio** ICRNL and IUCLC flags; also, default values of 6 characters and 0.1 seconds are assigned to MIN and TIME, respectively;
RAW	(if clear) sets the **termio** BRKINT, IGNPAR, ISTRIP, IXON, IXANY, OPOST, CS7, PARENB, ICANON, and ISIG flags; also, the default values control-D and null are assigned to the control characters EOF and EOL, respectively;
ODDP	(if set) if EVENP is also set, clears the **termio** INPCK flag; otherwise, sets the **termio** PARODD flag;
VTDELAY	(if set) sets the **termio** FFDLY flag;
VTDELAY	(if clear) clears the **termio** FFDLY flag;
BSDELAY	(if set) sets the **termio** BSDLY flag;
BSDELAY	(if clear) clears the **termio** BSDLY flag.

In addition, the **termio** CREAD bit is set, and, if the baud rate is 110, the CSTOPB bit is set.

When using **TIOCSETP**, the *ispeed* entry in the **sgttyb** structure is mapped into the appropriate speed in the **termio** CBAUD field. The *erase* and *kill* **sgttyb** entries are mapped into the **termio** erase and kill characters.

When the *gtty*(2) (*ioctl* **TIOCGETP**) command is executed, the *termio*(7) **TCGETA** command is first executed. The resulting **termio** structure is then mapped into the **sgttyb** structure, which is then returned to the user.

The following table shows how the **termio** flags are mapped into the old **sgttyb** structure. Note that all flags contained in the **sgttyb** structure that are not mentioned below are cleared.

HUPCL	(if set) sets the **sgttyb** HUPCL flag;
HUPCL	(if clear) clears the **sgttyb** HUPCL flag;
ICANON	(if set) sets the **sgttyb** RAW flag;
ICANON	(if clear) clears the **sgttyb** RAW flag;

XCASE	(if set) sets the **sgttyb** LCASE flag;
XCASE	(if clear) clears the **sgttyb** LCASE flag;
ECHO	(if set) sets the **sgttyb** ECHO flag;
ECHO	(if clear) clears the **sgttyb** ECHO flag;
ECHOK	(if set) sets the **sgttyb** NOAL flag;
ECHOK	(if clear) clears the **sgttyb** NOAL flag;
PARODD	(if set) sets the **sgttyb** ODDP flag;
PARODD	(if clear) clears the **sgttyb** ODDP flag;
INPCK	(if set) sets the **sgttyb** EVENP flag;
PARODD,	INPCK (if both clear) sets the **sgttyb** ODDP and EVENP flags;
ONLCR	(if set) sets the **sgttyb** CRMOD flag; also, if CR1 is set, the **sgttyb** CR1 flag is set, and if CR2 is set, the **sgttyb** CR2 flag is set;
ONLCR	(if clear) if CR1 is set, the **sgttyb** NL1 flag is set, and if CR2 is set, the **sgttyb** NL2 flag is set;
TAB3	(if set) sets the **sgttyb** XTABS flag;
TAB3	(if clear) clears the **sgttyb** XTABS flag;
TAB1	(if set) sets the **sgttyb** TBDELAY flag;
TAB1	(if clear) clears the **sgttyb** TBDELAY flag;
FFDLY	(if set) sets the **sgttyb** VTDELAY flag;
FFDLY	(if clear) clears the **sgttyb** VTDELAY flag;
BSDLY	(if set) sets the **sgttyb** BSDELAY flag;
BSDLY	(if clear) clears the **sgttyb** BSDELAY flag.

When using **TIOCGETP**, the **termio** CBAUD field is mapped into the *ispeed* and *ospeed* entries of the **sgttyb** structure. Also, the **termio** erase and kill characters are mapped into the *erase* and *kill* **sgttyb** entries.

Note that, since there is not a one-to-one mapping between the **sgttyb** and **termio** structures, unexpected results may occur when using the older **TIOCSETP** and **TIOCGETP** calls. Thus, the **TIOCSETP** and **TIOCGETP** calls should be replaced in all future code by the current equivalents, **TCSETA** and **TCGETA**, respectively.

SEE ALSO

stty(2), termio(7).

NAME

termio – general terminal interface

DESCRIPTION

All of the asynchronous communications ports use the same general interface, no matter what hardware is involved. The remainder of this section discusses the common features of this interface.

Opening a Terminal File

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, users' programs seldom open these files; they are opened by *getty* and become a user's standard input, output, and error files. When a process group leader without a controlling terminal opens a terminal that is not already a controlling terminal, that terminal becomes the controlling terminal for that process and the terminal's distinguished process group (*tty group ID*) is set to the process group of that process. The control terminal plays a special role in handling QUIT and INTERRUPT signals, as discussed below. It is also used in handling the process group control signals. The control terminal is inherited by a child process during a *fork*(2). A process can break this association by changing its process group using *setpgrp*(2). When the process group leader that acquired the controlling terminal terminates, the distinguished process group of the controlling terminal is set to zero (indicating no distinguished process group). This allows the terminal to be acquired as a controlling terminal by a new process group leader.

If the O_NDELAY bit (see *open*(2)) is clear, an *open* will block until the type of modem connection requested (see *modem*(2)) is completed. If the O_NDELAY bit is set, an *open* will succeed and return immediately without waiting for the modem connection requested to complete.

Reading Characters

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character input buffers become completely full, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. This limit is dependent on the particular implementation, but is at least 256. When the input limit is reached, all the saved characters are thrown away without notice.

Canonical Mode Input Processing (Erase and Kill Processing)

Normally, terminal input is processed in units of lines. A line is delimited by a new-line (ASCII LF) character, an end-of-file (ASCII EOT) character, or an end-of-line character. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, erase and kill processing is normally done. By default, the character # erases the last character typed, except that it will not erase beyond the beginning of the line. By default, the character **kills (deletes) the entire input line,** and optionally outputs a new-line character. Both these characters operate on a key-stroke basis, independently of any backspacing or tabbing that may have been done. Both the erase and kill characters may be entered literally by preceding them with the escape character (\). In this case the escape character is not read. The erase and kill characters may be changed.

Process Group Control During I/O

For those drivers that support process group control, if a process is not in the distinguished process group of its control terminal, and both the process group of the process and the control terminal are non-zero, the process is said to be a *background process*. Any attempts by a background process to read from its controlling terminal will cause the process group to be sent a SIGTTIN signal, which will normally cause the members of that process group to stop. If, however, a process is ignoring or holding the SIGTTIN signal, or (on systems that implement *vfork* separately from *fork*) has made a call to *vfork*(2) but not yet made a call to *exec*(2), then the process is instead returned an EIO error and no signal is sent to any process.

It is frequently undesirable for background processes to write to a terminal or to issue certain *ioctl* system calls which set tty parameters. In the following discussion, the response to writing should be assumed to also be the response to those *ioctl* calls. If the LTOSTOP bit is set, then a background process is prohibited from writing to that terminal. Attempts to write will cause the process group to be sent a SIGTTOU signal, which will normally cause the members of the process group to stop. If the LTOSTOP bit is not set, the process is ignoring or holding SIGTTOU signals, or is in the middle of the *vfork*(2) window described above, the process is allowed to write to the terminal.

Non-blocking Reads (O_NDELAY Flag)

If the O_NDELAY bit (see *open*(2) or *fcntl*(2)) is clear, the read request will complete whenever data is available. If the O_NDELAY bit is set, then the read request will complete, without blocking, in one of three ways: (1) If there is enough data available to satisfy the entire request, the read will complete successfully, having read all of the data, and return the number of bytes read; (2) If there is not enough data available to satisfy the entire request, the read will complete successfully, having read as much data as possible, and return the number of bytes it was able to read; (3) If there is no data available, the read will complete successfully, having read no data, and return a count of 0.

Special Characters

Certain characters have special functions on input. These functions and their default character values are summarized as follows:

INTR	(Rubout or ASCII DEL) generates an *interrupt* signal which is sent to all processes with the associated control terminal. Normally, each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location; see *signal*(2).
QUIT	(Control-| or ASCII FS) generates a *quit* signal. Its treatment is identical to the interrupt signal except that, unless a receiving process has made other arrangements, it will not only be terminated but a core image file (called **core**) will be created in the current working directory if the implementation supports core files.
ERASE	(#) erases the preceding character. It will not erase beyond the start of a line, as delimited by a NL, EOF, or EOL character.
KILL	(@) deletes the entire line, as delimited by a NL, EOF, or EOL character.
EOF	(Control-d or ASCII EOT) may be used to generate an end-of-file from a terminal. When received, all the characters waiting to be read are immediately passed to the program, without waiting for a new-line, and the EOF is discarded. Thus, if there are no characters waiting, which is to say the EOF occurred at the beginning of a line, zero characters will be passed back, which is the standard end-of-file indication.
NL	(ASCII LF) is the normal line delimiter. It can not be changed or escaped.
EOL	(ASCII NUL) is an additional line delimiter, like NL. It is not normally used.
STOP	(Control-s or ASCII DC3) can be used to temporarily suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read. While output is suspended, STOP characters are ignored and not read.
START	(Control-q or ASCII DC1) is used to resume output which has been suspended by a STOP character. While output is not suspended, START characters are ignored

and not read. The start/stop characters can not be changed or escaped.

The character values for INTR, QUIT, ERASE, KILL, EOF, and EOL may be changed to suit individual tastes. The ERASE, KILL, and EOF characters may be escaped by a preceding \ character, in which case no special function is done.

Writing Characters

When one or more characters are written, they are transmitted to the terminal as soon as previously-written characters have finished typing. Input characters are echoed by putting them in the output queue as they arrive. If a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold, the program is resumed.

Closing a Terminal File

When the last process to have a terminal device file open closes the file, the output data is drained before the close returns.

Modem Disconnect

When a modem disconnect is detected, a ***hang-up*** signal is sent to all processes that have this terminal as the control terminal. Unless other arrangements have been made, this signal causes the processes to terminate. If the hang-up signal is ignored, any subsequent read returns with an end-of-file indication. Thus, programs that read a terminal and test for end-of-file can terminate appropriately when hung up on.

Termio Structure

Several *ioctl*(2) system calls apply to terminal files. The primary calls use the following structure, defined in **<termio.h>**:

```
#define NCC     8
struct          termio {
    unsigned short   c_iflag;      /* input modes */
    unsigned short   c_oflag;      /* output modes */
    unsigned short   c_cflag;      /* control modes */
    unsigned short   c_lflag;      /* local modes */
    char             c_line;       /* line discipline */
    unsigned char    c_cc[NCC];    /* control chars */
};
```

The special control characters are defined by the array *c_cc*. The relative positions and initial values for each function are as follows:

0	VINTR	DEL
1	VQUIT	FS
2	VERASE	#
3	VKILL	@
4	VEOF	EOT
5	VEOL	NUL
6	reserved	
7	reserved	

Input Modes

The *c_iflag* field describes the basic terminal input control:

IGNBRK	0000001	Ignore break condition.
BRKINT	0000002	Signal interrupt on break.
IGNPAR	0000004	Ignore characters with parity errors.
PARMRK	0000010	Mark parity errors.
INPCK	0000020	Enable input parity check.
ISTRIP	0000040	Strip character.

INLCR	0000100	Map NL to CR on input.
IGNCR	0000200	Ignore CR.
ICRNL	0000400	Map CR to NL on input.
IUCLC	0001000	Map upper-case to lower-case on input.
IXON	0002000	Enable start/stop output control.
IXANY	0004000	Enable any character to restart output.
IXOFF	0010000	Enable start/stop input control.
IENQAK	0020000	Enable output pacing control.

If IGNBRK is set, the break condition (a character framing error with data all zeros) is ignored, that is, not put on the input queue and therefore not read by any process. Otherwise if BRKINT is set, the break condition will generate an interrupt signal and flush both the input and output queues. If IGNPAR is set, characters with other framing and parity errors are ignored.

If PARMRK is set, a character with a framing or parity error which is not ignored is read as the three-character sequence: 0377, 0, X, where X is the data of the character received in error. To avoid ambiguity in this case, if ISTRIP is not set, a valid character of 0377 is read as 0377, 0377. If PARMRK is not set, a framing or parity error which is not ignored is read as the character NUL (0).

If INPCK is set, input parity checking is enabled. If INPCK is not set, input parity checking is disabled. This allows output parity generation without input parity errors.

If ISTRIP is set, valid input characters are first stripped to 7-bits, otherwise all 8-bits are processed.

If INLCR is set, a received NL character is translated into a CR character. If IGNCR is set, a received CR character is ignored (not read). Otherwise if ICRNL is set, a received CR character is translated into a NL character.

If IUCLC is set, a received upper-case alphabetic character is translated into the corresponding lower-case character.

If IXON is set, start/stop output control is enabled. A received STOP character will suspend output and a received START character will restart output. All start/stop characters are ignored and not read. If IXANY is set, any input character, will restart output which has been suspended.

If IXOFF is set, start/stop input control is enabled. When the number of characters in the input queue exceeds a system defined value (high water mark), the system will transmit a STOP character. This should cause the attached device to stop transmitting data before the maximum allowed input has been reached. When enough characters have been read from the input queue that the number of remaining characters is less than another system defined value (low water mark), then the system will transmit a START character to cause input to be resumed. In order to avoid potential deadlock, the IXOFF bit is ignored in canonical mode whenever there is no line delimiter in the input buffer. In this case, the STOP character is not sent at the high water mark, but will be transmitted later if a delimiter is received. If all complete lines are read from the input queue leaving only a partial line with no line delimiter, the START character will be sent even though the number of characters may not be less than the low water mark. When ICANON is set and the input stream contains more characters between line delimiters than the high water mark allows, it is not guaranteed that IXOFF will prevent buffer overflow and data loss, since the STOP character may not be sent in time or at all.

If IENQAK is set, the system will transmit ASCII ENQ after every 80 characters sent and then wait until the terminal responds with ASCII ACK. The terminal will respond in this way when it has sufficiently emptied its buffer. If the terminal does not respond after 5 seconds, the system will resume transmission anyway. The ASCII ACK that the terminal sends will not get entered into the input queue if it was sent in response to ASCII ENQ.

The initial input control value is all-bits-clear.

Output Modes

The *c_oflag* field specifies the system treatment of output:

OPOST	0000001	Postprocess output.
OLCUC	0000002	Map lower case to upper on output.
ONLCR	0000004	Map NL to CR-NL on output.
OCRNL	0000010	Map CR to NL on output.
ONOCR	0000020	No CR output at column 0.
ONLRET	0000040	NL performs CR function.
OFILL	0000100	Use fill characters for delay.
OFDEL	0000200	Fill is DEL, else NUL.
NLDLY	0000400	Select new-line delays:
NL0	0	
NL1	0000400	
CRDLY	0003000	Select carriage-return delays:
CR0	0	
CR1	0001000	
CR2	0002000	
CR3	0003000	
TABDLY	0014000	Select horizontal-tab delays:
TAB0	0	
TAB1	0004000	
TAB2	0010000	
TAB3	0014000	Expand tabs to spaces.
BSDLY	0020000	Select backspace delays:
BS0	0	
BS1	0020000	
VTDLY	0040000	Select vertical-tab delays:
VT0	0	
VT1	0040000	
FFDLY	0100000	Select form-feed delays:
FF0	0	
FF1	0100000	

If OPOST is set, output characters are post-processed as indicated by the remaining flags, otherwise characters are transmitted without change.

If OLCUC is set, a lower-case alphabetic character is transmitted as the corresponding upper-case character. This function is often used in conjunction with IUCLC.

If ONLCR is set, the NL character is transmitted as the CR-NL character pair. If OCRNL is set, the CR character is transmitted as the NL character. If ONOCR is set, no CR character is transmitted when at column 0 (first position). If ONLRET is set, the NL character is assumed to do the carriage-return function; the column pointer will be set to 0 and the delays specified for CR will be used. Otherwise the NL character is assumed to do just the line-feed function; the column pointer will remain unchanged. The column pointer is also set to 0 if the CR character is actually transmitted.

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay. If OFILL is set, fill characters will be transmitted for delay instead of a timed delay. This is useful for high baud rate terminals which need only a minimal delay. If OFDEL is set, the fill character is DEL, otherwise NUL.

If a form-feed or vertical-tab delay is specified, it lasts for about 2 seconds.

New-line delay lasts about 0.10 seconds. If ONLRET is set, the carriage-return delays are used instead of the new-line delays. If OFILL is set, two fill characters will be transmitted.

Carriage-return delay type 1 is dependent on the current column position, type 2 is about 0.10 seconds, and type 3 is about 0.15 seconds. If OFILL is set, delay type 1 transmits two fill characters, and type 2, four fill characters.

Horizontal-tab delay type 1 is dependent on the current column position. Type 2 is about 0.10 seconds. Type 3 specifies that tabs are to be expanded into spaces. If OFILL is set, two fill characters will be transmitted for any delay.

Backspace delay lasts about 0.05 seconds. If OFILL is set, one fill character will be transmitted.

The actual delays depend on line speed and system load.

The initial output control value is all bits clear.

Control Modes

The *c_cflag* field describes the hardware control of the terminal:

CBAUD	0000037	Baud rate:
B0	0	Hang up
B50	0000001	50 baud
B75	0000002	75 baud
B110	0000003	110 baud
B134	0000004	134.5 baud
B150	0000005	150 baud
B200	0000006	200 baud
B300	0000007	300 baud
B600	0000010	600 baud
B900	0000011	900 baud
B1200	0000012	1200 baud
B1800	0000013	1800 baud
B2400	0000014	2400 baud
B3600	0000015	3600 baud
B4800	0000016	4800 baud
B7200	0000017	7200 baud
B9600	0000020	9600 baud
B19200	0000021	19200 baud
B38400	0000022	38400 baud
EXTA	0000036	External A
EXTB	0000037	External B
CSIZE	0000140	Character size:
CS5	0	5 bits
CS6	0000040	6 bits
CS7	0000100	7 bits
CS8	0000140	8 bits
CSTOPB	0000200	Send two stop bits, else one.
CREAD	0000400	Enable receiver.
PARENB	0001000	Parity enable.
PARODD	0002000	Odd parity, else even.
HUPCL	0004000	Hang up on last close.
CLOCAL	0010000	Local line, else dial-up.

The CBAUD bits specify the baud rate. The zero baud rate, B0, is used to hang up the connection. If B0 is specified, the modem control lines (see *modem*(7)) will cease to be asserted.

Normally, this will disconnect the line. For any particular hardware, impossible speed changes are ignored.

The CSIZE bits specify the character size in bits for both transmission and reception. This size does not include the parity bit, if any. If CSTOPB is set, two stop bits are used, otherwise one stop bit. For example, at 110 baud, two stops bits are required.

If PARENB is set, parity generation and detection is enabled and a parity bit is added to each character. If parity is enabled, the PARODD flag specifies odd parity if set, otherwise even parity is used.

If CREAD is set, the receiver is enabled. Otherwise no characters will be received.

The specific effects of the HUPCL and CLOCAL bits depend on the mode and type of the modem control in effect. See *modem*(7) for the details.

If HUPCL is set, the modem control lines for the port will be disconnected when the last process with the port open closes it or terminates.

If CLOCAL is set, a connection does not depend on the state of the modem status lines.

If the CLOCAL bit has been set, an *open* will return immediately without waiting for the connection. For those files on which the connection has not been established or has been lost, and for which the CLOCAL bit is not set, both *read* and *write* will return a zero character count. For *read*, this is equivalent to an end-of-file condition.

The initial hardware control value after open is B300, CS8, CREAD, HUPCL.

Local Modes

The *c_lflag* field of the argument structure is used by the line discipline to control terminal functions. The basic line discipline (0) provides the following:

ISIG	0000001	Enable signals.
ICANON	0000002	Canonical input (erase and kill processing).
XCASE	0000004	Canonical upper/lower presentation.
ECHO	0000010	Enable echo.
ECHOE	0000020	Echo erase character as BS-SP-BS.
ECHOK	0000040	Echo NL after kill character.
ECHONL	0000100	Echo NL.
NOFLSH	0000200	Disable flush after interrupt, quit, or suspend.

If ISIG is set, each input character is checked against the special control characters INTR, QUIT, and the suspend characters (see Process Group Control below). If an input character matches one of these control characters, the function associated with that character is performed. If ISIG is not set, no checking is done. Thus these special input functions are possible only if ISIG is set. These functions may be disabled individually by changing the value of the control character to an unlikely or impossible value (e.g., 0377).

If ICANON is set, canonical processing is enabled. This enables the erase and kill edit functions, and the assembly of input characters into lines delimited by NL, EOF, and EOL. If ICANON is not set, read requests are satisfied directly from the input queue. A read will not be satisfied until at least MIN characters have been received or the timeout value TIME has expired between characters. (See **Non-canonical Mode Input Processing (MIN/TIME Interaction**) below). This allows fast bursts of input to be read efficiently while still allowing single character input. The MIN and TIME values are stored in the position for the EOF and EOL characters, respectively. The time value represents tenths of seconds.

If XCASE is set, and if ICANON is set, an upper-case letter is accepted on input by preceding it with a \ character, and is output preceded by a \ character. In this mode, the following escape sequences are generated on output and accepted on input:

for:	*use*:
`	\'
|	\!
~	\^
{	\(
}	\)
\	\\

For example, **A** is input as **\a**, **\n** as **\\n**, and **\N** as **\\\n**.

If ECHO is set, characters are echoed as received.

When ICANON is set, the following echo functions are possible. If ECHO and ECHOE are set, the erase character is echoed as ASCII BS SP BS, which will clear the last character from a CRT screen. If ECHOE is set and ECHO is not set, the erase character is echoed as ASCII SP BS. If ECHOK is set, the NL character will be echoed after the kill character to emphasize that the line will be deleted. Note that an escape character preceding the erase or kill character removes any special function. If ECHONL is set, the NL character will be echoed even if ECHO is not set. This is useful for terminals set to local echo (so-called half duplex). Unless escaped, the EOF character is not echoed. Because EOT is the default EOF character, this prevents terminals that respond to EOT from hanging up.

If NOFLSH is set, the normal flush of the input and output queues associated with the quit, interrupt, and suspend characters will not be done.

The initial line-discipline control value is all bits clear.

Unless otherwise noted for a specific *ioctl* command, the *ioctl*s are restricted from use by background processes. An attempt to issue an *ioctl* from a background process will cause the process to block and may cause a SIGTTOU signal to be sent to the process group.

Non-canonical Mode Input Processing (MIN/TIME Interaction)

The MIN and TIME values are stored in c_cc[VMIN] and c_cc[VTIME], respectively, and are used when ICANON is clear. MIN represents the minimum number of characters that should be received when the read is satisfied (i.e., the characters are returned to the user). TIME is a timer of 0.10 second granularity that is used to timeout burst and short term data transmissions. The four possible values for MIN and TIME and their interactions are described below.

A. MIN > 0, TIME > 0

In this case TIME serves as an intercharacter timer and is activated after the first character is received. Since it is an intercharacter timer, it is reset after a character is received. The interaction between MIN and TIME is as follows: as soon as one character is received, the intercharacter timer is started. If MIN characters are received before the intercharacter timer expires (remember that the timer is reset upon receipt of each character), the read is satisfied. If the timer expires before MIN characters are received, the characters received to that point are returned to the user. Note that if TIME expires, at least one character will be returned because the timer would not have been enabled unless a character was received. In this case (MIN > 0, TIME > 0) the read will sleep until the MIN and TIME mechanisms are activated by the receipt of the first character.

B. MIN > 0, TIME = 0

In this case, since the value of TIME is zero, the timer plays no role and only MIN is significant. A pending read is not satisfied until MIN characters are received after any previous read completes (i.e., the pending read will sleep until MIN characters are received). A program that uses this case to handle record-based terminal I/O may block.

C. MIN = 0, TIME > 0

In this case, since MIN = 0, TIME no longer represents an intercharacter timer. It now serves as a read timer that is activated as soon as the read system call is processed. A read is satisfied as soon as a single character is received or the read timer expires. Note that in this case if the timer expires, no character will be returned. If the timer does not expire, the only way the read can be satisfied is if a character is received. Note that in this case the read will not sleep indefinitely waiting for a character. If no character is received within TIME * 0.10 seconds after the read is initiated, the read will return with zero characters.

D. MIN = 0, TIME = 0

In this case return is immediate. If characters are present, they will be returned to the user.

Some points to note about MIN and TIME:

1. In the above explanations one may notice that the interactions of MIN and TIME are not symmetric. For example, when MIN > 0 and TIME = 0, TIME has no effect. However, in the opposite case where MIN = 0 and TIME > 0, both MIN and TIME play a role in that MIN is satisfied with the receipt of a single character.
2. Also note that in case A (MIN > 0, TIME > 0), TIME represents an intercharacter timer while in case C (MIN = 0, TIME > 0), TIME represents a read timer.

These two points highlight the dual purpose of the MIN/TIME feature. Cases A and B, where MIN > 0, exist to handle burst mode activity (e.g., file transfer programs) where a program would like to process at least MIN characters at a time. In case A, the intercharacter timer is activated by a user as a safety measure while in case B it is turned off.

Cases C and D exist to handle single character timed transfers. These cases are readily adaptable to screen based applications that need to know if a character is present in the input queue before refreshing the screen. In case C the read is timed, while in case D it is not.

Another important note is that MIN is always just a minimum. It does not denote a record length. That is, if a program does a read of 20 bytes and MIN is 10 and 25 characters are present, 20 characters will be returned to the user. If the program had requested all characters, then all 25 characters would be returned to the user.

Primary TERMIO IOCTL Commands

The primary *ioctl*(2) system calls have the form:

```
ioctl (fildes, command, arg)
struct termio *arg;
```

The commands using this form are:

TCGETA	Get the parameters associated with the terminal and store in the *termio* structure referenced by *arg*. This command is allowed from a background process; however, the information may be subsequently changed by a foreground process.
TCSETA	Set the parameters associated with the terminal from the structure referenced by *arg*. The change is immediate.
TCSETAW	Wait for the output to drain before setting the new parameters. This form should be used when changing parameters that will affect output.
TCSETAF	Wait for the output to drain, then flush the input queue and set the new parameters.

Additional *ioctl*(2) calls have the form:

ioctl (fildes, command, arg)
int arg;

The commands using this form are:

TCSBRK	Wait for the output to drain. If *arg* is 0, then send a break (zero bits for at least 0.25 seconds).
TCXONC	Start/stop control. If *arg* is 0, suspend output; if 1, restart suspended output.
TCFLSH	If *arg* is 0, flush the input queue; if 1, flush the output queue; if 2, flush both the input and output queues.

The following command has the form:

ioctl (fildes, command, arg)
long *arg;

FIONREAD Returns in the long integer whose address is *arg* the number of characters immediately readable from the terminal device file. This command is allowed from a background process; however, the data itself may not be read from a background process.

System Asynchronous I/O

The following commands have the form:

ioctl (fildes, command, arg)
int *arg;

FIOSSAIOSTAT If the integer whose address is *arg* is non-zero, system asynchronous I/O is enabled. That is, enable SIGIO to be sent to the process currently designated with FIOSSAIOOWN (see below) whenever the terminal device file status changes from "no read data available" to "read data available". If no process has been designated with FIOSSAIOOWN, then enable SIGIO to be sent to the first process to open the terminal device file.

If the designated process has exited, the SIGIO signal will not be sent to any process.

If the integer whose address is *arg* is 0, system asynchronous I/O is disabled.

FIOGSAIOSTAT The integer whose address is *arg* is set to 1, if system asynchronous I/O is enabled. Otherwise, the integer whose address is *arg* is set to 0.

FIOSSAIOOWN Set process ID to receive the SIGIO signals with system asynchronous I/O to the value of the integer whose address is *arg*. The super-user may designate that any process receive the SIGIO signals. If the request is not made by the super-user, the calling process is only allowed to designate that itself or another process whose real or saved effective user ID matches its real or effective user ID, or a process which is a descendant of the calling process, receive the SIGIO signals. If no process can be found corresponding to that specified by the integer whose address is *arg*, the call will fail, with errno set to ESRCH. If the request is not made by the super-user, and the calling process attempts to designate a process other than itself or another process whose real or saved effective user ID matches its real or effective user ID, or a process which is not a descendant of the calling process, the call will fail, with errno set to EPERM.

If the designated process subsequently exits, the SIGIO signal will not be sent to any process.

The default on open of a terminal device file is that the process performing the open is set to receive the SIGIO signals.

FIOGSAIOOWN The integer whose address is *arg* is set to the process ID designated to receive SIGIO signals.

Non-blocking I/O

FIOSNBIO

If the integer whose address is *arg* is non-zero, non-blocking I/O is enabled. That is, subsequent reads and writes to the terminal device file will be handled in a non-blocking manner (see below). If the integer whose address is *arg* is 0, non-blocking I/O is disabled.

For reads, non-blocking I/O will prevent all read requests to that device file from blocking, whether the requests succeed or fail. Such a read request will complete in one of three ways: (1) If there is enough data available to satisfy the entire request, the read will complete successfully, having read all of the data, and return the number of bytes read; (2) If there is not enough data available to satisfy the entire request, the read will complete successfully, having read as much data as possible, and return the number of bytes it was able to read; (3) If there is no data available, the read will fail and *errno* will be set to EWOULDBLOCK.

For writes, non-blocking I/O will prevent all write requests to that device file from blocking, whether the requests succeed or fail. Such a write request will complete in one of three ways: (1) If there is enough space available in the system to buffer all the data, the write will complete successfully, having written out all of the data, and return the number of bytes written; (2) If there is not enough space in the buffer to write out the entire request, the write will complete successfully, having written as much data as possible, and return the number of bytes it was able to write; (3) If there is no space in the buffer, the write will fail and *errno* will be set to EWOULDBLOCK.

To prohibit non-blocking I/O from interfering with the O_NDELAY flag (see *open*(2) and *fcntl*(2)), the functionality of O_NDELAY always supercedes the functionality of non-blocking I/O. This means that if O_NDELAY is set, the driver will perform read requests in accordance with the definition of O_NDELAY. When O_NDELAY is not set, the definition of non-blocking I/O applies.

The default on open of a terminal device file is that non-blocking I/O is disabled.

FIOGNBIO

The integer whose address is *arg* is set to 1, if non-blocking I/O is enabled. Otherwise, the integer whose address is *arg* is set to 0.

Process Group Control

Some implementations support process group control. The following structure, used with process group control, is defined in <**bsdtty.h**>:

```
struct  ltchars        {
        unsigned char  t_suspc;    /* stop process character */
        unsigned char  t_dsuspc;   /* delayed stop process character */
        unsigned char  t_rprntc;   /* reserved; must be 0377 */
        unsigned char   t_flushc;     /* reserved; must be 0377 */
        unsigned char   t_werasc;     /* reserved; must be 0377 */
        unsigned char   t_lnextc;     /* reserved; must be 0377 */
};
```

The initial value for all these characters is 0377 which causes them to be disabled. The meaning for each character is as follows:

t_suspc is used to suspend the currently active process group. A *suspend* signal (SIGTSTP) is sent to all processses in the currently active process group. Normally, each process is forced to stop, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location; see *signal*(2). When enabled, the typical value for this character is control-z or ASCII SUB.

t_dsuspc functions like **t_suspc** does, except that the *suspend* signal is sent when a process reads the character rather than when it is typed. When enabled, the typical value for this character is control-y or ASCII EM.

Attempts to set any of the reserved characters to a value other than 0377 will cause the *ioctl* to be returned with an EINVAL error code and no change will occur.

The *ioctl*(2) system calls which use the above structure have the form:

ioctl (fildes, command, arg)
struct ltchars *arg;

The commands using this form are:

TIOCGLTC Get the process group control characters and store in the *ltchars* structure whose address is *arg*. This command is allowed from a background process; however, the information may be subsequently changed by a foreground process.

TIOCSLTC Set the process group control characters from the structure whose address is *arg*.

Additional process group control *ioctl*(2) commands have the form:

ioctl (fildes, command, arg)
unsigned int *arg;

The commands using this form are:

TIOCGPGRP Returns in the integer whose address is *arg* the currently active process group associated with the terminal. This command is allowed from a background process; however, the information may be subsequently changed by a foreground process.

TIOCSPGRP Sets the currently active process group associated with the terminal to be the one whose address is *arg*.

The following condition must be met for the request to succeed; otherwise, the error [EINVAL] is returned:

The new process group value must be in the range of valid process group ID values, or it must be zero ("no process group").

One or more of the following conditions must be met for the request to succeed; otherwise, the error [EPERM] is returned:

The effective user ID of the current process is super-user.

The process group ID whose address is *arg* matches the saved process group ID of the calling process.

If any processes exist with a process ID or process group ID that is the same as the process group whose address is *arg*, those processes must have the same real or saved user ID as the real or effective user ID of the calling process, or be descendants of the calling process.

TIOCLGET
Get the process group control mode word and store in the int pointed to by *arg*. This command is allowed from a background process; however, the information may be subsequently changed by a foreground process.

TIOCLSET
Set the process group control mode word from the int pointed to by *arg*.

TIOCLBIS
Use the int pointed to by *arg* as a mask of bits to set in the process group control mode word.

TIOCLBIC
Use the int pointed to by *arg* as a mask of bits to clear in the process group control mode word.

The following bit is defined in the process group control mode word:

LTOSTOP	Send SIGTTOU for background write or ioctl.

If the LTOSTOP bit is set, then an attempt by a process which is not in the distinguished process group to write data or to issue an *ioctl*(2) system call which will change the tty state will cause a SIGTTOU signal to be sent to that process group.

WARNINGS

Because various HP-UX implementations use non-serial interfaces which look like terminals (e.g. internal CRTs) or 'smart cards' which cannot implement the capabilities described above exactly, not all the systems can meet the standard stated above exactly. Each implementation is required to state any deviations from the standard as part of its system specific documentation.

FIOSSAIOSTAT is similar to 4.2 BSD FIOASYNC, with the addition of provisions for security. FIOGSAIOSTAT is of HP origin, complements FIOSSAIOSTAT, and allows saving and restoring system asynchronous I/O TTY state for BSD style job control. FIOSSAIOOWN is similar to 4.2 BSD FIOSETOWN, with the addition of provisions for security. FIOGSAIOOWN is similar to 4.2 BSD FIOGETOWN. Note also the difference that the 4.2 BSD version of this functionality used process groups, while the HPUX version only uses processes. FIOSNBIO is the same as 4.2 BSD FIONBIO, except that it does not interfere with the ATT O_NDELAY *open* and *fcntl* flag. FIOGNBIO is of HP origin, complements FIOSNBIO, and allows saving and restoring non-blocking I/O TTY state for BSD style job control.

HARDWARE DEPENDENCIES

Series 200, 300:

Data loss may occur with HP 98626/98644 serial interfaces if the effective combined data rate for all installed serial interfaces exceeds 2400 baud (for example, two interfaces running at 1200 baud and a third at 300 baud is equivalent to 2700 baud combined).

The *c_iflag* field parameter IXANY (enable any character to restart output) is not supported by the HP 98628B interface card.

The *c_iflag* field parameter IENQAK (enable output pacing control) is not supported.

Timed delays are not supported.

The HP 98628B interface does not support the following baud rates: 900, 7200, 38400.

The *c_lflag* field parameter XCASE is not supported.

Series 200, 300, 500

Job control and asynchronous I/O are not supported.

Series 500:

The baud rate of 38400 is not supported by the RS-232 interface.

HP27140 Six-Channel Modem Multiplexer:

Timed output delays (as opposed to fill-character delays) are not supported.

The XCASE flag is not supported.

These baud rates are not supported: 200, 38400, EXTA, and EXTB.

HP27128 Asynchronous Serial Interface, HP27130 Eight-Channel Multiplexer:

These baud rates are not supported: 200, 38400, EXTA, and EXTB.

There is no support for tab expansion, case mapping, or output delays for control characters.

The line kill character is always echoes as <backslash><CR><LF>, so the ECHOK flag is not setable, and will always have the same state as the ECHO flag.

When type-ahead limit is reached, input is not flushed, but further input is simply ignored.

The PARMRK flag is not supported.

The echoing of carriage-return and new-line characters may not be quite as expected in the more obscure driver configurations.

The echoing of the EOF character is not suppressed.

The ONLRET, ONOCR, and OCRNL flags are not supported.

The VMIN and VTIME parameters for raw terminal input are not supported.

The ECHONL flag is not supported.

When ECHOE is set and ECHO is clear, a <SP><BS> is not echoes for the erase character.

(27130 only) The CLOCAL flag is permanently set.

(27128 only) The default setting of baud rate, bits per character, parity, and CLOCAL bit are determined by the switches on the interface.

(27128 only) The "direct connect" cable (female connector) does not contain a Data Carrier Detect line, so a hangup signal will be sent if the CLOCAL flag is cleared when this cable is being used.

Model 520 Console, HP98700 Terminal, Pseudo Terminal (pty): Since these devices do not deal with real asynchronous serial data links, the following flags have no effect: IGNPAR, PARMRK, INPCK, IXOFF, IENQAK, CBAUD, CSIZE, CSTOPB, PARENB, PARODD, HUPCL, and CLOCAL.

Series 800:

ENQ/ACK protocol and IENQAK bit are not supported.

Timed output delays are not directly supported. If used, an appropriate number of fill characters (based on the current baud rate) is output. The total time to output the fill characters is at least as long as the time requested.

The baud rate of 38400 is not supported by the RS-232 interface.

The system specified input flow control values are as follows: low water mark is 60, high water mark is 180, and maximum allowed input is 512.

AUTHOR

Termio was developed by HP, AT&T, and the University of California, Berkeley.

FILES

/dev/console
/dev/tty*

SEE ALSO

mknod(1M), stty(1), fork(2), ioctl(2), setpgrp(2), signal(2), stty(2), blmode(3C), sttyV6(7), tty(7), modem(7).

NAME

tty – controlling terminal interface

DESCRIPTION

The file **/dev/tty** is, in each process, a synonym for the control terminal associated with the process group of that process, if any. It is useful for programs or shell sequences that wish to be sure of writing messages on the terminal no matter how output has been redirected. It can also be used for programs that demand the name of a file for output, when typed output is desired and it is tiresome to find out what terminal is currently in use.

FILES

/dev/tty
/dev/tty*

SEE ALSO

termio(7).

MANUAL COMMENT CARD

HP-UX Reference

HP Part Number 09000-90010 4/87

Please help us improve this manual. Circle the numbers in the following statement that best indicate how useful you found this manual. Then add any further comments in the spaces below. **In appreciation of your time, we will enter your name in a quarterly drawing for an HP calculator.** Thank you.

The information in this manual:

Is poorly organized	1	2	3	4	5	Is well organized
Is hard to find	1	2	3	4	5	Is easy to find
Doesn't cover enough	1	2	3	4	5	Covers everything
Has too many errors	1	2	3	4	5	Is very accurate

Particular pages with errors? ____________________

Comments: ____________________

Name: ____________________

Job Title: ____________________

Company: ____________________

Address: ____________________

☐ Check here if you wish a reply.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Fort Collins Systems Division
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525

HEWLETT
PACKARD

HP Part Number
09000-90010

Microfiche No. 09000-99010
Printed in U.S.A.   4/87

09000-90665
For Internal Use Only